Orpheus 4™

User's Manual
Volume 1

TurboPower Software Company
Colorado Springs, CO

www.turbopower.com

First Edition September 1997
Second Edition May 1998
Third Edition December 1998
Fourth Edition June 2001

License Agreement

Table Of Contents

Volume 1

Volume 2

Chapter 1: Introduction

Orpheus is a collection of native, 32-bit Visual Component Library (VCL) components that dramatically extend the user interface design capabilities of Borland Delphi and C++Builder.

Orpheus controls are optimized and fully integrated with the Delphi and C++Builder development environments. Since the VCL components compile directly into your applications there's nothing extra to distribute and install with your programs unless you decide to use run-time packages. Best of all, Orpheus' complete source code is included so you can see exactly how it works.

Orpheus offers a very wide range of components. A high-level overview of the main sections of the library follows.

LookOut Bar

The LookOut Bar is a navigational icon bar that is both intuitive and instantly recognizable to users. Icons representing various facets of program functionality can be visually and logically divided by folder groups. Folders smoothly scroll into view along with their associated icons with a simple mouse click. Comprehensive icon handling features including painting, highlighting, animations, size, and sound are supported.

Folders can now serve as component containers. Set the folder's FolderType property to ftContainer and you can drop any visual component onto a folder's client area.

ReportView

The ReportView is a major new control in Orpheus that displays detailed or summarized data in a collapsible, configurable grid format. The ReportView lets you define and switch among various Views of your data. No-code sorting lets you sort columns of data with just a mouse click. In addition, data can be aggregated into collapsible groups with automatic data summarization. Orpheus includes three different ReportViews to meet your specific programming needs.

Entry Fields

Entry field components provide validated data entry, automatic internationalization, easy-to-use picture masks, and numerous other properties and methods. Most of each component's functionality is set at design time using the Object Inspector, but you can customize nearly every aspect of the entry field's behavior at run time, too.

New in Orpheus 4 is the O32FlexEdit control. It provides a multi-line edit control with a number of features like advanced border control, context sensitive line display and input validation via built-in support for the new validators. See “Chapter 13: Validators” on page 453for more information on the new validators.

State control

Orpheus provides components that can store and retrieve state information for your programs automatically. With these components, your applications can maintain state information between executions, restoring selected program defaults easily. The Orpheus state components let you choose to store state information in the Registry, an INI file, or in a medium of your own devising.

O32XMLStore component extends XML file support to the State control components. Now you can store your program’s state information in XML format.

Table

The table component displays and edits data in rows and columns. You can edit cells not only with normal VCL edit controls, but also with Orpheus picture mask controls, numeric controls, check boxes, and combo boxes. Bitmaps and labels can be displayed on the grid as well. The appearance of cells (type, font, color, width, height, alignment, hidden state, and read-only state) is fully customizable. The table can hold up to 16,000 columns by 2 billion rows. Data for the table is obtained from a user-defined event handler, allowing you to supply the data from any source. In addition, there is a data-aware version of the table enabling you to display and edit data from a data source.

Text editors

The text editor component provides a sophisticated text editor with real-time word wrap, bookmarks, search/replace, multi-step undo/redo, and 16MB capacity. The text stream is managed as a linked list of paragraphs of up to 32KB each. The text file editor adds text file I/O to the editor component. A data-aware version of the text editor is also provided to allow editing of database memo fields.

The text editors have been enhanced for Orpheus 4. New features include syntax highlighting, adjustable margin lines, and line numbers.

Combo boxes

Orpheus includes a variety of combo boxes that are automatically stocked with commonly required data such as all the fonts on a user’s machine. Combo boxes are also provided for navigating the disk’s directory structure, choosing printers, files, file associations, database aliases, and database tables. These combo boxes can also have “history”, which automatically lists the most recently accessed items at the top of the drop-down list.

Viewers

The file viewer uses a virtual memory management scheme to display text or binary files of any size. Orpheus viewers also include a hex display mode for displaying binary data. In addition, the Viewers support searching, scroll bars, and bookmarks. The Orpheus text file viewer component can read any text file into a memory collection of text lines and supports scrolling and displaying the file.

Lists

Orpheus provides a variety of list types that fill almost any list requirement. The search list is an incremental search list box. The check list box lets you to display a check mark or an ‘X’ beside selected items. The column list displays the values of a specified database field in a column.

Virtual list box

Virtual list boxes use event handlers to supply the display string, color, selection state, and protection state of each item in the list. Because of this design, you can theoretically display up to 2 billion items, with the item data stored in a memory array, a tree, a linked list, a disk file, or some other data structure. The list box supports multiple selection, protected items, and any number of item colors.

Tabbed notebook

The tabbed notebook component provides for multi-page forms using single-row or multi–row 3D tabs. The notebook also features the ability to have tabs on any edge of the component. Tabs are automatically sized and placed based on the desired tab orientation at design time or run time. An owner draw option gives you complete control over the appearance of the tabs. An advanced feature of the Orpheus notebook can help you conserve Windows resources. When activated, this feature can automatically create and destroy windowed controls when a particular page is activated and deactivated. Following is a list of features new to Orpheus 4:

- Tab text orientation: Rotate the tab text so that users don't have to lean their head to the side to read the tab labels.
- Tab row locking: Prevent the tab rows from re-arranging when the user selects tabs from different rows.
- Delphi component palette emulation: Creates a spinner to scroll tabs horizontally when there are more tabs than can be displayed in the number of allowed rows.

Spinner

The Orpheus spinner component implements a spinner control. It allows you to offer an alternative to normal data entry by allowing the user to adjust an associated control's value by clicking on the spin buttons.

Buttons

The ButtonHeader component allows you to select sections of a header. This could be used, for example, above a list of related items to specify the sort parameters for the list. The AttachedButton is a TBitButton that you can attach to other components. The SpeedButton is an alternative to the standard TSpeedbutton control that adds the capability to auto-repeat while the button is pressed, a "flat" and "transparent" modes.

Labels

The label components provide capabilities similar to the standard TLabel component, but go further. TOvcLabel lets you create eye-catching captions for your applications. It expands on the VCL TLabel with special shading, color, and highlight capabilities. The rotated label allows you to place a label on a form at any angle and size using a TrueType font. The picture label allows you to display data in accordance with a picture mask that you set. A data-aware version of the picture label connects to a data source and displays data from a field in a database. The display label is a data-aware label that connects to a data source and displays the DisplayLabel property of a TField. The URL component is a label component that emulates a hyperlink on a Web page or an e-mail address.

Date and time components

TOvcCalendar allows you to view an entire month at a time. It supports the years from 1600 to 3999. A data-aware version of the calendar connects to a data source and allows visual manipulation of the database's date or date/time fields. TOvcClock is an analog clock that can display the system's current time, updated every second, or a static time to display schedules, meeting times, and so on. Both the clock face and the clock hands are customizable. Following is a list of features new to Orpheus 4:

- Digital Clock Display: select a led-style digital display for the clock.
- Advanced timer functionality.

Miscellaneous components

There are many miscellaneous components in Orpheus that defy categorization. Some of these have been requested by Orpheus customers, and others were invented to meet a particular need of TurboPower's. For example, inside Orpheus you will also find:

- A search edit component that provides an incremental search mechanism similar to the search capability in the Windows on-line help index.
- A database index selection component that provides a drop-down list of the indexes available in an attached dataset.
- An enhanced meter component with a three dimensional, two-color progress bar for reporting the status of long events. In addition, the meter in Orpheus lets you perform dazzling effects using your own bitmaps for the used and unused portions of the display.
- A flexible MRU component for automatically adding a list of “most recently used” documents to your File menu or elsewhere in your application.

Also included are form splitters that allow you to split a window into two sections, timer pools that use a single Windows timer to manage dozens of application timer events, and a data transfer component that provides the equivalent of the ObjectWindows Library transfer mechanism for Delphi and C++Builder forms.

System Requirements

Much of Orpheus 4 has been updated to take advantage of newer, 32-bit features of Delphi. As a result, Orpheus 4 is only supported in Delphi 3 and above and C++Builder 3 and abouve. For optimal performance you must have the following hardware and software:

1. A computer capable of running Microsoft Windows 95, Windows 98, Windows NT, or Windows 2000. At least 32MB of RAM is recommended.
2. Borland Delphi Version 3, 4, 5, or 6 or C++Builder 3, 4 or, 5.
3. An installation of all Orpheus files and example programs for one compiler requires about 10MB of disk space.

Installation

Orpheus can be installed directly from the CD-ROM.

The setup program

Insert the TurboPower Product Suite CD-ROM and follow the instructions presented by the setup program.

SETUP installs Orpheus in C:\Program Files\Orpheus by default. You can specify a different directory if desired. You can choose a full or partial installation. Full installation is recommended, but if you need to conserve disk space, use custom installation to install only selected portions of Orpheus.

SETUP installs to your start menu under Programs\TurboPower\Orpheus by default (a different name can be specified, if desired).

Installing for multiple compiler versions

Orpheus supports all Delphi compilers version 3 and above and all version of C++Builder 3 and above. However, the compiled file format is different for each version. The Orpheus setup program allows you to select compiler support for each of the different version of Delphi or C++Builder.

By default, only the installed compilers will be selected when Orpheus is installed for the first time. If Orpheus is being re-installed or upgraded, only support for the previously supported compilers will be selected. If you have installed a new compiler and wish to install Orpheus support for that compiler, you must explicitly select support for that compiler.

Installing into C++Builder

Orpheus can be used with either Delphi or C++Builder. If you install the C++Builder help file and examples then you will find the help file in the ORPHEUS \HELP\CBUILDER directory and the examples in ORPHEUS\EXAMPLES\CBUILDER.

Header and object files for each version of C++Builder are installed into the \HPP* directory located in the Orpheus root directory. For example, headers filed for C++Builder 5 will be located in the \HPP5 directory. If you need to generate header and object files to support other C++Builder compilers, you can either re-install Orpheus and select the new compiler, or use the DCC32.EXE program to compile the OVCREG.PAS and OVCREGDB.PAS files using “-jphn” as the command line options.

Component installation

Delphi and C++Builder packages

To avoid version conflicts with applications using different versions of the Orpheus packages, each version of Orpheus comes with packages using slightly different names. This allows you to use the same Orpheus installation with multiple versions of Delphi and C++Builder.

The Orpheus packages have the following form:

ONnnXxVv.BPL (or .DPL in Delphi 3.)

Nnn is the version number of Orpheus. Xx are product-specific characters. Vv is the version of VCL supported. So, O400_D50.BPL is an Orpheus version 4.00, design time package for Delphi 5, O400_D30.DPL is the same package except for Delphi 3, etc.

Orpheus packages are automatically installed into Delphi and C++Builder if Orpheus is installed using the setup program. If you need to install the packages manually, you can install them using the “Install Packages” menu option.

The two run-time packages must be located somewhere on your system PATH. Starting with Delphi 4, the default directory for package files is Projects\BPL located off of the compiler’s root directory. For Delphi 4 – 6 and C++Builder 4 and 5, the setup program will install the packages to this directory by default. For Delphi 3 and C++Builder 3, the setup program will install the package files to the Windows system directory (Windows\System for Windows 95, Windows 98, and Windows ME or Winnt\System32 for Windows NT and Windows 2000).

If you want to locate the package files in a different location, you must make sure that the location appears in your system’s search path.

Be sure to alter the library path so that it includes the path to the Orpheus source files (Tools|Environment Options – Library Page) or add the Orpheus path to your system Path environment. This allows the compiler to find the Orpheus source files when required.

For C++Builder you must also make sure that the proper hpp* path appears in the library search path.

The component palette will be updated with several new Orpheus tabs. If you registered the data-aware components, tabs for ‘Orpheus (DB)’ are added.

Installing integrated help

The Orpheus help system is typically installed into Delphi and C++Builder by the SETUP program, but steps for manual installation are provided if the need arises.

Installing integrated help for Delphi 4 - 6 and C++Builder.

To install the help into Delphi or C++Builder, use the Help | Customize option from the IDE. Select the Contents page and browse to the desired Orpheus\Help directory and load the *.CNT file. In the Index and Link pages, do the same thing with the *.HLP files. When you are finished, select File | Save Project.

Installing integrated help for Delphi 3

To manually install the Orpheus help into Delphi 3, edit the Delphi3.cnt file (in the Delphi Help directory) and add the following line to the index section:

```
:Index Orpheus Reference =orph32.hlp
```

The first time you attempt to access Orpheus help, Delphi/Windows won't be able to locate the help file and will ask if you want to find the file yourself. Answer yes to this question and browse for the Orpheus help file (ORPH32.HLP), which should be in the Orpheus/Help/Delphi directory. This step will only be required the first time you access Orpheus' help.

Demonstration and example programs

Look for a list of example programs in EXAMPLES.HLP in the ORPHEUS\EXAMPLES directory. Orpheus examples are provided for both Delphi and C++Builder.

The demonstration programs are documented in "Chapter 34: Demo Programs" on page 1277. The example programs are provided so you can see how to use the various Orpheus components. See the source code for these programs for more information.

Organization of this Manual

This manual is organized as follows:

1. Chapter 1 is the introduction.
2. Chapter 2 describes the base classes and components that are used by all of the Orpheus components.
3. Chapters 3 through 34 describe the Orpheus components.
4. Chapter 35 describes the components that have been deprecated in version 4.
5. Chapter 36 discusses several demonstration programs that provide good examples of how to use most of the Orpheus components.
6. A separate identifier index and subject index are provided.

Each chapter starts with an overview of the classes and components discussed in that chapter. The overview also includes a hierarchy for those classes and components. Each class and component is then documented individually, in the following format:

Overview

Describes the class or unit.

Hierarchy

Shows the ancestors of the class being described, generally stopping at a VCL class. The hierarchy also lists the unit in which each class is declared and the number of the first page of the documentation of each ancestor. Some classes in the hierarchy are identified with a number in a bullet: ❶. This indicates that some of the properties, methods, or events listed for the class being described are inherited from this ancestor and documented in the ancestor class.

Properties

Lists all the properties in the class. Some properties may be identified with a number in a bullet: ❶. These properties are documented in the ancestor class from which they are inherited.

Methods

Lists all the methods in the class. Some methods may be identified with a number in a bullet: ❶. These methods are documented in the ancestor class from which they are inherited.

Events

Lists all the events in the unit. Some events may be identified with a number in a bullet: ❶. These events are documented in the ancestor class from which they are inherited.

Exceptions

A list of all the exceptions generated by the class or component. All exceptions are documented in "Orpheus Exception Classes" on page 68.

Reference section

Details the properties, methods, and events of the class or component. These descriptions are in alphabetical order. They have the following format:

- Declaration of the property, method, or event.
- Default value for properties, if appropriate.
- A short, one-sentence purpose. A ↳symbol is used to mark the purpose to make it easy to skim through these descriptions.
- Description of the property, method, or event. Parameters are also described here.
- Examples are provided in many cases.
- The "See also" section lists other properties, methods, or events that are pertinent to this item.

Throughout the manual, the 💣symbol is used to mark a warning or caution. Please pay special attention to these items.

On-line help

Although this manual provides a complete discussion of each component, keep in mind that there is an alternative source of information available. Help is available when you press <F1> with the caret positioned on the property, method, event, or component.

Naming conventions

To avoid class name conflicts with components and classes included with Delphi or from other third party suppliers, all Orpheus class names begin with "TOvc" or "TO32." The "Ovc" stands for Orpheus Visual Component and O32 is for the new 32-bit Orpheus components. Source files also start with "Ovc" or "O32" to make it easier to distinguish Orpheus source files from other files.

“Custom” in a component name means that the component is a base class for descendent components. Components with “Custom” as part of the class name do not publish any properties; instead descendants will publish those properties that are applicable to the derived component. If you create descendent components, use these custom classes and publish only the properties that you need instead of descending from the component class itself.

Design conventions

Within a class, Orpheus chooses between dynamic and virtual methods on the basis of performance requirements. Virtual methods are chosen for time-critical operations that might need to be overridden in descendants. Dynamic methods, which consume less code space, are used when performance is less important. Static methods are used when there is no reason ever to override the method.

All methods and variables in Orpheus are either public or protected (as opposed to private). If you look through the source code, you may come across sections of class declarations similar to this:

```
TOvcCustomEditor = class(TOvcEditBase)
protected {private}
    FAutoIndent      : Boolean;
    FBorderStyle     : TBorderStyle;
    FByteLimit       : LongInt;
    FFixedFont       : TOvcFixedFont;
    FHideSelection   : Boolean;
    FHighlightColors : TOvcColors;
    FInsertMode      : Boolean;
```

These should be treated as private sections. In other words, you should not routinely access these variables or methods from descendant classes. The section is defined as protected rather than private to give you access in cases that we may not have anticipated. If you access a method or variable in one of these sections, you should do so as a last resort and only after investigating and understanding the potential effects on other areas of the component’s code.

Each Orpheus class may include additional fields and methods that are protected and are documented only in the source code. It is unlikely that you will ever need to use or understand these, but the interested programmer should study the source code for additional ideas.

Technical Support

The best way to get an answer to your technical support question is to post it in the Orpheus newsgroup on our news server (news.turbopower.com). Many of our customers find the newsgroups a valuable resource where they can learn from other's experiences and share ideas in addition to getting answers to questions.

To get the most from the newsgroups, we recommend that you use dedicated newsreader software. More newsgroup information can be found at www.turbopower.com/tpslive.

Newsgroups are public so please do not post your product serial number, 16-character product unlocking code or any other private numbers (such as credit card numbers) in your messages.

Chapter 2: Building Blocks

This chapter provides information on the core classes and components used by nearly all Orpheus components. It describes the following components, classes, and routines:

- TOvcCustomControl and the TOvcCustomControlEx are classes that are used as the basis for most of the components provided in Orpheus.
- TOvcComponent is the base class for all non-visual components in Orpheus. It implements certain common data structures used internally by Orpheus and the About property which shows the current version.
- TO32Component is a replacement for TOvcComponent. It is a base class for new non-visual components in Orpheus. It is identical to TOvcComponent except that it uses standard Delphi streaming.
- TO32Control is the new version of the TOvcControl class. It provides all of the same functionality except it uses standard Delphi streaming.
- TOvcController component is available to all components derived from the TOvcCustomControlEx class. The TOvcController component provides a central location for error reporting and key-to-command translation.
- TOvcGraphicControl is the ancestor for all the graphic controls in Orpheus that do not require a window handle.
- The TOvcCommandProcessor allows you to use alternate keystrokes for nearly all data entry and component navigation tasks.
- TOvcCollection and TOvcCollectible define classes that provide collections and the ability to stream those collections and their contents.
- TO32Collection and TO32CollectionItem are replacements for the TOvcCollection and TOvcCollectible classes. They are direct descendants of the TCollection and TCollectionItem classes.
- Orpheus Date and Time Routines: The StDate unit is borrowed from our SysTools product to provide routines that allow you to store dates and times in compact formats, to convert them to other forms, and to perform date and time arithmetic.
- A hierarchical listing of all the exceptions defined by Orpheus is provided at the end of the chapter.

TOvcCustomControl Class

TOvcCustomControl serves as a common base class for many of the Orpheus components. Its primary purpose is to provide the ability to display an attached label. See “TOvcAttachedLabel Component” on page 762 for more information.

Hierarchy

TCustomControl (VCL)

    TOvcCustomControl (OvcBase)

Properties

About	AttachedLabel	LabelInfo

Events

AfterEnter	AfterExit	OnMouseWheel

Reference Section

About — read-only property

```
property About : string
```

↳ About shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

AfterEnter — event

```
property AfterEnter : TNotifyEvent
```

↳ AfterEnter defines an event handler that is called after the component receives the focus.

Unlike the OnEnter event, which is called during the focus change, AfterEnter is not called until after the focus change is complete.

TNotifyEvent is defined in the VCL's Classes unit.

AfterExit — event

```
property AfterExit : TNotifyEvent
```

↳ AfterExit defines an event handler that is called after the component receives the focus.

Unlike the OnExit event, which is called during the focus change, AfterExit is not called until after the focus change is complete

TNotifyEvent is defined in the VCL's Classes unit.

AttachedLabel — property

```
property AttachedLabel : TOvcAttachedLabel
```

↳ AttachedLabel provides access to the TOvcAttachedLabel component.

The TOvcAttachedLabel is normally accessed only at design time, but this property makes it possible to alter the attached label at run-time if necessary.

See "TOvcAttachedLabel Component" on page 762 for more information.

LabelInfo property

```
property LabelInfo : TOvcLabelInfo
```

↳ LabelInfo provides access to the status of the attached label.

TOvcLabelInfo (see page 764) groups the Visible, OffsetX, and OffsetY properties that determine whether the attached label is visible and where it is positioned.

See also: AttachedLabel

OnMouseWheel event

```
property OnMouseWheel : TMouseWheelEvent
```

```
TMouseWheelEvent = procedure(
  Sender : TObject; Shift : TShiftState;
  Delta, XPos, YPos : SmallInt) of object;
```

↳ OnMouseWheel defines an event handler that is fired when the mouse wheel is moved.

Sender is the component receiving the mouse input. Shift is the state of the keyboard shift keys. Delta is the change (+ or -) in the position of the mouse wheel. XPos and YPos represent the current mouse cursor position.

TOvcCustomControlEx Class

TOvcCustomControlEx serves as a common base class for many of the Orpheus components. Its primary purpose is to provide a Controller property for all its descendants.

During the creation of any descendant of the TOvcCustomControlEx class, the parent form is searched for the existence of a TOvcController component. If one is found, it is assigned as the controller for this component. If no TOvcController component is found, one is created and is then assigned as the controller for this component. See “TOvcController Component” on page 28 for additional information.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase). 16

TOvcCustomControlEx (OvcBase)

Properties

❶ About	❶ AttachedLabel
Controller	❶ LabelInfo

Events

❶ AfterEnter	❶ AfterExit	❶ OnMouseWheel

Controller property

```
property Controller : TOvcController
```

Default: The first TOvcController object on the form.

↳ Controller is the TOvcController object that is attached to this component.

If this property is not assigned, some or all the features provided by the component will not be available. The Controller property is not published, but descendants can publish it as needed (as most of the Orpheus components do). See “TOvcController Component” on page 28 for additional information.

TOvcComponent Class

TOvcComponent is the base class for all non-visual components in Orpheus. It implements certain common data structures used internally by Orpheus and the About property which shows the current version.

Hierarchy

TComponent (VCL)

    TOvcComponent (OvcBase)

Properties

About

Reference Section

About — read-only property

```
property About : string
```

About shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. The Orpheus about box can be displayed by double-clicking this property or selecting the dialog button to the right of the property value.

TO32Component Class

TO32Component is a replacement for the TOvcComponent class. It is a direct descendant of the TComponent class and is identical to TComponent except that it implements the Orpheus About property. It differs from the TOvcComponent class in that it uses standard VCL streaming instead of the old Orpheus specific streaming methods which were used to support Delphi 1 and 2.

Hierarchy

TComponent (VCL)

  TO32Component (OvcBase)

Properties

  About

Reference Section

About — read-only property

```
property About : string
```

About shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

TO32CustomControl Class

The TO32CustomControl is a replacement for the TOvcCustomControl class. It provides all of the same benefits as the TOvcCustomControl except that it uses standard VCL streaming.

Hierarchy

TCustomControl (VCL)

TO32CustomControl (OvcBase)

Properties

About	AttachedLabel	LabelInfo

Events

AfterEnter	AfterExit	OnMouseWheel

Reference Section

About — read-only property

```
property About : string
```

↳ About shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

AfterEnter — event

```
property AfterEnter : TNotifyEvent
```

↳ AfterEnter defines an event handler that is called after the component receives the focus.

Unlike the OnEnter event, which is called during the focus change, AfterEnter is not called until after the focus change is complete.

TNotifyEvent is defined in the VCL's Classes unit.

AfterExit — event

```
property AfterExit : TNotifyEvent
```

↳ AfterExit defines an event handler that is called after the component receives the focus.

Unlike the OnExit event, which is called during the focus change, AfterExit is not called until after the focus change is complete.

TNotifyEvent is defined in the VCL's Classes unit.

AttachedLabel — property

```
property AttachedLabel : TOvcAttachedLabel
```

↳ AttachedLabel provides access to the TOvcAttachedLabel component.

The TOvcAttachedLabel is normally accessed only at design time, but this property makes it possible to alter the attached label at run-time if necessary.

See "TOvcAttachedLabel Component" on page 762 for more information.

LabelInfo property

```
property LabelInfo : TOvcLabelInfo;
```

LabelInfo provides access to the status of the attached label.

TOvcLabelInfo (see page 764) groups the Visible, OffsetX, and OffsetY properties that determine whether the attached label is visible and where it is positioned.

See also: AttachedLabel

OnMouseWheel event

```
property OnMouseWheel : TMouseWheelEvent
```

```
TMouseWheelEvent = procedure(
  Sender : TObject; Shift : TShiftState;
  Delta, XPos, YPos : Word) of object;
```

OnMouseWheel defines an event handler that is fired when the mouse wheel is moved.

Sender is the component receiving the mouse input. Shift is the state of the keyboard shift keys. Delta is the change (+ or -) in the position of the mouse wheel. XPos and YPos represent the current mouse cursor position.

TOvcController Component

The TOvcController serves a crucial role for many of the Orpheus components by providing a central location for error reporting and key-to-command translation.

Many Orpheus components that require user input via the keyboard employ the key-to-command translation facilities provided by the EntryCommands property. The configurable key-to-command translation tables allow you to produce applications that can load and use different command tables instead of the ones compiled into the application.

In some situations it might be necessary to attach components from several forms to a common controller on a different form or data module. This can be done by selecting the File|Use Unit menu item in Delphi. In C++Builder, select the File| Include Unit Hdr menu item. The MDIEDIT demo program demonstrates how to assign the Controller at run time.

See "TOvcCommandProcessor Class" on page 50 for additional information.

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

TOvcController (OvcBase)

Properties

❶ About	Handle	OnPostEdit
EntryCommands	InsertMode	OnPreEdit
EntryOptions	OnDelayNotify	OnTimerTrigger
Epoch	OnError	TimerPool
ErrorPending	OnGetEpoch	
ErrorText	OnIsSpecialControl	

Methods

DelayNotify	MarkAsUninitialized	ValidateEntryFieldsEx
IsSpecialButton	ValidateEntryFields	ValidateTheseEntryFields

Reference Section

DelayNotify method

```
procedure DelayNotify (Sender : TObject; NotifyCode : Word);
```

↳ DelayNotify starts the chain of events that will fire the OnDelayNotify event.

Sender is intended to be the object where the call takes place, but can be any data that you care to have passed to the OnDelayNotify event. NotifyCode is intended to provide a value to the OnDelayNotify event that will indicate the purpose of the notification. After calling DelayNotify, a message is posted using PostMessage (not sent using SendMessage or Perform) to the controller. (Posting allows other messages in the message queue to be processed.)

EntryCommands property

```
property EntryCommands : TOvcCommandProcessor
```

Default: Default Command Table

↳ EntryCommands provides keystroke to command translation.

The EntryCommands property is published for use by other Orpheus components. It provides access to a common command processor that can be used by several components at the same time. The property editor associated with this property allows you to manipulate all aspects of the command tables. See “TOvcCommandProcessor Class” on page 50 for additional information.

```
property EntryOptions : TOvcEntryFieldOptions

TOvcBaseEFOptions = set of TOvcBaseEFOption;

TOvcBaseEFOption = (
  efoAutoAdvanceChar, efoAutoAdvanceLeftRight,
  efoAutoAdvanceUpDown, efoAutoSelect,
  efoBeepOnError, efoInsertPushes);
```

EntryOptions determines options for all entry fields that use this controller.

Each of these options is described in the following table:

Option	**Description**
efoAutoAdvanceChar	This option determines what happens if the caret is at the end of the entry field and you enter another character. If set, the focus is advanced to the next component in the tab order (the equivalent of pressing <Tab>). Otherwise, the caret remains where it is.
efoAutoAdvanceLeftRight	This option determines what happens if the caret is at the beginning or end of the entry field and a cursor movement command is issued. The first case to consider is when the caret is at the beginning of the field and a command that would normally move the caret to the left is issued. If AutoAdvanceLeftRight is True, the focus is moved to the previous component in the tab order (the equivalent of pressing <Shift><Tab>). If AutoAdvanceLeftRight is False, the caret remains where it is. The second case to consider is when the caret is at the end of a field and a command that would normally move the caret to the right is issued. If AutoAdvanceLeftRight is True, the focus is advanced to the next component in the tab order (the equivalent of pressing <Tab>). If AutoAdvanceLeftRight is False, the caret remains where it is.

Option	Description
efoAutoAdvanceUpDown	This option determines whether the cursor keys change the focus. If AutoAdvanceUpDown is True, the <Up> key is treated the same as <ShiftTab>, which moves the focus to the previous component on the form, and the <Down> key is treated the same as <Tab>, which moves the focus to the next component on the form. When you set ArrowIncDec to True, AutoAdvanceUpDown is automatically set to False because the up and down arrow keys cannot be used for both purposes at once. Conversely, when you set AutoAdvanceUpDown to True, ArrowIncDec is set to False.
efoAutoSelect	This option determines whether the field is selected when it receives the focus. If AutoSelect is True, the contents of the field are selected when the field receives the focus. This does not apply if the field is selected using the mouse.
efoBeepOnError	This option determines whether a beep is sounded when an invalid character is entered. If BeepOnError is True, MessageBeep is called when an invalid character is entered (for example, if <A> is entered in a numeric field) or when a character is entered and the field is full (and the InsertPushes property is set to False).
efoInsertPushes	This option determines what happens when characters are entered into a completely filled entry field. If InsertPushes is True, as it is by default, and a character is entered into a full entry field (or subfield), a character at the right-most position in the field (or subfield) is “pushed” off the end. If InsertPushes is False, characters entered into a full entry field (or subfield) are ignored.

Epoch property

```
property Epoch : Integer
```

Default: Current century

Epoch specifies the starting year of a 100 year period.

ErrorPending run-time property

```
property ErrorPending : Boolean
```

Default: False

ErrorPending indicates whether an entry field has generated an error.

This property is set to True when an Orpheus entry field fails validation or False if the field validation detects no errors. Other components, such as the notebook and table, use this property to detect errors in components that they own.

ErrorText run-time property

```
property ErrorText : string
```

ErrorText allows the user to set an error message from a UserValidation routine.

The ErrorText property provides a location that user error handlers can store error text for use by an OnError event handler. The string would be assigned in the user validation routine:

```
  Controller.ErrorText := 'error message';
```

This string is later passed to the OnError event handler.

This property will always contain text from the last error unless explicitly cleared.

See also: OnError

Handle — run-time, read-only property

```
property Handle : hWnd
```

Handle is the window handle of this component.

InsertMode — run-time property

```
property InsertMode : Boolean
```

Default: True

InsertMode indicates insert or overwrite mode.

This property is checked by the entry field and the editor components each time they receive the focus to determine the insert/overwrite state. It also allows the insert or overwrite state of all components using this controller to be changed when any one component changes its insert or overwrite state.

IsSpecialButton — virtual method

```
function IsSpecialButton(H : hWnd) : Boolean;
```

IsSpecialButton determines whether the specified window handle is one of the button components for a special button.

IsSpecialButton returns True if the OnIsSpecialControl event handler returns True. Otherwise, it returns False.

Note: This method is used by the Orpheus entry fields to determine if validation should be performed.

MarkAsUninitialized — method

```
procedure MarkAsUninitialized(Uninitialized : Boolean);
```

MarkAsUninitialized flags all entry fields as uninitialized.

This method cycles through all entry field components on the form, setting their Uninitialized property to the value contained in the Uninitialized parameter.

Note: This method sets the Uninitialized property for all fields on the form at the same time. If you don't need to set the Uninitialized property for all fields at once, you can directly set the Uninitialized property for the necessary fields.

See also: TOvcBaseEntryField.Uninitialized

OnDelayNotify event

```
property OnDelayNotify : TDelayNotifyEvent

TDelayNotifyEvent = procedure(
  Sender : TObject; NotifyCode : Word) of object;
```

↳ OnDelayNotify provides an event that is fired by calling the DelayNotify method.

OnError event

```
property OnError : TDataErrorEvent

TDataErrorEvent = procedure(
  Sender : TObject; ErrorCode : Word;
  const ErrorMessage : string) of object;
```

↳ OnError defines an event handler that is called when an error occurs.

The method assigned to the OnError event is called by Orpheus entry fields when field validation (or user validation) detects an error. Sender is the component that generated the error. ErrorCode is one of the error codes defined in the OvcData unit or a user-defined error value. ErrorMessage is a string that describes the error.

The following example displays an error reporting dialog that informs the user of an error generated by an entry field:

```
  procedure
  TForm1.OvcController1Error(Sender : TObject;
  ErrorCode : Word;
  ErrorMessage : string);
  var
  ErrorDlg : TErrorDlg;
  begin
  ErrorDlg := TErrorDlg.Create(Self);
  {set appropriate ErrorDlg fields to describe the error}
  try
  ErrorDlg.ShowModal;
  finally
  ErrorDlg.Free;
  end;
  end;
  ...
  OvcController1.OnError := OvcController1Error;
```

See also: ErrorText, TOvcBaseEntryField.Controller

OnGetEpoch event

```
property OnGetEpoch : TGetEpochEvent
```

```
TGetEpochEvent = procedure (
  Sender : TObject; var Epoch : Integer) of object;
```

OnGetEpoch defines an event handler that allows you to provide the Epoch value used by other Orpheus controls.

The Epoch value and its use is discussed in the “The Epoch property” on page 321.

OnIsSpecialControl event

```
property OnIsSpecialControl : TIsSpecialControlEvent
```

```
TIsSpecialControlEvent = procedure(
  Sender : TObject; Control : TWinControl;
  var Special : Boolean) of object;
```

OnIsSpecialControl defines an event handler that allows you to specify that the passed Control should receive special treatment.

Setting the Special property to True informs the Sender not to perform validation on the field that is about to lose the focus.

Control is the control that is about to receive the focus. If you do not wish validation performed when the focus is moving to this control, set Special to True.

OnPostEdit event

```
property OnPostEdit : TPostEditEvent
```

```
TPostEditEvent = procedure(
  Sender : TObject; GainingControl : TWinControl) of object;
```

OnPostEdit defines an event handler that is called when the focus changes.

The method assigned to the OnPostEdit event is called to provide notification when an entry field loses the focus. Sender is the component that lost the focus. GainingControl is the component that received the focus, or nil if there is no gaining control.

Unlike the OnExit event, which is called during the focus change, the OnPostEdit event is called after the focus change is complete.

Use the OnPostEdit event when you perform the same post-focus processing actions for multiple entry fields. If you need to perform post-focus processing for a single entry field component, use its AfterExit event.

The following example displays the name of the entry field that is losing the focus:

```
procedure
TForm1.OvcController1PostEdit(Sender : TObject;
GainingControl : TWinControl);
begin
Panel2.Caption :=
  'PostEdit for: ' + (Sender as TOvcBaseEntryField).Name;
end;
```

See also: AfterExit, OnPreEdit

OnPreEdit event

```
property OnPreEdit : TPreEditEvent
```

```
TPreEditEvent = procedure(
  Sender : TObject; LosingControl : TWinControl) of object;
```

↳ OnPreEdit defines an event handler that is called when the focus changes.

The method assigned to the OnPreEdit event is called to provide notification when an entry field component receives the focus. Sender is the component that received the focus. LosingControl is the component that lost the focus, or nil if there is no losing control.

Unlike the OnEnter event, which is called during the focus change, the OnPreEdit event is called after the control receives the focus.

Use the OnPreEdit event when you perform the same pre-focus processing actions for multiple entry fields. If you need to perform pre-focus processing for a single entry field component, use its AfterEnter event.

The following example displays the name of the entry field that is gaining the focus:

```
procedure
TForm1.OvcController1PreEdit(Sender : TObject;
  LosingControl : TWinControl);
begin
  Panel1.Caption := 'PreEdit for: ' +
  (Sender as TOvcBaseEntryField).Name;
end;
```

See also: AfterEnter, OnPostEdit

OnTimerTrigger event

```
property OnTimerTrigger : TTriggerEvent
```

OnTimerTrigger defines an event handler that is called when a timer pool trigger occurs.

The method assigned to the OnPreEdit event is attached (internally) to the timer pool's OnAllTriggers event and provides notification of all timer pool triggers.

See "TOvcTimerPool Component" on page 864 for information on how to use the timer pool class.

See also: TimerPool

TimerPool run-time property

```
property TimerPool : TOvcTimerPool
```

TimerPool allows access to the contained TOvcTimerPool object.

This property allows other Orpheus components to access a common timer pool instance.

See "TOvcTimerPool Component" on page 864 for information on how to use the timer pool class.

See also: OnTimerTrigger

ValidateEntryFields method

```
function ValidateEntryFields : TComponent;
```

ValidateEntryFields asks each entry field to validate its contents.

The ValidateEntryFields method cycles through all the components on the form looking for any Orpheus entry fields. When one is found, ValidateEntryFields asks the entry field to validate itself. If the result of the validation is an error, ValidateEntryFields stops cycling through the components and returns the entry field component that reported the error. If there are no entry fields on the form or if all entry fields are valid, ValidateEntryFields returns nil.

If an error is detected, it is reported via default error handling or by calling the OnError event handler. The focus is moved to the field that generated the error.

This method is typically used to verify that all entry fields on the form contain valid data prior to actually closing the form and processing its data. Your code should exit and allow the user to correct the error if ValidateEntryFields returns a non-nil value.

See also: TOvcBaseEntryField.ValidateSelf, ValidateEntryFieldsEx

ValidateEntryFieldsEx method

```
function ValidateEntryFieldsEx(
  ReportError, ChangeFocus : Boolean): TComponent;
```

ValidateEntryFieldsEx asks each entry field to validate its contents and provides control over error reporting and focus.

ValidateEntryFieldsEx works just like ValidateEntryFields, but it also allows you to control whether an error is reported (set ReportError to True) and whether the focus is forced back to the field containing the error (set ChangeFocus to True).

See also: ValidateEntryFields

ValidateTheseEntryFields method

```
function ValidateTheseEntryFields(
  const Fields : array of TComponent) : TComponent;
```

ValidateTheseEntryFields asks the specified entry fields to validate their contents.

Returns nil if no error. Otherwise, it returns a pointer to the field with error.

TOvcCollection Class

TOvcCollection implements Orpheus collections. A collection is a series of component instances of the same type that are automatically streamed as child components by the component, which owns the collection.

Hierarchy

TPersistent (VCL)

  TOvcCollection (OvcBase)

Properties

Count	Item
ItemClass	Owner

Events

OnChanged

Reference Section

Count — property

```
property Count : Integer
```

Count returns the number of items in the collection.

Item — property

```
property Item[Index : Integer] : TOvcCollectible
```

Item returns a particular item in the collection.

ItemClass — property

```
property ItemClass : TOvcCollectibleClass
```

ItemClass returns the type of the items in the collection.

OnChanged — event

```
property OnChanged : TNotifyEvent
```

OnChanged is called whenever a published property for an item in the collection changes.

Owner — property

```
property Owner : TComponent
```

Owner returns a reference to the component which owns the collection.

TOvcCollectible Class

TOvcCollectible is the base class for components that are items in Orpheus collections. Examples of this include field definitions in ReportViews, and sections in the button header.

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

TOvcCollectible (OvcBase)

Properties

❶	About	DisplayText	Name
	Collection	Index	

Reference Section

Collection property

```
property Collection : TOvcCollection
```

↳ Collection refers to the collection, which is the owner of the current item.

DisplayText run-time, read-only property

```
property DisplayText : string
```

↳ DisplayText is used by the standard collection editor to retrieve an item display string.

Index property

```
property Index : Integer
```

↳ Index determines the position of the collectible in the owner collection.

Index is not published but is stored implicitly by the collections as the sequence in which collectibles are streamed.

Name property

```
property Name : String
```

↳ Name determines the name of the collectible.

The name can be blank, but if it is present, it must be a valid identifier name and it must be unique.

TOvcCollectibleControl Class

TOvcCollectibleControl is the base class for visual components that are items in Orpheus collections.

Hierarchy

TComponent (VCL)

- ❶ TOvcComponent (OvcBase)
  - TOvcCollectibleControl (OvcBase)

Properties

❶	About	DisplayText	Name
	Collection	Index	

Reference Section

Collection property

```
property Collection : TOvcCollection
```

Collection refers to the collection which is the owner of the current item.

DisplayText run-time, read-only property

```
property DisplayText : string
```

DisplayText is used by the standard collection editor to retrieve an item display string.

Index property

```
property Index : Integer
```

Index determines the position of the collectible in the owner collection.

Index is not published but is stored implicitly by the collections as the sequence in which collectibles are streamed.

Name property

```
property Name : String
```

Name determines the name of the collectible.

The name can be blank, but if it is present, it must be a valid identifier name and it must be unique.

TO32Collection Class

TO32Collection is a replacement for the TOvcCollection and TOvcCollectible classes. TO32Collection is a direct descendant of the TCollection and TCollectionItem classes and as such implement standard VCL streaming. It serves as a replacement for the TOvcCollection class.

Hierarchy

TCollection (VCL)

TO32Collection (OvcBase)

Properties

Count	ItemClass	ReadOnly
Item	Owner	

Methods

Add	ItemByName	ParentForm

Events

OnChanged

Reference Section

Add — method

```
function Add : TO32CollectionItem;
```

Add creates a new item in the collection and returns a reference to it.

Count — property

```
property Count : Integer
```

Count returns the number of items in the collection.

Item — property

```
property Item[Index : Integer] : TO32CollectionItem
```

Item returns a particular item in the collection.

ItemByName — method

```
function ItemByName(Name: String); TO32CollectionItem;
```

Returns a reference to the item specified by Name. If Name is an empty string then ItemByName returns nil.

ItemClass — property

```
property ItemClass : TCollectionItemClass
```

ItemClass returns the type of the items in the collection.

OnChanged — event

```
property OnChanged : TNotifyEvent
```

OnChanged is called whenever a published property for an item in the collection changes.

Owner — property

```
property Owner : TPersistent
```

Owner returns a reference to the component, which owns the collection.

ParentForm method

```
function ParentForm: TForm;
```

Returns a reference to the parent form of the control which owns the collection.

ReadOnly property

```
property ReadOnly : Boolean
```

ReadOnly determines whether or not changes to the collection are allowed.

TO32CollectionItem Class

TO32CollectionItem is a replacement for the TOvcCollection and TOvcCollectible classes. It is a direct descendants of the TCollection and TCollectionItem classes and as such implement standard VCL streaming. TO32CollectionItem is the base class for components that are items in Orpheus 4.0 collections. Examples of this include folder and items in the lookout bar component.

Hierarchy

TCollectionItem (VCL)

    TO32CollectionItem (OvcBase)

Properties

About	DisplayText	Name

Reference Section

About — read-only property

```
property About : string
```

About shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. The Orpheus about box can be displayed by double-clicking this property or selecting the dialog button to the right of the property value.

DisplayText — property

```
property DisplayText : string
```

Returns the text that will be displayed in relation to the item.

A common use of DisplayText is in the TO32LookoutBar Folders and Items. DisplayText is the value that is displayed on the folder buttons and adjacent to each item.

Name — property

```
property Name : string
```

Returns the name of the item.

TOvcCommandProcessor Class

The command processor allows your applications to provide a unique ability that is not found in other Windows programs. You can define and use alternate keystrokes for nearly all data entry and component navigation tasks. And with only a small programming effort, you can give your users the capability too. The TOvcCommandProcessor class provides key-to-command mapping.

TOvcCommandProcessor is used by nearly all the Orpheus components that do not descend from one of the VCL components. Three built-in command tables provide key-to-command mapping for the Orpheus components. The Default Command Table is used by all the Orpheus components except TOvcTable. The WordStar Command Table provides command-key mappings for the common WordStar key sequences. The Grid Command Table is used by the TOvcTable component. These command tables can be modified (at design time or at run time), deleted, replaced, stored, and read from disk. You can add your own command tables and design your applications so that your users can configure the command tables.

TOvcCommandProcessor translates Windows messages to commands for the Orpheus components. Message structures are passed to the command processor's Translate method which then searches its active command tables for an entry that matches the key sequences stored in the message structure. If one is found, the corresponding command is returned to the calling component.

An instance of the TOvcCommandProcessor is created and owned by each TOvcController component. Other Orpheus components have access to the command processor through their Controller property.

Commands

Each command table consists of one or more key-to-command mappings, where keys are assigned to the commands listed in the following table. The commands are Word constants that are defined in the OvcConst unit.

Some commands apply only to specific components, as shown in Table 2.4 (e.g., the ccBotRightCell command is processed only by the table component). The components that use each command are listed by code in the “Used By” column. A description of the codes follows the table.

Table 2.1: *Component specific commands*

Command	Value	Used By	Description
ccNone	0		No command or an unknown key sequence.
ccBack	1	ae, ed, ef	Delete the character to the left of the caret.
ccBotOfPage	2	ed, vw	Move the caret to the bottom of the window.
ccBotRightCell	3	tbl	Move the focus to the cell at the bottom-right of the table.
ccCompleteDate	4	pfae, pf	Insert the current day, month, or year (depending on the position of the caret) in a date picture field or array editor. Or, if the field is selected, replace the selection with the current date.
ccCompleteTime	5	pfae, pf	Insert the current hour, minute, or second (depending on the position of the caret) in a time picture field or array editor. Or, if the field is selected, replace the selection with the current time.
ccCopy	6	ae, ed, ef, vw, tbl	Copy the selection to the clipboard.
ccCtrlChar	7	pf, pfae, sf, sfae	Treat the next character as a control character.
ccCut	8	ae, ed, ef, tbl	Copy the selection to the clipboard and then delete it.
ccDec	9	ae, ef	Decrement the current value by one ordinal amount.

Table 2.1: *Component specific commands (continued)*

Command	Value	Used By	Description
ccDel	10	ae, ed, ef	Delete the current character. Or, if text is selected, delete the selection.
ccDelBol	11	pf, pfae, sf, sfae	Delete text from the caret position to the beginning of the line.
ccDelEol	12	ed, pf, pfae, sf, sfae	Delete text from the caret position to the end of the line.
ccDelLine	13	ae, ed, ef	Delete the current line.
ccDelWord	14	ae, ed, ef	Delete the word to the right of the caret.
ccDown	15	ae, cal, ed, ef, nbk, tbl, vlb, vw	Move the caret or the focus down to the next line (ed, vw), the next component (ef), the next list item (ae, vlb), the next week (cal), the next tab (nbk), or the next row (tbl).
ccEnd	16	cal, ed, nbk, pf, pfae, sf, sfae, tbl, vlb, vw	Move the caret or the focus to the end of the current line (ed, vw), end of the field (pf, pfae, sf, sfae), last day of the month (cal), last item in the list (vlb), last tab (nbk), last column (tbl).
ccExtBotOfPage	25	vw, tbl	Extend the selection to the bottom of the page.
ccExtendDown	17	ed, tbl, vlb, vw	Extend the selection down to the next line (ed, vw), the next list element (vlb), or the next row (tbl).
ccExtendEnd	18	ae, ed, ef, tbl, vlb, vw	Extend the selection to the end of the field (ae, ef), the end of the current line (ed, vw), the last list item (vlb), or the last column (tbl).

Table 2.1: *Component specific commands (continued)*

Command	Value	Used By	Description
ccExtendHome	19	ae, ed, ef, tbl, vlb, vw	Extend the selection to the start of the field (ae, ef), the start of the current line (ed, vw), the first list item (vlb), or the first column (tbl).
ccExtendLeft	20	ae, ed, ef, tbl, vw	Extend the selection left one character (ae, ed, ef, vw) or left one column (tbl).
ccExtendPgDn	21	ed, tbl, vlb, vw	Extend the selection down one page.
ccExtendPgUp	22	ed, tbl, vlb, vw	Extend the selection up one page
ccExtendRight	23	ae, ed, ef, tbl, vw	Extend the selection right one character (ae, ed, ef, vw) or right one column (tbl.
ccExtendUp	24	ed, tbl, vlb, vw	Extend the selection up one line (ed, vw), one list item (vlb), or one row (tbl).
ccExtFirstPage	26	tbl, vw	Extend the selection to the first row (the column remains unchanged)
ccExtLastPage	27	tbl, vw	Extend the selection to the last row (the column remains unchanged)
ccExtTopOfPage	28	vw, tbl	Extend the selection to the top of the page.
ccExtWordLeft	29	ae, ed, ef, tbl, vw	Extend the selection left one word (ae, ed, ef, vw) or left one page (tbl).
ccExtWordRight	30	ae, ed, ef, tbl, vw	Extend the selection right one word (ae, ed, ef, vw) or right one page (tbl).
ccFirstPage	31	ae, cal, ed, tbl, vw	Move the focus or caret to the first list item (ae), the first day of the month (cal), the first character of text (ed, vw), or the first cell in the table (tbl).

Table 2.1: *Component specific commands (continued)*

Command	Value	Used By	Description
ccGotoMarker0-9	32-41	ed, vw	Move to the previously set marker (0-9).
ccHome	42	cal, ed, nbk, pf, pfae, sf, sfae, tbl, vlb, vw	Move the caret or focus to the start of the field (pf, pfae, sf, sfae), the start of the line (ed, vw), the first day of the month (cal), the top list item (vlb), first tab (nbk) or the first column (tbl).
ccInc	43	ae, ef	Increment the current value by one ordinal amount.
ccIns	44	ae, ed, ef	Toggle insert mode.
ccLastPage	45	ae, cal, ed, tbl, vw	Move the focus or caret to the last list item (ae), the last day of the month (cal), the last character of text (ed, vw), or the last cell in the table (tbl).
ccLeft	46	ae, cal, ed, ef, nbk, tbl, vlb, vw	Move the caret left one position (ae, ed, ef, vlb, vw), left to the previous day (cal), left to the previous tab (nbk), or left to the previous cell (tbl).
ccNewLine	47	ed	Create a new line.
ccNextPage	48	ae, cal, ed, tbl, vlb, vw	Move the focus or the caret to the page following the current page (ae, ed, tbl, vlb, vw), or the next month (cal).
ccPageLeft	49	tbl	Move left a page (one window width).
ccPageRight	50	tbl	Move right a page (one window width).
ccPaste	51	ae, ed, ef	Paste text from the clipboard.

Table 2.1: *Component specific commands (continued)*

Command	Value	Used By	Description
ccPrevPage	52	ae, cal, ed, tbl, vlb, vw	Move the focus or the caret to the previous page (ae, ed, tbl, vlb, vw) or the previous month (cal).
ccRedo	53	ed	Redo the last undone operation.
ccRestore	54	ae, ef	Restore the default field value.
ccRight	55	ae, cal, ed, ef, nbk, tbl, vlb, vw	Move the caret right one position (ae, ed, ef, vlb, vw), right to the next day (cal), right to the next tab (nbk), or right to the next cell (tbl).
ccScrollDown	56	ed, vw	Scroll down one line.
ccScrollUp	57	ed, vw	Scroll up one line.
ccSetMarker0-9	58-67	ed, vw	Set the position of marker (0-9) to the current caret position.
ccTab	68	ed	Insert a tab into the text stream.
ccTableEdit	69	tbl	enter / exit table edit mode.
ccTopLeftCell	70	tbl	Move to the top left cell in a table.
ccTopOfPage	71	ed, vw, tbl	Move the caret to the top of the window.
ccUndo	72	ed	undo the last operation.
ccUp	73	ae, cal, ed, ef, nbk, tbl, vlb, vw	Move the caret or the focus up to the previous line (ed, vw), the previous component (ef), the previous list item (ae, vlb), the previous week (cal), the previous tab (nbk), or the previous row (tbl).

Table 2.1: *Component specific commands (continued)*

Command	Value	Used By	Description
ccWordLeft	74	ae, ed, ef	Move the caret left one word.
ccWordRight	75	ae, ed, ef	Move the caret right one word.
ccUser0- ccUserXxx	256..65535	ae, cal, ed, ef, tbl, vlb, vw	Call the user-defined method assigned to the OnUserCommand event (ccUser0 through ccUser9 are defined in OvcConst).

```
ae   = TOvcSimpleArrayEditor, TOvcPictureArrayEditor, and
       TOvcNumericArrayEditor
cal  = TOvcCalendar
ed   = TOvcEditor and TOvcTextFileEditor
ef   = TOvcSimpleField, TOvcPictureField, and TOvcNumericField
nbk  = TOvcNotebook
nf   = TOvcNumericField
nfae = TOvcNumericArrayEditor
pf   = TOvcPictureField
pfae = TOvcPictureArrayEditor
sf   = TOvcSimpleField
sfae = TOvcSimpleArrayEditor
tbl  = TOvcTable
vlb  = TOvcVirtualListbox
vw   = TOvcViewer and TOvcTextFileViewer
```

Command tables

When a command processor is first created (this normally happens when a TOvcController component is placed on the form), three command tables are automatically added to the internal list of command tables.

The Default Command Table contains command-key mappings that apply to nearly all the Orpheus components. These are the CUA-style commands. The WordStar Command Table provides command-key mappings for the common WordStar key sequences. The Grid Command Table provides command-key mappings that are used by the “TOvcTable Component” on page 1023.

Since the command tables are completely configurable, you can add to the recognized key sequences or even create a completely new command table. The mechanism that performs keystroke to command mapping is described in “TOvcCommandProcessor Class” on page 50.

Each command table can be either active or inactive. The Default Command Table, shown in Table 2.2, is the only command table that is active by default.

Table 2.2: *Default Command Table*

Key Sequence	**Command**
`<Backspace>`	`ccBack`
`<ShiftBackspace>`	`ccBack`
`<CtrlPgDn>`	`ccBotOfPage`
`<CtrlIns>`	`ccCopy`
`<CtrlC>`	`ccCopy`
`<ShiftDel>`	`ccCut`
`<CtrlX>`	`ccCut`
`<Del>`	`ccDel`
`<Down>`	`ccDown`
`<End>`	`ccEnd`
`<CtrlShiftPgDn>`	`ccExtBotOfPage`
`<ShiftDown>`	`ccExtendDown`
`<ShiftEnd>`	`ccExtendEnd`
`<ShiftHome>`	`ccExtendHome`
`<ShiftLeft>`	`ccExtendLeft`
`<ShiftPgDn>`	`ccExtendPgDn`
`<ShiftPgUp>`	`ccExtendPgUp`
`<ShiftRight>`	`ccExtendRight`
`<ShiftUp>`	`ccExtendUp`
`<CtrlShiftHome>`	`ccExtFirstPage`
`<CtrlShiftEnd>`	`ccExtLastPage`
`<CtrlShiftPgUp>`	`ccExtTopOfPage`
`<CtrlShiftLeft>`	`ccExtWordLeft`
`<CtrlShiftRight>`	`ccExtWordRight`
`<CtrlHome>`	`ccFirstPage`
`<Ctrl0>`	`ccGotoMarker0`
`<Ctrl9>`	`ccGotoMarker9`

Table 2.2: *Default Command Table (continued)*

Key Sequence	**Command**
`<Home>`	`ccHome`
`<Ins>`	`ccIns`
`<CtrlEnd>`	`ccLastPage`
`<Left>`	`ccLeft`
`<CtrlEnter>`	`ccNewLine`
`<Tab>`	`ccNextCtrl`
`<CtrlTab>`	`ccNextCtrl`
`<PgDn>`	`ccNextPage`
`<ShiftIns>`	`ccPaste`
`<CtrlV>`	`ccPaste`
`<ShiftTab>`	`ccPrevCtrl`
`<CtrlShiftTab>`	`ccPrevCtrl`
`<PgUp>`	`ccPrevPage`
`<CtrlShiftZ>`	`ccRedo`
`<AltBackspace>`	`ccRestore`
`<CtrlZ>`	`ccRestore`
`<Right>`	`ccRight`
`<CtrlShift0>`	`ccSetMarker0`
`<CtrlShift9>`	`ccSetMarker9`
`<CtrlPgUp>`	`ccTopOfPage`
`<CtrlZ>`	`ccUndo`
`<Up>`	`ccUp`
`<CtrlLeft>`	`ccWordLeft`
`<CtrlRight>`	`ccWordRight`

The WordStar Command Table is shown the Table 2.3.

Table 2.3: *WordStar Command Table*

Key Sequence	Command
`<CtrlH>`	`ccBack`
`<CtrlP>`	`ccCtrlChar`
`<CtrlG>`	`ccDel`
`<CtrlQ><Y>`	`ccDelEol`
`<CtrlY>`	`ccDelLine`
`<CtrlT>`	`ccDelWord`
`<CtrlX>`	`ccDown`
`<CtrlQ><D>`	`ccEnd`
`<CtrlQ><0>`	`ccGotoMarker0`
`<CtrlQ><9>`	`ccGotoMarker9`
`<CtrlQ><S>`	`ccHome`
`<CtrlV>`	`ccIns`
`<CtrlS>`	`ccLeft`
`<CtrlC>`	`ccNextPage`
`<CtrlR>`	`ccPrevPage`
`<CtrlQ><L>`	`ccRestore`
`<CtrlD>`	`ccRight`
`<CtrlZ>`	`ccScrollDown`
`<CtrlW>`	`ccScrollUp`
`<CtrlK><0>`	`ccSetMarker0`
`<CtrlK><9>`	`ccSetMarker9`
`<CtrlE>`	`ccUp`
`<CtrlA>`	`ccWordLeft`
`<CtrlF>`	`ccWordRight`

The Grid Command Table is shown in Table 2.4.

Table 2.4: *Grid Command Table*

Key Sequence	**Command**
`<CtrlEnd>`	`ccBotOfPage`
`<CtrlDown>`	`ccBotRightCell`
`<Down>`	`ccDown`
`<End>`	`ccEnd`
`<CtrlShiftPgDn>`	`ccExtBotOfPage`
`<ShiftDown>`	`ccExtendDown`
`<ShiftEnd>`	`ccExtendEnd`
`<ShiftHome>`	`ccExtendHome`
`<ShiftLeft>`	`ccExtendLeft`
`<ShiftPgDn>`	`ccExtendPgDn`
`<ShiftPgUp>`	`ccExtendPgUp`
`<ShiftRight>`	`ccExtendRight`
`<ShiftUp>`	`ccExtendUp`
`<CtrlShiftHome>`	`ccExtFirstPage`
`<CtrlShiftEnd>`	`ccExtLastPage`
`<CtrlShiftPgUp>`	`ccExtTopOfPage`
`<CtrlShiftLeft>`	`ccExtWordLeft`
`<CtrlShiftRight>`	`ccExtWordRight`
`<CtrlPgUp>`	`ccFirstPage`
`<Home>`	`ccHome`
`<CtrlPgDn>`	`ccLastPage`
`<Left>`	`ccLeft`
`<PgDn>`	`ccNextPage`
`<CtrlLeft>`	`ccPageLeft`
`<CtrlRight>`	`ccPageRight`
`<PgUp>`	`ccPrevPage`
`<Right>`	`ccRight`

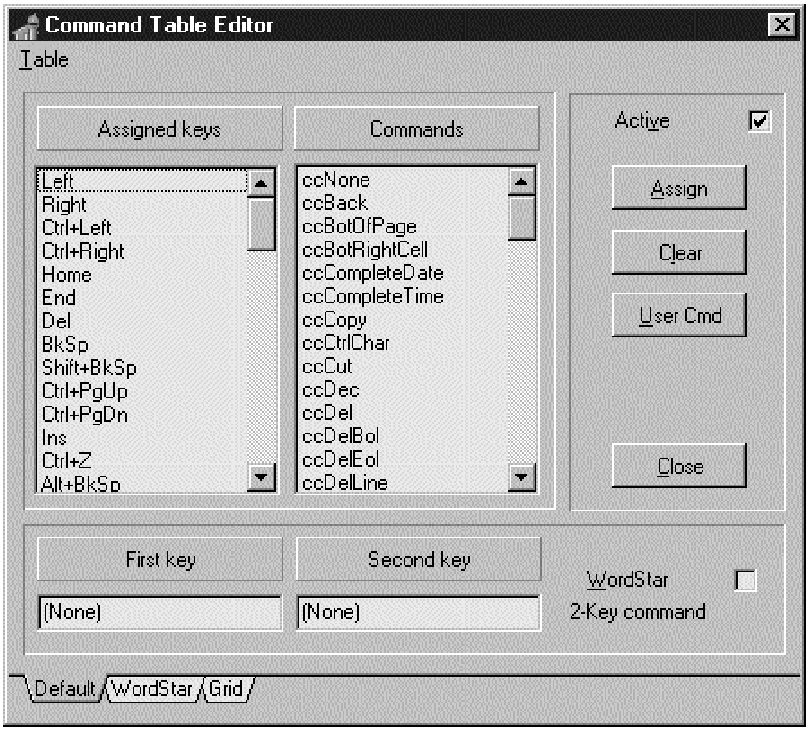
Table 2.4: *Grid Command Table (continued)*

Key Sequence	Command
`<CtrlUp>`	`ccTopLeftCell`
`<CtrlHome>`	`ccTopOfPage`
`<Up>`	`ccUp`

Command Table Editor

Nearly all operations supported by the TOvcCommandProcessor are accessible through the Command Table Editor. It is made available to the Orpheus components through the EntryCommands property of the TOvcController component.

This form allows you to configure command tables and the command-key assignments for use by the Orpheus components on the form (see Figure 2.1).



Left
Right
Ctrl+Left
Ctrl+Right
Home
End
Del
BkSp
Shift+BkSp
Ctrl+PgUp
Ctrl+PgDn
Ins
Ctrl+Z
Alt+BkSp

ccNone
ccBack
ccBotOfPage
ccBotRightCell
ccCompleteDate
ccCompleteTime
ccCopy
ccCtrlChar
ccCut
ccDec
ccDel
ccDelBol
ccDelEol
ccDelLine

Active: ✓
First key: (None)
Second key: (None)
WordStar 2-Key command: ☐

Figure 2.1: Command Table Editor

Assigned keys

This list box displays the currently assigned key sequences. Selecting one of the key sequences in this list box also selects the command that is assigned to the key sequence in the Commands list box.

Commands

This list box displays all available commands. Note that not all commands are handled by each Orpheus component. See “Commands” on page 50 for information about which commands are supported for each of the Orpheus components.

Active

This check box indicates if the command table is active or not. If this check box is checked, then the command processor’s Translate method will scan this table when called.

First key and second key

These special edit components display the key name of the key or keys that are pressed rather than the actual character. For example, if you hold the control key and press F1, “Ctrl+F1” is displayed.

WordStar 2-Key command

This check box indicates that the entered key sequence should be treated as a WordStar-style command. A WordStar key sequence has two parts. The first part is usually something like <CtrlQ> and the second is a single letter or number. When a WordStar-style key sequence is entered by the user, the control key can be used with the first part only or with both parts. For example, <CtrlQ><A> and <CtrlQ><CtrlA> are both acceptable.

If you check this box and the entered keys are not an appropriate WordStar-style key sequence, an error is generated and the check box is cleared.

Assign

This button maps (or assigns) the key sequence entered in the “First key” and “Second key” edit fields to the selected command in the “Commands” list box.

Clear

This button clears (or deletes) an existing command-key mapping from the command table. The selected key sequence in the “Assigned keys” list box is removed from the table.

UserCmd

This button assigns the key sequence entered in the “First key” and “Second key” edit fields as a user-defined command. User-defined commands allow you to provide special actions when the key sequence is entered by the user. User-defined commands are available for those Orpheus components that publish an OnUserCommand event (for example, the Orpheus entry fields).

When you press “User Cmd”, the next available user-defined command is assigned and displayed in the “Commands” list box. Valid user-defined commands are ccUserFirst to ccUser65535. ccUserFirst is defined in the OvcConst unit.

Help

This button displays help text for the Command Table Editor.

Close

This button exits and closes the form. Changes made to the command tables are immediately transferred to the project form. Using this button or closing the form using the system menu will not undo any changes that were made previously. If you need to undo changes, close the current project without saving.

Default, WordStar, and Grid Tabs

These tabs are used to select a command table to view or modify.

Command Table Editor menu options

The only item in the Command Table Editor menu is Table (see Figure 2.2). It gives you access to a group of operations involving command tables.

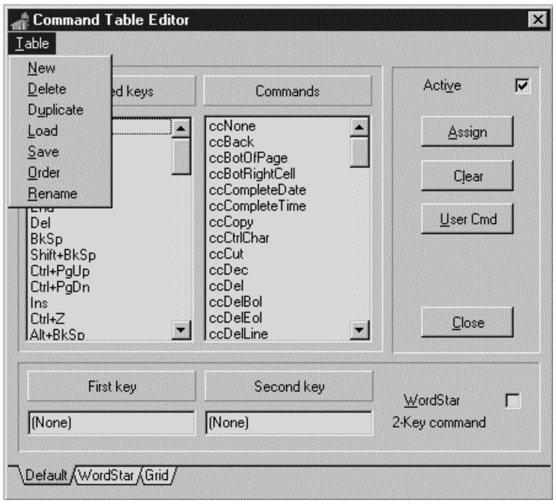


Figure 2.2: Command Table Editor menu option

New

This menu option creates a new (empty) command table using "NewTable" for the name. A number is appended to the end of the name if another table with the same name already exists. Use Rename to change the name of the command table.

Delete

This menu option deletes the current command table (the selected tab) from the command processor's internal list of command tables. You are asked for confirmation before the command table is deleted.

Caution: There is no undo for this operation.

Duplicate

This menu option creates a new command table (as described in Table|New). It then copies all existing command-key mappings from the current command table (the selected tab) to the new command table.

Load

This menu option allows you to load a previously stored command table from a file. If a table with the same name already exists, the loaded table name is changed by appending a number to it.

Save

This menu option writes the current command table (the selected tab) to a file.

Order

This menu option displays the “Change Scan Order” form (see Figure 2.3).

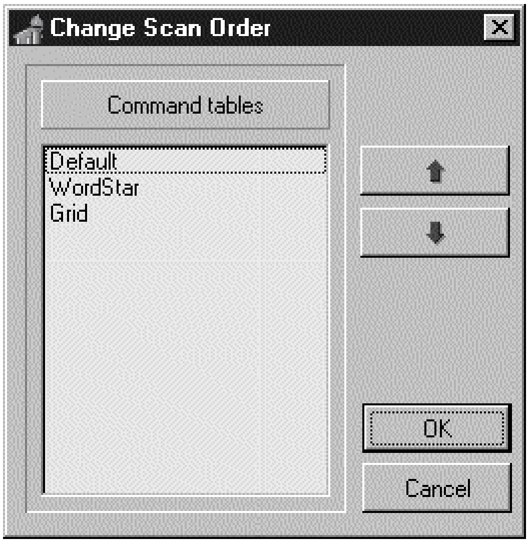


Figure 2.3: ChangeScan Order form

This form is used to change the order in which command tables are shown on the tabs at the bottom of the Command Table Editor. More importantly, this changes the order in which the tables are scanned when the command processor is searching for a key sequence. If a key sequence is contained in more than one command table, the command processor’s search terminates when it finds the first match.

To change the order, select a command table and then use the up or down arrow buttons to reposition it.

Rename

This menu option displays the “Rename Table” form.

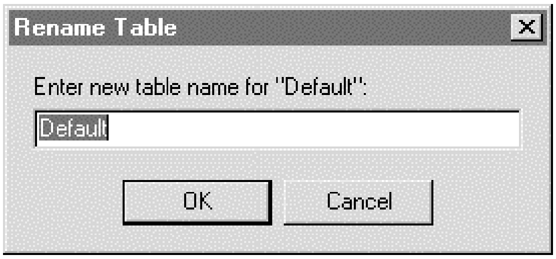


Figure 2.4: Rename Table form

Enter the new name for the current command table (the selected tab) and press OK to change the name, or press Cancel to exit without changing the name.

Hierarchy

TPersistent (VCL)

  TOvcCommandProcessor (OvcCmd)

Properties

Count	Table

Methods

Add	DeleteCommand	ResetCommandProcessor
AddCommand	DeleteCommandTable	SaveCommandTable
AddCommandRec	Exchange	SetScanPriority
ChangeTableName	GetCommandCount	Translate
Clear	GetCommandTable	TranslateKey
CreateCommandTable	GetCommandTableIndex	TranslateKeyUsing
Delete	LoadCommandTable	TranslateUsing

Add method

```
procedure Add(CT : TOvcCommandTable);
```

Add a command table to the list of tables.

This method adds the TOvcCommandTable object specified by CT to the list of command tables maintained by the command processor. Multiple command tables can be created to increase functionality or to allow use of different commands.

See also: AddCommand, CreateCommandTable

AddCommand method

```
procedure AddCommand(
  const TableName : string; Key1, ShiftState1, Key2,
  ShiftState2 : Byte; Command : Word);
```

Add a command and key sequence to the command table.

TableName is the name of one of the command tables currently maintained by the command processor object. The table name is not case sensitive. Key1 and optionally Key2 contain the key codes. ShiftState1 and ShiftState2 contain the shift-state constants if shift keys must be pressed. Command is the command to associate with the key sequence. If there is no second key, set Key2 and ShiftState2 both to zero.

If the specified TableName does not exist, an ETableNotFound exception is raised. If the key sequence specified in Key1, ShiftState1, Key2, and ShiftState2 is already assigned within TableName, an EDuplicateCommand exception is raised.

See the VCL Messages unit or HPP file, or the online help system for information about the constants used to identify keys. Some virtual key codes used in the tables are not defined by the VCL. They are defined in OVCCONST.PAS (VK_A, the 'a' key, is a simple example of one of these constants).

The possible values for ShiftState1 and ShiftState2 are listed in the following table:

Constant	**ShiftState1 Value**	**ShiftState2 Value**
ss_None	$00	No shift key.
ss_Shift	$02	<Shift>

Constant	ShiftState1 Value	ShiftState2 Value
ss_Ctrl	$04	<Ctrl>
ss_Alt	$08	<Alt>
ss_Wordstar	$80	The second key of a WordStar command, which is accepted if no shift key is pressed or if the <Ctrl> key is pressed.

The ss_Xxx constants are defined in the OvcData unit.

The following example adds a key mapping for the <ShiftAltF5> key sequence to the DEFAULT command table. When the command processor sees this key sequence, it will return the ccUser5 command.

```
  const
  ccUser5 = ccUserFirst + 5;
  ...
  AddCommand('DEFAULT', VK_F5, ss_Shift +  ss_Alt, 0, 0, ccUser5);
```

The following example adds a WordStar-style key sequence to the WORDSTAR command table. After executing this command, pressing <CtrlQ><S> or <CtrlQ><CtrlS> returns the ccHome command.

```
  AddCommand('WORDSTAR', VK_Q, ss_Ctrl, VK_S, ss_Wordstar, ccHome);
```

See also: AddCommandRec, DeleteCommand

AddCommandRec method

```
procedure AddCommandRec(
  const TableName : string; const CR : TOvcCmdRec);
```

Add a command record to the command table.

This procedure is identical to the AddCommand procedure except that the key and command information is passed as a TOvcCmdRec instead of in individual parameters.

If the specified TableName does not exist, an ETableNotFound exception is raised. If the key sequence specified in CR is already assigned within TableName, an EDuplicateCommand exception is raised.

TOvcCmdRec has the following format:

```
TOvcCmdRec = record
case Byte of
    0 : (Key1  : Byte;  {first key’s virtual key code}
         SS1   : Byte;  {shift state of first key}
         Key2  : Byte;  {second key’s virtual key code}
         SS2   : Byte;  {shift state of second key}
         Cmd   : Word); {command to return for this entry}
1 : (Keys  : LongInt);  {for sorting, searching, and storing}
end;
```

TOvcCmdRec is defined in the OvcData unit.

See also: AddCommand

ChangeTableName — method

```
procedure ChangeTableName(const OldName, NewName : string);
```

↳ ChangeTableName changes the name of a command table.

The name of the command table is changed from OldName to NewName. If the specified OldName does not exist, an ETableNotFound exception is raised.

The following example renames the DEFAULT table to MYTABLE:

```
ChangeTableName('DEFAULT', 'MYTABLE');
```

Clear — method

```
procedure Clear;
```

↳ Clear removes all tables from the internal command table list.

This procedure removes all command tables from the internal command table list, in contrast to Delete or DeleteCommandTable, both of which remove only one table at a time.

See also: Delete, DeleteCommandTable

Count — run-time, read-only property

```
property Count : Integer
```

↳ Count is the number of command tables.

This property provides the current number of command tables maintained by the command processor.

CreateCommandTable method

```
function CreateCommandTable(
  const TableName : string; Active : Boolean) : Integer;
```

CreateCommandTable creates a command table and adds it to the list of tables.

This method creates a new command table with the name TableName. The table is marked as active if Active is True. The new command table is initially empty.

The function result is the index into the internal list of command tables. If the table could not be created or TableName already exists, the function result is -1.

The following example creates a new command table named SAMPLE, which is initially active:

```
  CreateCommandTable('SAMPLE', True);
```

See also: AddCommand, AddCommandRec, Delete, DeleteCommandTable

Delete method

```
procedure Delete(Index : Integer);
```

Deletes a command table from the list of tables.

The Delete procedure deletes an existing table from the internal command table list. Index is the location of the command table in the list. If the specified table does not exist, an ETableNotFound exception is raised.

The following example removes the first command table if there are at least two existing command tables:

```
  if Count > 1 then
  Delete(0);
```

See also: CreateCommandTable, DeleteCommandTable

DeleteCommand method

```
procedure DeleteCommand(const TableName : string;
  Key1, ShiftState1, Key2, ShiftState2 : Byte);
```

DeleteCommand deletes a command and key sequence from a command table.

This procedure removes a single entry from the command table specified by TableName. If an entry is found that matches the Key1, ShiftState1, Key2, and ShiftState2 parameters exactly, it is removed from the command table. See AddCommand for a description of these parameters. If the specified TableName does not exist, an ETableNotFound exception is raised.

The following example deletes the command table entry corresponding to the <ShiftAltF5> key sequence from the DEFAULT command table:

```
DeleteCommand('DEFAULT', VK_F5, ss_Shift + ss_Alt, 0, 0);
```

See also: AddCommand

DeleteCommandTable method

```
procedure DeleteCommandTable(const TableName : string);
```

↳ Delete removes a command table from the list of tables.

This procedure deletes the table specified by TableName from the internal list of command tables. If the specified TableName does not exist, an ETableNotFound exception is raised.

The following example deletes the SAMPLE command table from the command processor's command table list:

```
DeleteCommandTable('SAMPLE');
```

Exchange method

```
procedure Exchange(Index1, Index2 : Integer);
```

↳ Exchange swaps the locations of two command tables.

This method swaps the locations of the command tables specified by Index1 and Index2. This is normally done so that one table is scanned prior to the other.

The following example swaps the location of the tables at the index 2 and index 5 positions:

```
Exchange(2, 5);
```

See also: Add, Delete, DeleteCommandTable

GetCommandCount method

```
function GetCommandCount(const TableName : string) : Integer;
```

↳ GetCommandCount returns the number of commands in the command table.

This method gets the number of commands in the command table specified by TableName. If the table does not exist, the function returns zero. If the specified TableName does not exist, an ETableNotFound exception is raised.

The following example tests to see if the DEFAULT command table has at least one entry before doing something:

```
  if GetCommandCount('DEFAULT') > 0 then begin
  {do something}
  end;
```

GetCommandTable method

```
function GetCommandTable(
  const TableName : string) : TOvcCommandTable;
```

↳ GetCommandTable returns a pointer to a command table.

This method gets a pointer to the command table specified by TableName. If the table does not exist, nil is returned as the function result.

The following example gets a pointer to the DEFAULT command table:

```
  var
  CT : TOvcCommandTable;
  ...
  CT := GetCommandTable('DEFAULT');
  if CT <> nil then begin
  {do something with the command table, CT}
  end;
```

GetCommandTableIndex method

```
function GetCommandTableIndex(const TableName : string) : Integer;
```

↳ GetCommandTableIndex returns the index of a command table.

This method searches the command processor's internal list of command tables for TableName and returns its index. If the table does not exist, -1 is returned.

LoadCommandTable virtual method

```
function LoadCommandTable(const FileName : string) : Integer;
```

↳ LoadCommandTable creates a command table and fills it from a text file.

This function creates a new command table and then fills it using the contents of the text file FileName. If LoadCommandTable is successful, the function result is the index into the internal command table list. If it is not successful, the function result is -1.

The text file specified by FileName must have the following format:

```
TableName
Key1 Shift1 Key2 Shift2 ccCommand
Key1 Shift1 Key2 Shift2 ccCommand
...
```

TableName is a string representing the name of the table and is assigned as the name of the table when the command table is loaded. Key1 and Key2 are the standard virtual key code values (VK_Xxx constants) defined in the Windows unit. Shift1 and Shift2 represent the shift-state flag values (ss_xxx constants) associated with Key1 and Key2 respectively. See the AddCommand method on page 68 for information about the shift-state flags. ccCommand is the command assigned to the key sequence. See “Commands” on page 50 for a complete list of the available commands and their values.

The following example reads the table name and two command records from the SAMPLE.CMD text file, assigns “Default” to the TableName property, and inserts the command records into the internal command record list.

Contents of SAMPLE.CMD:

```
Default
           37   0   0   0    23
           39   0   0   0    32
           ...

LoadCommandTable('SAMPLE.CMD');
```

See also: AddCommand, SaveCommandTable

ResetCommandProcessor method

```
procedure ResetCommandProcessor;
```

ResetCommandProcessor re-initializes the command processor.

This procedure re-initializes the command processor for the next call to Translate or TranslateUsing. ResetCommandProcessor is called to ensure that previous partial commands are cleared (for example, the partial commands involved with WordStar style key sequences).

See also: Translate, TranslateUsing

SaveCommandTable — virtual method

```
procedure SaveCommandTable(const TableName, FileName : string);
```

SaveCommandTable saves a command table to a text file.

This procedure writes the name and command-key assignments from TableName to the text file specified by FileName. The format of the resulting file is shown in LoadCommandTable. If the specified TableName does not exist, an ETableNotFound exception is raised.

If a file with the same name already exists, it is overwritten without warning.

The following example writes the contents of the DEFAULT command table to a text file named MYTABLE.CMD:

```
  SaveCommandTable('DEFAULT', 'MYTABLE.CMD');
```

See also: LoadCommandTable

SetScanPriority — method

```
procedure SetScanPriority(Names : array of string);
```

SetScanPriority reorders the list of command tables.

This procedure is used to change the order in which the command tables are scanned. Names is a string array containing table names in the desired scan order. The strings contained in Names are not case sensitive. If a specified table name does not exist, an ETableNotFound exception is raised.

The following example rearranges the command tables so that MYTABLE is scanned first, followed by DEFAULT and WORDSTAR. It assumes that all three of the command tables are contained in the list of command tables.

```
  SetScanPriority(['MYTABLE', 'DEFAULT', 'WORDSTAR']);
```

See also: TranslateUsing

Table — run-time property

```
property Table[Index : Integer] : TOvcCommandTable
```

Table allows access to command tables.

The Table property allows access to the command tables stored in the command processor object.

The following example gets a pointer to the first command table if there is at least one table in the command processor:

```
var
CT : TOvcCommandTable;
...
if Count > 0 then
CT := Table[0];
```

See also: Count

Translate method

```
function Translate(var Msg : TMessage) : Word;
```

Translate converts Windows messages to commands.

This method is used to convert one of several types of Windows messages into a command recognizable by other components. TMessage is defined by the VCL.

Msg can be the message result from receiving any one of the following Windows messages:

```
WM_CHAR
WM_KEYDOWN
WM_SYSKEYDOWN
```

The following example translates WM_KEYDOWN messages into commands:

```
procedure TEntryField.WMKeyDown(var Msg : TWMKeyDown);
var
  Cmd : Word;
begin
  {see if this command should be processed by us}
  Cmd := Controller.EntryCommands.Translate(TMessage(Msg));
  if Cmd <> ccNone then
  {do something with the command}
else
inherited;
end;
```

See also: ResetCommandProcessor, TranslateKey, TranslateKeyUsing, TranslateUsing

TranslateKey method

```
function TranslateKey(Key : Word; ShiftState : TShiftState) : Word;
```

↳ TranslateKey converts virtual key codes to commands.

This method is used to convert one of several virtual key codes into a command recognizable by other components. Key is a virtual key code. TShiftState is defined in the VCL's Classes unit.

See also: ResetCommandProcessor, Translate, TranslateUsing, TranslateKeyUsing

TranslateKeyUsing method

```
function TranslateKeyUsing(
  const Tables : array of string; Key : Word;
  ShiftState : TShiftState) : Word;
```

↳ TranslateUsing translates a virtual key code into a command using the supplied tables.

This function is used instead of the normal Translate or TranslateKey method when you want to translate a virtual key code into a command but don't want to use the current scan order. Call TranslateKeyUsing, passing it the names of the command tables in the order you want them scanned. If you need to use the same scan order for several calls, you should first call SetScanPriority and then use TranslateKey, because TranslateKey is slightly faster than TranslateKeyUsing.

All command tables specified in the Tables parameter are scanned regardless of the state of each command table's Active property. Therefore, this method can be used to force the scanning of tables that would not be scanned by Translate or TranslateKey. If a specified table name does not exist, an ETableNotFound exception is raised.

Key is a virtual key code. TShiftState is defined in the VCL's Classes unit.

See also: ResetCommandProcessor, SetScanPriority, Translate, TranslateKey, TranslateUsing

TranslateUsing method

```
function TranslateUsing(
  Tables : array of string; var Msg : TMessage) : Word;
```

↳ TranslateUsing translates a message into a command using the supplied tables.

This function is used instead of the normal Translate or TranslateKey method when you want to translate a message into a command but don't want to use the current scan order. Call TranslateUsing, passing it the names of the command tables in the order you want them scanned. If you need to use the same scan order for several calls, you should first call SetScanPriority and then use Translate, because Translate is slightly faster than TranslateUsing.

All command tables specified in the Tables parameter are scanned regardless of the state of each command table's Active property. Therefore, this method can be used to force the scanning of tables that would not be scanned by Translate. If a specified table name does not exist, an ETableNotFound exception is raised.

TMessage is defined by the VCL.

The following example finds the command corresponding to the message stored in Msg and assigns the result to the Cmd variable. The WORDSTAR Command Table is scanned first.

```
var
  Cmd : Word;
...
Cmd := TranslateUsing(['WORDSTAR', 'DEFAULT'], Msg)
```

See also: ResetCommandProcessor, SetScanPriority, Translate, TranslateKey, TranslateKeyUsing

TOvcCommandTable Class

TOvcCommandTable provides access to commands used by the Orpheus command processor (TOvcCommandProcessor) on page 50. TOvcCommandTable provides several properties and methods that allow access to the keyboard-to-command translations used by the command processor. The Command list contains instances of the TOvcCmdRec structure.

Hierarchy

TPersistent (VCL)

  TOvcCommandTable (OvcCmd)

Properties

Commands	IsActive
Count	TableName

Methods

AddRec	Exchange	LoadFromFile
Clear	IndexOf	Move
Delete	InsertRec	SaveToFile

Reference Section

AddRec — method

```
function AddRec(const CR : TOvcCmdRecc) : Integer;
```

↳ AddRec adds a TOvcCmdRec to the internal list of commands.

Clear — method

```
procedure Clear;
```

↳ Clear deletes all command records from the internal list.

Commands — property

```
property Commands[Index : Integer] : TOvcCmdRecc
```

↳ Commands provides access to the internal list of command records.

Index represent the item in the command list. Valid values are from 0 to Count - 1.

See also: Count

Count — property

```
property Count : Integer
```

↳ Count returns the number of commands stored in the command list.

Delete — method

```
procedure Delete(Index : Integer);
```

↳ Delete removes the item specified by Index from the command list.

Exchange — method

```
procedure Exchange(Index1, Index2 : Integer);
```

↳ Exchange switches the list locations of the two specified records.

IndexOf — method

```
function IndexOf(const CR : TOvcCmdRecc) : Integer;
```

↳ IndexOf returns the index of the specified record.

InsertRec method

```
procedure InsertRec(Index : Integer; const CR : TOvcCmdRecc);
```

InsertRec adds the record specified by CR to the list of commands at the specified Index.

IsActive property

```
property IsActive : Boolean
```

IsActive determines whether the command table is used.

If IsActive is False, the table is not considered when the command processor is performing key-to-command translations.

LoadFromFile method

```
procedure LoadFromFile(const FileName: string);
```

LoadFromFile reads command entries from a text file.

Move method

```
procedure Move(CurIndex, NewIndex : Integer);
```

Move physically moves the item specified by CurIndex to the NewIndex location in the command list.

SaveToFile method

```
procedure SaveToFile(const FileName: string);
```

SaveToFile writes the contents of the command list to a text file.

TableName property

```
property TableName : string
```

TableName determines the name of the command table.

TOvcGraphicControl Class

TOvcGraphicControl is the base class for all graphic components in Orpheus. It implements certain common data structures used internally by Orpheus and the About property which shows the current version.

Hierarchy

TGraphicControl (VCL)

  TOvcGraphicControl (OvcBase)

Properties

  About

Reference Section

About read-only property

```
property About : string
```

About shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. The Orpheus about box can be displayed by double-clicking this property or selecting the dialog button to the right of the property value.

Date and Time Routines

The StDate unit is borrowed from our SysTools product to provide routines that allow you to store dates and times in compact formats, to convert them to other forms, and to perform date and time arithmetic.

The term “Julian date” means different things to different people. Among astronomers and historians, it has a very precise and technical meaning. Among programmers, however, the term has acquired a much broader meaning, something like “a date stored as an integer value representing the serial number of days from a given point in time.” It is in that widely understood sense that the term is used here.

StDate defines date and time data types:

```
TStDate = LongInt;
TStTime = LongInt;
```

Dates are stored in long integer format as the number of days since January 1, 1600. Times are represented as the number of seconds since midnight.

When one of the StDate routines needs to return an error to indicate an invalid date or time, invalid date values are represented by BadDate (-1) and invalid time values are represented by BadTime (-1).

Procedures/Functions

These are the procedures and functions provided by the StDate unit. The following reference section has detailed descriptions for each one.

AstJulianDate	DayOfWeekDMY	RoundToNearestMinute
AstJulianDatePrim	DaysInMonth	StDateToDateTime
AstJulianDateToStDate	DecTime	StDateToDMY
BondDateDiff	DMYtoStDate	StTimeToDateTime
Convert2ByteDate	HMStoStTime	StTimeToHMS
Convert4ByteDate	IncDate	TimeDiff
CurrentDate	IncDateTime	ValidDate
CurrentTime	IncDateTrunc	ValidTime
DateDiff	IncTime	WeekOfYear
DateTimeDiff	IsLeapYear	
DayOfWeek	RoundToNearestHour	

Reference Section

AstJulianDate function

```
function AstJulianDate(Julian : TStDate) : Double;
```

AstJulianDate converts a TStDate to an astronomical Julian Date.

This function returns the number of days since the astronomical Julian Date zero point (1/1/4173BC). Astronomical Julian Dates are stored as the number of whole and fractional days since this zero point.

The astronomical Julian Day starts at noon, not midnight. Therefore, for any given date at midnight, the decimal part of the AstJulianDate result is .5 (not .0).

The following example returns 2449600.5:

```
  AJD := AstJulianDate(
  DateStringToStDate('mm/dd/yyyy',
  '10/10/1995', 1950));
```

See also: AstJulianDateToStDate

AstJulianDatePrim function

```
function AstJulianDatePrim(
  Year, Month, Date : Integer; UT : TStTime) : Double;
```

Returns an Astronomical Julian Date for any year, even those outside MinYear..MaxYear

AstJulianDateToStDate function

```
function AstJulianDateToStDate(
  AstJD : Double; Truncate : Boolean) : TStDate;
```

AstJulianDateToStDate converts an astronomical Julian Date to a TStDate.

Setting Truncate to True truncates the intermediate function result (i.e., it always rounds down). Setting it to False rounds the final result to the nearest whole number. If the value computed by the function is outside the range of MinDate..MaxDate, then BadDate is returned.

The following example computes the SysTools Date from Julian date 2450000.8 and returns 144552 if Truncate is True and 144553 if False:

```
  T1 := AstJulianDateToStDate(2450000.8, True);
```

See also: AstJulianDate, DateStringHMStoAstJD

```
function BondDateDiff(
  Date1, Date2 : TStDate; DayBasis : TStBondDateType) : TStDate;
```

```
TStBondDateType = (bdtActual, bdt30E360, bdt30360)
```

BondDateDiff calculates the number of days between two dates using one of three financial methods.

If Date1 is greater than Date2, the function swaps them before computing the value.

The three financial methods that can be used are Actual, Eurobond, and Old Eurobond. Actual uses the actual number of days between the two dates. Eurobond assumes that all months have 30 days. If one or both of the dates is for the 31st of the month, it is assumed for the purpose of the calculation to be the 30th. Old Eurobond goes a step further by checking the day part of the date, Day. If Day1 = 31, it is set to 30, e.g., July 31 becomes July 30. Then, if Day2 = 31 and Day1 >= 30, Day2 is set to 30. For example, if computing from July 31, 1995 to August 31, 1997, Day1 is set to 30 as is Day2. On the other hand, if going from February 28, 1995 to August 31, 1997, Day1 is 28 and Day2 is 31.

The three possible values for DayBasis are listed in the following table:

Value	**Description**
bdtActual	The actual number of days.
bdt30E360	Eurobond. The number of days based on day months and 360 days/year. This is the Eurobond method.
bdt30360	31 day months and 360 days/year. This is the Old Eurobond method.

The following example computes the number of days between two dates using the specified TStBondDateType:

```
var
  DeltaT : TStDate;
  DB : TStBondDateType;

  DB := bdt30E360;
  DeltaT := BondDateDiff(
  DateStringToStDate('mm/dd/yyyy', '10/10/1995', 1950),
  DateStringToStDate('mm/dd/yyyy', '04/15/1996', 1950), DB);
```

See also: DateDiff

Convert2ByteDate function, StDate

```
function Convert2ByteDate(TwoByteDate : Word) : TStDate;
```

Convert2ByteDate converts an Object Professional two-byte date to a TStDate four-byte date.

This routine is provided primarily for the convenience of users of Turbo Professional, Object Professional, and Object Professional for C++. A Julian date value created by these packages can be in a two byte date format. Convert2ByteDate converts this two byte date format into the four byte format used by StDate.

The following example converts a two-byte date to a four-byte date:

```
var
  OldDate : Word;
  NewDate : TStDate;

NewDate := Convert2ByteDate(OldDate);
```

See also: Convert4ByteDate

Convert4ByteDate function

```
function Convert4ByteDate(FourByteDate : Date) : Word;
```

Convert4ByteDate converts a TStDate to an Object Professional two-byte date.

This routine is provided primarily for the convenience of users of Turbo Professional, Object Professional, and Object Professional for C++. It converts a Julian date value created with StDate (four-byte format) into the two-byte format implemented optionally in those packages.

This routine can also be used if you want to minimize the amount of space occupied by date variables, and the dates fall into a limited range. The two-byte format is used to represent dates in the range January 1, 1900 to December 31, 2078.

The following example converts a four-byte date into a two-byte date:

```
var
  Date2 : Word;
  Date4 : TStDate;

Date2 := Convert4ByteDate(Date4);
```

See also: Convert2ByteDate

CurrentDate function

```
function CurrentDate : TStDate;
```

↳ CurrentDate returns today's date as a Julian date.

This routine calls Windows to determine the current date, then converts the returned values (day, month, and year) to a TStDate variable, which is returned as the function result.

The following example retrieves the current date and stores it in the variable D:

```
var
  D : TStDate;
...
D := CurrentDate;
```

See also: CurrentTime

CurrentTime function

```
function CurrentTime : TStTime;
```

↳ CurrentTime returns the current time in seconds since midnight.

This routine calls Windows to determine the current time of day, then converts the values it returns (hours, minutes, and seconds) to a TStTime variable, which is returned as the function result.

The following example calculates the number of seconds that elapsed between two events happening within the same day:

```
var
  Start, Elapsed : TStTime;
...
Start := CurrentTime;
...
Elapsed := CurrentTime - Start;
```

See also: CurrentDate

DateDiff function

```
procedure DateDiff(
  Date1, Date2 : TStDate; var Days, Months, Years : Integer);
```

↳ DateDiff returns the difference in days, months, and years between two valid Julian dates.

This routine computes the absolute number of days, months, and years, between Date1 and Date2. The order of Date1 and Date2 is not important. To calculate the number of days only, subtract the smaller date from the larger.

The following example sets Days = 1, Months =0, and Years = 0:

```
  DateDiff(Today, Today+1, Days, Months, Years);
```

See also: BondDateDiff, DateTimeDiff, TimeDiff

DateTimeDiff procedure, StDate

```
procedure DateTimeDiff(var DT1, DT2 : TStDateTimeRec;
  var Days : LongInt; var Secs : LongInt);
```

```
TStDateTimeRec = record
  D : TStDate;
  T : TStTime;
end;
```

DateTimeDiff returns the difference in days and seconds between two points in time.

The function returns the absolute difference between the two dates. Therefore, the order of the two is not important.

The following example sets Days = 1 and Secs = 1:

```
  var
    DT1, DT2 : TStDateTimeRec;
    Days, Secs : LongInt;
  begin
    DT1.D := Today;
    DT1.T := CurrentTime;
    DT2.D := IncDate(DT1.D, 1, 0, 0);
    DT2.T := IncTime(DT1.T, 0, 0, 1);
    DateTimeDiff(DT1, DT2, Days, Secs);
  end;
```

See also: CurrentDate, CurrentTime, DateDiff, IncDate, IncTime, TimeDiff

DayOfWeek function

```
function DayOfWeek(Julian : TStDate) : TDayType;
```

```
TDayType = (
  Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
```

DayOfWeek returns the day of the week for a Julian date.

If Julian is not a valid date, DayOfWeek returns an invalid ordinal value of 7, which causes a range exception (if range checking is turned on) when used.

The following example obtains a string representing the current day of the week. Since TDayType is an enumerated data type, the first element (Sunday) has an ordinal value of 0. Adding 1 to this value yields the correct index into the array of day names, which starts at 1.

```
  var
      S : string;
  ...
  S := LongDayNames[Ord(DayOfWeek(CurrentDate))+1];
```

See also: CurrentDate, DayOfWeekDMY

DayOfWeekDMY function

```
function DayOfWeekDMY(
  Day, Month, Year : Integer; Epoch : Integer) : TDayType;
```

```
TDayType = (
  Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
```

↳ DayOfWeekDMY returns the day of the week for the day, month, and year.

If the date represented by Day, Month, and Year is not valid, DayOfWeekDMY returns an invalid ordinal value of 7, which causes a range exception (if range checking is turned on) when used.

If Year is less than 100 (a two-digit year), Epoch must be a four-digit year defining the starting year of the 100-year period. See the “The Epoch property” on page 321 for more information.

The following examples both return Friday (ordinal value 5):

```
  DOW := DayOfWeekDMY(13, 11, 1987, 0);
  DOW := DayOfWeekDMY(13, 11, 87, 1960);
```

See also: DayOfWeek

DaysInMonth function

```
function DaysInMonth(Month, Year : Integer) : Integer;
```

↳ DaysInMonth returns the number of days in the specified month and year.

The following example returns 29, since 1992 is a leap year:

```
  var
    Days : Integer;
  ...
  Days := DaysInMonth(2, 1992);
```

DecTime function

```
function DecTime(
  T : TStTime; Hours, Minutes, Seconds : Byte) : TStTime;
```

DecTime subtracts the specified hours, minutes, and seconds from the specified time of day.

The result of DecTime is adjusted to account for the rollover at midnight.

The following example sets HourAgo equal to 01:00:00 am and ThreeHoursAgo to 11:00:00 pm (note the rollover):

```
  var
    RightNow : TStTime;
    HourAgo : TStTime;
    ThreeHoursAgo : TStTime;
  ...
  RightNow := HMStoStTime(2, 0, 0); {02:00:00 am}
  HourAgo := DecTime(RightNow, 1, 0, 0);
  ThreeHoursAgo := DecTime(RightNow, 3, 0, 0);
```

See also: HMStoStTime, IncTime

DMYtoStDate function

```
function DMYtoStDate(
  Day, Month, Year : Integer; Epoch : Integer) : TStDate;
```

DMYtoStDate converts from day, month, year to a Julian date.

If the date is invalid, BadDate is returned.

If Year is less than 100 (a two-digit year), Epoch must be a four-digit year defining the starting year of the 100-year period. See the “The Epoch property” on page 321 for more information.

The following example sets DateToRemember equal to the number of days between January 1, 1600 and November 13, 1987:

```
  var
  DateToRemember : TStDate;
  ...
  DateToRemember := DMYtoStDate(13, 11, 1987, 0);
```

See also: StDateToDMY

HMStoStTime function

```
function HMStoStTime(Hours, Minutes, Seconds : Byte) : TStTime;
```

↳ HMStoStTime converts from hours, minutes, seconds to a SysTools time value.

Given a time of day in Hours, Minutes, and Seconds, this routine returns the time in seconds since midnight. Hours is in 24-hour format. If the time is invalid, BadTime is returned.

See also: StTimeToHMS

IncDate function

```
function IncDate(
  Julian : TStDate; Days, Months, Years : Integer) : TStDate;
```

↳ IncDate adjusts a Julian date by the specified number of days, months, and years.

IncDate adds (or subtracts) the specified number of Days, Months, and Years to (or from) a date. Months and Years are added before Days, and no overflow or underflow checks are made.

The following example demonstrates one tricky case involving IncDate, the case where Julian represents a date at the end of a month and the addition of one or more months can result in a day of the month that doesn't exist in the new month. The result should be 02/29/1993, but that date doesn't exist. So what you get is a seemingly illogical result that is consistent with IncDate's simplistic philosophy of date arithmetic: 03/01/1993. That is, the value in D2 would represent 02/29/1993 if there were such a date, but because February has only 28 days in that year, the result is (02/28/1993)+1, or 03/01/1993. If this behavior is unacceptable for your application, use IncDateTrunc instead.

```
var
D1 : TStDate;
  D2 : TStDate;
...
D1 := DMYtoStDate(29, 1, 1993); {01/29/1993}
D2 := IncDate(D1, 0, 1, 0);
```

IncDateTime procedure

```
procedure IncDateTime(
  dT1 : TStDateTimeRec; var DT2 : TStDateTimeRec;
  Days : Integer; Secs : LongInt);
```

```
TStDateTimeRec = record
  D : TStDate;
  T : TStTime;
end;
```

IncDateTime adjusts a date and time by the specified number of days and seconds.

If DT1 is a valid date, this routine increments or decrements the date field by Days and the time field by Time. The result is placed in DT2. To decrement a date, use negative values for both Days and Secs. The following example adds the equivalent of 8 hours to StartTime and returns the computed date and time in StopTime:

```
  var
    StartTime, StopTime : TStDateTimeRec;
  ...
  IncDateTime(StartTime, StopTime, 0, 8*SecondsInHour);
```

See also: IncDate, IncTime

IncDateTrunc function

```
function IncDateTrunc(
  Julian : TStDate; Months, Years : Integer) : TStDate;
```

IncDateTrunc adjusts a Julian date by the specified number of months and years.

IncDateTrunc adds or subtracts the specified number of months and years to or from a date. This routine is essentially the same as IncDate, but with two important differences. First, it takes no Days parameter, just Months and Years. And second, it truncates the day of the month in cases where the addition of Months results in an invalid date.

The following example demonstrates what is meant by the word truncate. Because 02/29/1993 does not exist, D2 is set to 03/01/1993 and D3 is set to 02/28/93.

```
var
  D1 : TStDate;
  D2 : TStDate;
  D3 : TStDate;
...
D1 := DMYtoStDate(29, 1, 1993); {01/29/1993}
D2 := IncDate(D1, 0, 1, 0);
D3 := IncDateTrunc(D1, 1, 0);
```

See also: DMYtoStDate, IncDate

IncTime function

```
function IncTime(
  T : TStTime; Hours, Minutes, Seconds : Byte) : TStTime;
```

↳ IncTime adds the specified hours, minutes, and seconds to the specified time of day.

The result of IncTime is adjusted to account for the rollover at midnight.

The following example sets HourFromNow equal to 11:00:00 pm and ThreeHoursFromNow to 1:00:00 am (note the rollover):

```
var
  RightNow : TStTime;
  HourFromNow : TStTime;
  ThreeHoursFromNow : TStTime;
...
RightNow := HMStoStTime(22, 0, 0); {10:00:00 pm}
HourFromNow := IncTime(RightNow, 1, 0, 0);
ThreeHoursFromNow := IncTime(RightNow, 3, 0, 0);
```

See also: DecTime, HMStoStTime

IsLeapYear function

```
function IsLeapYear(Year : Integer) : Boolean;
```

↳ IsLeapYear returns True if the specified year is a leap year.

Year is assumed to be in the range 1600 to 3999 and no error checking is done.

RoundToNearestHour function

```
function RoundToNearestHour(
  const T : TStTime; Truncate : Boolean) : TStTime;
```

↳ RoundToNearestHour rounds the given time to the nearest hour.

If Truncate is True, the function sets the minutes and seconds in T to 0 and returns the result (e.g., 4:31:02 becomes 4:00:00). If Truncate is False, the time is rounded to the nearest hour (e.g., 4:31:02 becomes 5:00:00). T is assumed to be a valid time variable.

If Truncate is False and the minutes part of the time is 30 and the seconds part is 0, the function rounds up.

See also: RoundToNearestMinute

RoundToNearestMinute function

```
function RoundToNearestMinute(
  const T : TStTime; Truncate : Boolean) : TStTime;
```

↳ RoundToNearestMinute rounds the specified time to the nearest minute.

If Truncate is True, the function sets the seconds in T to 0 and returns the result (e.g., 4:01:32 becomes 4:01:00); if False, it rounds the time to the nearest minute (e.g., 4:01:32 becomes 4:02:00). T is assumed to be a valid time variable. If Truncate is False and the seconds part is of the time is 30, the function rounds up.

See also: RoundToNearestHour

StDateToDateTime function

```
function StDateToDateTime(D : TStDate) : TDateTime;
```

↳ StDateToDateTime converts a SysTools date to a VCL date/time.

This routine converts a SysTools TStDate value to a VCL TDateTime value, which is returned as the function result. The time (decimal) part of the TDateTime value is set to 0. No validity checks are performed on the date specified by D.

See also: DateTimeToStDate, StTimeToDateTime

StDateToDMY procedure

```
procedure StDateToDMY(
  Julian : TStDate; var Day, Month, Year : Integer);
```

StDateToDMY converts a Julian date to day, month, year.

If Julian equals BadDate, then Day, Month, and Year are all set to 0.

The following example sets Day, Month, and Year to today's date:

```
  var
    Day : Integer;
    Month : Integer;
    Year : Integer;
  ...
  StDateToDMY(CurrentDate, Day, Month, Year);
```

See also: CurrentDate, DMYtoStDate

StTimeToDateTime function

```
function StTimeToDateTime(T : TStTime) : TDateTime;
```

StTimeToDateTime converts a SysTools time to a VCL date/time.

This routine converts a SysTools TStTime value to a VCL TDateTime value, which is returned as the function result. The date (whole) part of the TDateTime value is set to 0. No validity checks are performed on the time specified by T.

See also: DateTimeToStTime, StDateToDateTime

StTimeToHMS procedure

```
procedure StTimeToHMS(
  T : TStTime; var Hours, Minutes, Seconds : Byte);
```

StTimeToHMS converts a SysTools time variable to hours, minutes, and seconds.

If T equals BadTime, then Hours, Minutes, and Seconds are all set to 0.

The following example sets Hours, Minutes, and Seconds to the current time:

```
  StTimeToHMS(CurrentTime, Hours, Minutes, Seconds);
```

TimeDiff function

```
procedure TimeDiff(
  const Time1, Time2 : TStTime;
  var Hours, Minutes, Seconds : Byte);
```

TimeDiff returns the absolute difference between two times.

The procedure calculates the number of Hours, Minutes, and Seconds between Time1 and Time2. TimeDiff computes the absolute difference so the order of the times is not important.

The following example sets Hours to 1, Minutes to 2, and Seconds to 3:

```
  var
    Time1, Time2 : TStTime;
    Hours, Minutes, Seconds : Byte;
  begin
    Time1 := CurrentTime;
    Time2 := IncTime(1, 1, 2, 3);
    TimeDiff(Time1, Time2, Hours, Minutes, Seconds);
  end;
```

See also: DateDiff, DateTimeDiff

ValidDate function

```
function ValidDate(
  Day, Month, Year : Integer; Epoch : Integer) : Boolean;
```

ValidDate verifies that the specified day, month, and year is a valid date.

ValidDate returns True if the date specified by Day, Month, and Year is valid and False if it isn’t.

If Year is less than 100 (a two-digit year), Epoch must be a four-digit year defining the starting year of the 100-year period. See the “The Epoch property” on page 321 for more information. If Year is greater than 100, it must be in the range 1600-3999.

ValidTime function

```
function ValidTime(Hours, Minutes, Seconds : Integer) : Boolean;
```

ValidTime verifies that the specified hours, minutes, and seconds represent a valid time.

ValidTime returns True if the time specified by Hours, Minutes, and Seconds is valid and False if it isn’t. Valid times are in the range from 0 hours, 0 minutes, 0 seconds through 23 hours, 59 minutes, 59 seconds.

```
function WeekOfYear(Julian : TStDate) : Byte;
```

WeekOfYear returns the week number of a specified date.

The routine uses the ISO standard for determining the week, which states that the first week of a year is the one that includes the first Thursday of that year. A week is defined as a seven-day period within a calendar year, which starts on Monday.

The following example returns 41:

```
  WOY := WeekOfYear(
  DateStringToStDate('mm/dd/yyyy',
  '10/10/1995', 1950));
```

See also: DayOfWeek

Orpheus Exception Classes

The OvcExcpt unit interfaces exception classes that are used by the Orpheus components. This section describes all those exception classes. Exceptions are used in Orpheus wherever they simplify the source code or enhance the error handling capability of a component or class. Standard VCL exceptions, such as those relating to file input and output, are used whenever appropriate. Orpheus exceptions are used only for conditions that are unique to Orpheus components.

One case in which exceptions are not used is for validation errors in entry fields. Instead, an error-handling event is called to handle the error. This is because the actual error can be generated outside the Orpheus component. For example, entry fields allow you to perform additional validation and report errors that are unknown to Orpheus. If an exception is raised in the OnError event handler, the form sees the exception, not the entry field. Because of this, and to provide a flexible error handling mechanism, exceptions are not used for validation errors.

One point worth noting – there are a couple of cases where Orpheus components and/or demo programs may raise exceptions that are handled internally. This is normal operation, and you will not even notice it happening unless you have the “Break on Exception” option turned on for the compiler’s integrated debugger. This option is on by default, and we recommend turning it off unless you specifically need it for a debugging session.

Base exception

Exception (VCL)

    EOvcException(OvcExcpt)

```
EOvcException = class(Exception)
```

EOvcException is the base Orpheus exception class. All other Orpheus exception classes are derived from EOvcException. This exception is raised for error conditions in the Orpheus property editors only. Errors specific to a particular component or class cause more specific exceptions.

General exceptions

Exception (VCL)

    EOvcException (OvcExcpt)

        ENoTimersAvailable (OvcExcpt)

```
ENoTimersAvailable = class(EOvcException)
```

An Orpheus component requested a Windows timer handle and none are available.

Command processor exceptions

Exception (VCL)

  EOvcException (OvcExcpt)

    ECmdProcessorError (OvcExcpt)

      EDuplicateCommand (OvcExcpt)

      ETableNotFound (OvcExcpt)

```
CmdProcessorError = class(EOvcException)
```

Base exception class for the TOvcCommandProcessor class. The only error conditions that are raised using this base class are ones that are generated at design time.

```
EDuplicateCommand = class(ECmdProcessorError)
```

Raised when you attempt to add a command key assignment that already exists in the command table. The same command can be assigned to different key sequences, but the same key sequence cannot be assigned to more than one command within the same command table.

```
ETableNotFound = class(ECmdProcessorError)
```

The specified command table cannot be found in the internal command table list.

Controller exceptions

Exception (VCL)

  EOvcException (OvcExcpt)

    ENoControllerAssigned (OvcExcpt)

```
EControllerError = class(EOvcException)
```

Base exception class for the TOvcController component. The only error conditions that are raised using this base class are ones that are generated at design time.

```
ENoControllerAssigned = class(EControllerError)
```

Raised during component creation if a TOvcController component is not assigned to the component's Controller property. Components that publish a Controller property must have a TOvcController component assigned.

Editor exceptions

Exception (VCL)

    EOvcException (OvcExcpt)

        EEditorError (OvcExcpt)

            EInvalidLineOrCol (OvcExcpt)

            EInvalidLineOrPara (OvcExcpt)

```
EEditorError = class(EOvcException)
```

Base exception class for the TOvcEditor and TOvcTextFileEditor components. See the OnError event in “TOvcEditor Component” on page 775 for a list of the error conditions that cause this exception to be raised.

```
EInvalidLineOrCol = class(EEditorError)
```

The specified line or column is not within the valid range. Valid line numbers are 1 through the number of lines currently in the editor’s text stream. Valid column numbers are 1 through the length of the specified line.

```
EInvalidLineOrPara = class(EEditorError)
```

The specified line or paragraph is not within the valid range. Valid line numbers are 1 through the number of lines currently in the editor’s text stream. Valid paragraph numbers are 1 through the number of paragraphs currently in the editor’s text stream.

Entry field exceptions

Exception (VCL)

    EOvcException (OvcExcpt)

        EEntryFieldError (OvcExcpt)

            EInvalidDataType (OvcExcpt)

            EInvalidPictureMask (OvcExcpt)

            EInvalidRangeValue (OvcExcpt)

            EInvalidDateForMask (OvcExcpt)

`EEntryFieldError = class(EOvcException)`

Base exception class for the entry field components. See the OnError event in "TOvcBaseEntryField Class" on page 363for a list of the error conditions that cause this exception to be raised.

`EInvalidDataType = class(EEntryFieldError)`

The attempted operation is inappropriate for the current field type and option settings. For example, an attempt was made to use the AsBoolean property to obtain the value of an entry field that has a string data type.

`EInvalidPictureMask = class(EEntryFieldError)`

An attempt was made to assign an invalid character or string as the field's picture mask.

`EInvalidRangeValue = class(EEntryFieldError)`

Generated when you assign a range value that is outside the valid range.

`EInvalidDateForMask = class(EEntryFieldError)`

The entered date can not be resolved using the current epoch range. This error can only occur with date masks using a two-digit year mask.

Fixed font exceptions

Exception (VCL)

- EOvcException (OvcExcpt)
  - EFixedFontError (OvcExcpt)
    - EInvalidFixedFont (OvcExcpt)
    - EInvalidFontParam (OvcExcpt)

`EFixedFontError = class(EOvcException)`

Base exception class for the TOvcFixedFont class. The only error conditions that are raised using this base class are ones that are generated at design time.

`EInvalidFixedFont = class(EFixedFontError)`

Raised when you attempt to assign a font that is not fixed-pitch to an instance of a TOvcFixedFont. The font assigned to a TOvcFixedFont object must have a fixed pitch.

`EInvalidFontParam = class(EFixedFontError)`

Raised when you attempt to assign a class other than a TFont or a TOvcFixedFont using the TOvcFixedFont's Assign method.

MRU exceptions

Exception (VCL)

EOvcException (OvcExcpt)

EMenuMRUError (OvcExcpt)

```
EMenuMRUError = class(EOvcException);
```

This is a general exception class for the TOvcMenuMRU component. This exceptions is raised if you attempt to call the Add method when the Style property is msSplit, if you attempt to call AddSplit when the Style is msNormal, if you pass an invalid parameter to the AddSplit method, or if you attempt to call AddSplit when no menu item is assigned.

Notebook exceptions

Exception (VCL)

EOvcException (OvcExcpt)

ENotebookError (OvcExcpt)

EInvalidPageIndex (OvcExcpt)

EInvalidTabFont (OvcExcpt)

```
ENotebookError = class(EOvcException)
```

Base exception class for the TOvcNotebook component. The only error conditions that are raised using this base class are ones that are generated at design time.

```
EInvalidPageIndex = class(ENotebookError)
```

The index points to a notebook page that does not exist or it is disabled or hidden. The valid range for index values is 0 to the number of notebook pages - 1.

```
EInvalidTabFont = class(ENotebookError)
```

When right-side tabs are used, the notebook's font must be a TrueType font. That is because the font used must be rotated 90 degrees for right-side tabs, and that can only be done with TrueType fonts.

ReportView exceptions

Exception (VCL)

- EOvcException (OvcExcpt)
  - EReportViewError (OvcExcpt)
    - EUnknownView (OvcExcpt)
    - EItemNotFound (OvcExcpt)
    - EItemAlreadyAdded (OvcExcpt)
    - EUpdatePending (OvcExcpt)
    - EItemIsNotGroup (OvcExcpt)
    - ELineNoOutOfRange (OvcExcpt)
    - ENotMultiSelect (OvcExcpt)
    - EItemNotInIndex (OvcExcpt)
    - ENoActiveView (OvcExcpt)
    - EOnCompareNotAsgnd (OvcExcpt)
    - EGetAsFloatNotAsg (OvcExcpt)
    - EOnFilterNotAsgnd (OvcExcpt)

`EReportViewError = class(EOvcException)`

Base exception class for the TOvcNotebook component. The only error conditions that are raised using this base class are ones that are generated at design time.

`EUnknownView = class(EReportViewError)`

The specified view name is unknown or undefined.

`EItemNotFound = class(EReportViewError)`

This exception is raised when RemoveData or ChangeData are called with data which can not be found in the ReportView's data index.

`EItemAlreadyAdded = class(EReportViewError)`

The record specified in AddData is already present in the ReportView index.

`EUpdatePending = class(EReportViewError)`

Raised when an attempt is made to perform an operation which is not allowed while updates are pending. During a BeginUpdate/EndUpdate pair, only AddData, ChangeData and RemoveData are allowed operations.

`EItemIsNotGroup = class(EReportViewError)`

GroupRef property is accessed with a line number that does not specify a group.

`ELineNoOutOfRange = class(EReportViewError)`

Selected property is accessed using an invalid line number.

`ENotMultiSelect = class(EReportViewError)`

Selected operation is invalid while MultiSelect is False.

`ENoActiveView = class(EReportViewError)`

Accessed property or method requires an active view and no view is currently active.

`EOnCompareNotAsgnd = class(EReportViewError)`

Raised by the ReportView component when no OnCompare event handler has been assigned.

`EGetAsFloatNotAsg = class(EReportViewError)`

The ReportView component has activated a view with totals and no OnGetFieldAsFloat event handler has been assigned.

`EOnFilterNotAsgnd = class(EReportViewError)`

The ReportView component has activated a filtered view and no OnFilter event handler has been assigned.

Rotated Label exceptions

```
Exception (VCL)
   EOvcException (OvcExcpt)
        ERotatedLabelError (OvcExcpt)
             EInvalidLabelFont (OvcExcpt)
```

`ERotatedLabelError = class(EOvcException)`

Base exception class for the TOvcRotatedLabel component. The only error conditions that are raised using this base class are ones that are generated at design time.

`EInvalidLabelFont = class(ERotatedLabelError)`

An attempt was made to assign a font that is not a TrueType font when the rotated label's Angle property is not zero. Rotated label components with Angle property values other than zero require a TrueType font.

Sparse Array exceptions

Exception (VCL)
- EOvcException (OvcExcpt)
  - ESparseArrayError (OvcExcpt)
    - ESAEAtMaxSize (OvcExcpt)
    - ESAEOutOfBounds (OvcExcpt)

```
ESparseArrayError = class(EOvcException)
```

Base exception class for the TOvcSparseArray class. The only error conditions that are raised using this base class are ones that are generated at design time.

```
ESAEAtMaxSize = class(ESparseArrayError)
```

The sparse array is at its maximum size and no more elements can be added. The maximum size for a sparse array is 320,000 elements. This is defined as the MaxSparseArrayItems constant in the OvcSpAry unit.

```
ESAEOutOfBounds = class(ESparseArrayError)
```

The sparse array index is not in the valid range. The valid range for a sparse array index is 0 through the number of elements in the sparse array minus one.

Table exceptions

Exception (VCL)
- EOvcException (OvcExcpt)
  - ETableError (OvcExcpt)

```
ETableError = class(EOvcException)
```

Base exception class for the TOvcTable component.

Timer Pool exceptions

Exception (VCL)

    EOvcException (OvcExcpt)

        ETimerPoolError (OvcExcpt)

            EInvalidTriggerHandle (OvcExcpt)

```
ETimerPoolError = class(EOvcException)
```

Base exception class for the TOvcTimerPool component. The only error conditions that are raised using this base class are ones that are generated at design time.

```
EInvalidTriggerHandle = class(ETimerPoolError)
```

An attempt was made to reference a trigger handle that is invalid.

Viewer exceptions

Exception (VCL)

    EOvcException (OvcExcpt)

        EViewerError (OvcExcpt)

            ERegionTooLarge (OvcExcpt)

```
EViewerError = class(EOvcException)
```

Base exception class for the TOvcFileViewer and TOvcTextFileViewer components. The only error conditions that are raised using this base class are ones that are generated at design time.

```
ERegionTooLarge = class(EViewerError)
```

Text to be copied to the Windows clipboard is larger than 64KB.

Virtual Listbox exceptions

Exception (VCL)

  EOvcException (OvcExcpt)

    EVirtualListBoxError (OvcExcpt)

      EOnSelectNotAssigned (OvcExcpt)

      EOnIsSelectedNotAssigned (OvcExcpt)

```
EVirtualListboxError = class(EOvcException)
```

Base exception class for the TOvcVirtualListbox component. The only error conditions that are raised using this base class are ones that are generated at design time.

```
EOnSelectNotAssigned = class(EVirtualListboxError)
```

Multiple selection is enabled, but the OnSelect event is not assigned. If the MultiSelect property is True, a method must be assigned to the OnSelect event to store the selection status of all elements in the list. This exception indicates a programming error and should not be caught in a try block since doing so would serve no purpose other than to hide the error. When this exception is raised, a message is displayed indicating that a method is not assigned to the OnSelect event.

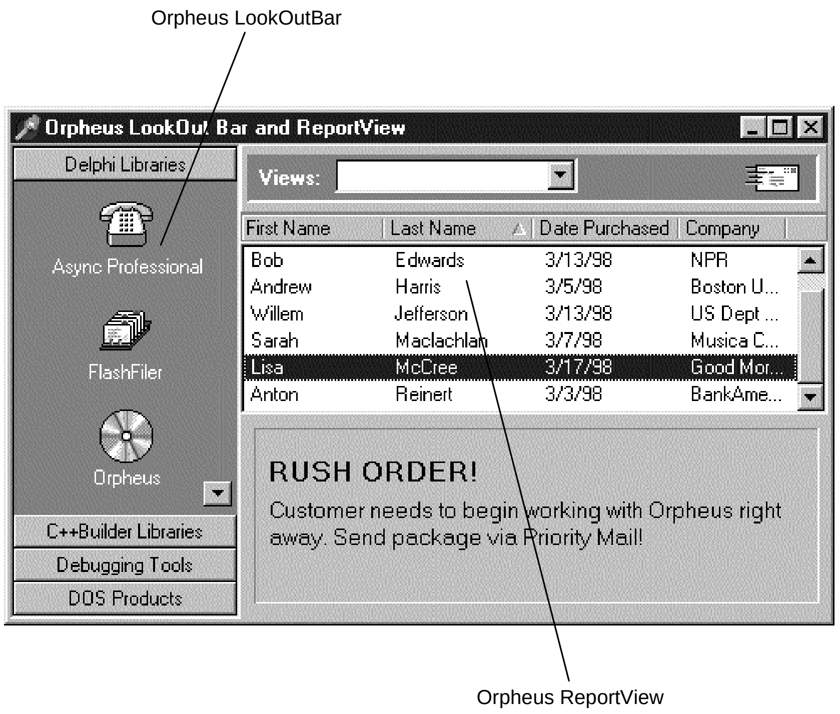
```
EOnIsSelectedNotAssigned = class(EVirtualListboxError)
```

Multiple selection is enabled, but the OnIsSelected event is not assigned. If the MultiSelect property is True, a method must be assigned to the OnIsSelected event to provide the selection status of all elements in the list. This exception indicates a programming error and should not be caught in a try block since doing so would serve no purpose other than to hide the error. When this exception is raised, a message is displayed indicating that a method is not assigned to the OnIsSelected event.

Chapter 3: LookOutBar Component

The LookOutBar component has been updated. The modifications caused enough interface changes that the old version (TOvcLookOutBar) was deprecated and a completely new version was created with a new name. For documentation on the old TOvcLookOutBar, see the Deprecated Components chapter on page 1159.

The TO32LookOutBar component emulates the control you see on the left side of Microsoft Outlook ™. The contents of the component are arranged by folders and items. Each TO32LookOutBar contains one or more folders. Within each folder are one or more folder items. By default, folders are depicted as buttons on the TO32LookOutBar. Clicking on a folder's button or tab makes that folder the active folder and displays its contents (the items in the folder). Clicking on a folder item generates an event that can be used to perform some action in the program. A typical use for an TO32LookOutBar is to place the bar in the left side of a form with different item views displayed to the right of the bar. As shown below, using an TO32LookOutBar and TOvcReportView together is a powerful combination.



First Name	Last Name	Date Purchased	Company
Bob	Edwards	3/13/98	NPR
Andrew	Harris	3/5/98	Boston U...
Willem	Jefferson	3/13/98	US Dept ...
Sarah	Maclachlan	3/7/98	Musica C...
Lisa	McCree	3/17/98	Good Mor...
Anton	Reinert	3/3/98	BankAme...

RUSH ORDER!

Customer needs to begin working with Orpheus right away. Send package via Priority Mail!

Figure 3.1: LookOutBar and ReportView

TO32 LookOutBar folders contain other components. Set any folder’s FolderType property to ftContainer and it becomes the equivalent of a ScrollBox that accepts other components dropped on it at design time.

Only one folder’s contents can be visible at a given time. If a ftDefault type folder contains more items than can be displayed in its client area, spinner buttons will be displayed. If a ftContainer type folder contains more items than can be displayed then it will create scroll bars that can be used to scroll the client area. Clicking the spinner buttons or manipulating the scroll bars allows you scroll the active folder’s client area to gain access to the additional items.

In a ftDefault type folder, folder items can be displayed in one of two ways. When large icons are used (images greater than 16 by 16 pixels), the image is centered in the client area of the TO32LookOutBar with the item’s text below the image. When small icons are used, the icon image appears near the left edge of the TO32LookOutBar with text to the right of the image. Icon is a generic term; the image list can contain either icons or bitmaps. Use a VCL TImageList component to store the item images.

The TO32LookOutBar supports in-place, run time editing for folders and ftDefault type folder items. Calling the RenameFolder method invokes in-place editing of folder captions. In-place editing of folder item captions is invoked by calling RenameItem. Users can then directly type a new caption for the folder or item. Pressing the Enter key on the keyboard accepts the new caption and the updates TO32LookOutBar. Pressing the ESC key abandons edits and the previous caption is restored. Persistence for folder and folder item captions can be implemented by using the Orpheus component state components. See page 257 for more information.

Note: RenameItem will not work for folder items if the folder’s FolderType is ftContainer.

There are four primary folder and item events supplied by the TO32LookOutBar:

- The OnFolderClick event notifies you when a folder changes.
- The OnItemClick event notifies you when a folder item is clicked. You can use this event to change the view associated with the item clicked.
- The OnFolderChange event notifies you when a folder is about to change.
- The OnFolderChanged event notifies you after a folder has changed.

Folders can be added at design time using the Folder Editor. To invoke the Folder Editor double-click the TO32LookOutBar on your form. Alternatively you can double-click the value column next to the FolderCollection property in the Object Inspector. The Folder Editor contains buttons to add, remove, or reorder folders. Click on a folder in the Folder Editor to display that folder's properties in the Object Inspector. Folder items are added at design time via the Folder Item Editor. This property editor is displayed when you double-click next to a folder's ItemCollection property in the Object Inspector. As with the Folder Editor, clicking a folder item in the Folder Items Editor displays that item's properties in the Object Inspector.

TO32LookOutBar Component

Hierarchy

TCustomControl (VCL)

❶ TO32CustomControl (OvcBase). 16

❷ TO32CustomControlEx (OvcBase) 19

TO32LookOutBar (OvcLkOut)

Properties

- ❶ About
- ActiveFolder
- ActiveItem
- AllowRearrange
- ❶ AttachedLabel
- BackgroundColor
- BackgroundImage
- BackgroundMethod
- ButtonHeight
- ❷ Controller
- DrawingStyle
- FolderCollection
- FolderCount
- Folders
- Images
- ItemFont
- ❶ LabelInfo
- PreviousFolder
- PreviousItem
- PlaySounds
- ScrollDelta
- SoundAlias

Methods

- BeginUpdate
- EndUpdate
- GetFolderAt
- GetItemAt
- InsertFolder
- InsertItem
- InvalidateItem
- RemoveFolder
- RemoveItem
- RenameFolder
- RenameItem

Events

- ❶ AfterEnter
- ❶ AfterExit
- OnDragDrop
- OnDragOver
- OnFolderChange
- OnFolderChanged
- OnFolderClick
- OnItemClick
- OnMouseOverItem
- ❶ OnMouseWheel

Reference Section

ActiveFolder — property

```
property ActiveFolder : Integer
```

The index number of the active folder.

Read ActiveFolder to determine the index number of the active folder. Set ActiveFolder to programmatically set the active folder. Folders are 0-based with the first folder at index 0, the second at index 1, and so on.

See also: FolderCount, InsertFolder, RenameFolder

ActiveItem — run-time, read-only property

```
property ActiveItem : Integer
```

The index number of the active folder's active item.

Active Item is 0-based with the first item in each folder at index 0.

Note: Components contained in folders of type ftContainer are not numbered and are not accessible via the ActiveItem property.

See also: ActiveFolder, TO32LookoutFolder.Items, TO32LookoutFolder.FolderType

AllowRearrange — property

```
property AllowRearrange : Boolean
```

Default: True

Determines whether or not the items within a folder can be rearranged by dragging.

When AllowRearrange is True, items can be moved via drag and drop. When AllowRearrange is False, items cannot be rearranged. Folder items can be dragged within a folder or between folders. When dragging between folders, the target folder will be expanded when the mouse cursor is dragged over a folder button.

Allow Rearrange only applies to default folders and their items. For instance, you cannot drag a default item to a container folder.

See also: Items, TO32LookoutFolder.FolderType

BackgroundColor property

```
property BackgroundColor : TColor
```

Default: clInactiveCaption

↳ The background color of the Lookout Bar.

BackgroundColor is applied to the background when the BackgroundMethod property is set to bmNone.

See also: BackgroundImage, BackgroundMethod

BackgroundImage property

```
property BackgroundImage : TBitmap
```

↳ The image that is displayed in the Lookout Bar's background.

By default BackgroundImage is nil and the color of the background area is determined by the BackgroundColor property. Set BackgroundImage to display a bitmap on the background of the component instead. The background fill method (none, normal, or stretch) is determined by the BackgroundMethod property.

See also: BackgroundColor, BackgroundMethod

BackgroundMethod property

```
property BackgroundMethod : TO32BackgroundMethod
```

Default: bmNormal

The painting method to paint the background.

The possible values for BackgroundMethod are:

Value	**Description**
bmNone	No background bitmap is applied. The background is filled using the color specified in BackgroundColor.
bmNormal	The bitmap specified in BackgroundImage is used to paint the background. The image is not sized to fit the component.
bmStretch	The background is filled with the bitmap specified in the BackgroundImage property. The image is stretched to fit the size of the component.

Setting BackgroundMethod to bmNormal or bmStretch has no effect if BackgroundImage is not assigned

See also: BackgroundColor, BackgroundImage

BeginUpdate method

```
procedure BeginUpdate;
```

Temporarily disables repainting of the control.

Call BeginUpdate to disable repainting of the component’s items if you are programmatically adding many items to a folder. Call EndUpdate after adding items to force the component to be repainted.

Warning: Calls to BeginUpdate and EndUpdate are cumulative so each call to BeginUpdate must have a corresponding EndUpdate call.

The following example disables painting of the component, adds items to the first folder in the folder list, and then re-enables painting:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I  : Integer;
begin
  O32LookOutBar1.BeginUpdate;
  for I := 0 to 19 do
    O32LookOutBar1.InsertItem(
      'Item ' + IntToStr(I + 1), 0, I, I);
  O32LookOutBar1.EndUpdate;
end;
```

See also: EndUpdate

ButtonHeight property

```
property ButtonHeight : Integer
```

Default: 20

The height of the Lookout Bar's folder button or tab (Depending on the value of DrawingStyle).

See also: DrawingStyle

DrawingStyle property

```
property DrawingStyle : TO32FolderDrawingStyle
```

```
TO32FolderDrawingStyle = (
  dsCooltab, dsDefButton, dsEtchedButton, dsStandardTab);
```

The style used to draw the Lookout Bar's folder indicators.

See the following table for information on how the LookOutBar is drawn for each selection:

DrawingStyle	**Result**
dsDefButton	The border and folder buttons are drawn using the Microsoft Outlook97™ style.
dsEtchedButton	The component is drawn using the Outlook98™, etched style.
dsCoolTab	The component is drawn using the Microsoft Outlook97™ style except the folder buttons are drawn as rounded, 3D tabs.
dsStandardTab	The component is drawn using the Microsoft Outlook97™ style except the folder buttons are drawn as standard, squared off tabs.

EndUpdate method

```
procedure EndUpdate;
```

Repaints the component after adding items with painting disabled.

Call EndUpdate to repaint the component after adding items when repainting has been disabled (by calling BeginUpdate).

Warning: Calls to BeginUpdate and EndUpdate are cumulative so each call to BeginUpdate must have a corresponding EndUpdate call.

See also: BeginUpdate

FolderCollection property

```
property FolderCollection : TO32Collection
```

The list of folders for the component.

A TO32LookOutBar contains one or more folders. FolderCollection contains the list of folders. Each folder's characteristics are set at design time via the FolderCollection property editor or at run time through code. To enumerate a TO32LookOutBar's folders, use the Folders property. To add folders at run time, call the InsertFolder method. To remove folders at run time, call the RemoveFolder method.

See also: FolderCount, Folders, InsertFolder, RemoveFolder

FolderCount run-time, read-only property

```
property FolderCount : Integer
```

↳ The number of folders in the TO32LookOutBar.

See also: Folders, FolderCollection

Folders property

```
property Folders[Index : Integer] : TO32LookoutFolder
```

↳ An indexed property containing the folders in the TO32LookOutBar.

Use Folders at run time to access a particular folder by its folder index or to enumerate the folders. The following example changes the caption of the second folder in the list:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  O32LookOutBar1.Folders[1].Caption := 'Inbox';
end;
```

See also: FolderCollection, FolderCount

GetFolderAt method

```
function GetFolderAt(X, Y : Integer) : Integer;
```

↳ Returns the folder index of the folder at the given coordinates.

See also: Folders, GetItemAt

GetItemAt method

```
function GetItemAt(X, Y : Integer) : Integer;
```

↳ Returns the item index of the default folder item at the given coordinates.

If the active folder is of ftContainer type then GetItemAt returns -1.

See also: GetFolderAt, TO32LookoutFolder.Items

Images property

```
property Images : TImageList
```

↳ The image list that contains the images used to display folder items.

Set Images to the name of the image list that contains the folder item images. Set each folder item's IconIndex property to the index corresponding to the image to display for that item. The folder items' display characteristics are determined by the folder's IconSize property. For large icons, 32 by 32 pixels is the recommended size. For small icons, 16 by 16 pixel images are recommended.

Use a VCL TImageList component to store the item images.

See also: TO32LookoutItem.IconIndex, TO32LookoutFolder.IconSize

InsertFolder method

```
procedure InsertFolder(
  const ACaption : string; AFolderIndex : Integer);
```

↳ Adds a folder to the OvcLookOutBar.

ACaption is the caption for the new Folder. AFolderIndex is the position in the list where the new item will be inserted. The following example adds a new folder at the bottom of a TO32LookOutBar:

```
  procedure TForm1.Button1Click(Sender : TObject);
  begin
    O32LookOutBar1.InsertFolder(
      'Outbox', OvcLookOutBar1.FolderCount);
  end;
```

See also: InsertItem, FolderCollection, Folders, RemoveFolder

InsertItem method

```
procedure InsertItem(const ACaption : string;
  AFolderIndex, AItemIndex, AIconIndex : Integer);
```

↳ Adds a folder item into a folder of type ftDefault.

Call InsertItem to programmatically add an item to a folder at run time. ACaption is the caption for the new item. AFolderIndex is the index number of the parent folder for the new item. AItemIndex is the position within the folder where the new item will be added. AIconIndex is the index of the image within the image list that will be used to display the new item.

The following example inserts a new item at the top of the first folder in a TO32LookOutBar (folder index 0, item index 0). The image index is set to the sixth image in the image list (image index 5).

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  O32LookOutBar1.InsertItem('New Item', 0, 0, 5);
end;
```

Note: Calls to InsertItem are ignored if AFolderIndex refers to a folder of type ftContainer.

See also: InsertFolder, RemoveItem

InvalidateItem method

```
procedure InvalidateItem(FolderIndex, ItemIndex : Integer);
```

Forces a redraw of the specified item.

ItemFont property

```
property ItemFont : TFont
```

The font that is used to draw the text associated with a folder item. If ItemFont is unassigned, the OvcLookOutBar's font will be used.

OnDragDrop event

```
property OnDragDrop : TO32LOBDragDropEvent
```

```
TO32LOBDragDropEvent = procedure(
  Sender, Source: TObject; X, Y: Integer;
  FolderIndex, ItemIndex : Integer)of object;
```

Defines an event handler that is generated when a drop occurs from another component.

Source is the component that initiated the drag event. X is the cursor x position and Y is the cursor y position. State is the drag state (see the VCL help for an explanation of TDragState). FolderIndex is the index of the folder on which the drop occurred. ItemIndex is the item index position of the drop within the folder. The value of ItemIndex varies depending on whether you are allowing a drop on folders, on items, or both. If you are allowing drop on folders, ItemIndex is the index where a new item will be inserted (when you call InsertItem, for example). If you are allowing drop on items within a folder, ItemIndex is the index number of the item that was under the cursor when the drop occurred.

See also: OnDragOver

OnDragOver event

```
property OnDragOver : TO32LOBDragOverEvent
```

```
TO32LOBDragOverEvent = procedure(
  Sender, Source : TObject; X, Y : Integer; State : TDragState;
  var AcceptFolder, AcceptItem : Boolean) of object;
```

Defines an event handler that is generated when an item is being ragged from another component.

Source is the component that initiated the drag event. X is the cursor x position and Y is the cursor y position. State is the drag state (see the VCL help for an explanation of TDragState). AcceptFolder controls whether or not a folder can accept a drop. When AcceptFolder is True a line is drawn on the folder at the cursor position. An item can be dropped on the folder at the indicated position. AcceptItem controls whether or not a folder item can accept a drop. When AcceptItem is True, the folder item under the mouse cursor is highlighted to indicate that a drop on the folder item is valid. If neither AcceptFolder or AcceptItem are True then the no drop cursor is displayed and the OnDragDrop event is not generated. AcceptFolder does not have to be True to accept drops on folder items.

See also: OnDragDrop

OnFolderChange event

```
property OnFolderChange : TFolderChangeEvent
```

```
TFolderChangeEvent = procedure(
  Sender : TObject; Index : Integer; var AllowChange : Boolean;
  Dragging : Boolean) of object;
```

Defines an event handler that is generated when the ctive folder is about to change.

Index is the index number of the folder that is about to become active. AllowChange determines whether or not the folder can change. By default AllowChange is True. Setting AllowChange to False within your event handler prevents the folder change from taking place. Dragging is a flag that indicates whether a folder item is being dragged. You may want to allow a particular folder to change if the user is attempting to drag an item from one folder to another, but disallow the folder change during normal operation.

See also: OnFolderChanged

OnFolderChanged event

```
property OnFolderChanged : TFolderChangedEvent
```

```
TFolderChangedEvent = procedure(
  Sender : TObject; Index : Integer) of object;
```

Defines an event handler that is generated when the active folder changes.

Index is the index number of the new active folder. OnFolderChange is a notification event. As such, it is generated after the active folder changes. To detect a folder state change before the folder changes, use the OnFolderChange event.

See also: OnFolderChange

OnFolderClick event

```
property OnFolderClick : TFolderClickEvent
```

```
TFolderClickEvent = procedure(
  Sender : TObject; Button : TMouseButton;
  Shift : TShiftState; Index : Integer) of object;
```

Defines an event handler that is generated when a folder button is clicked.

Button is the mouse button that was use to click the folder. Shift is the keyboard state when the folder was clicked. Index is the folder index of the folder that was clicked. Provide an event handler for OnFolderClick to determine when a folder button is clicked.

See also: OnItemClick

OnItemClick event

```
property OnItemClick : TItemClickEvent
```

```
TItemClickEvent = procedure(
  Sender : TObject; Button : TMouseButton;
  Shift : TShiftState; Index : Integer) of object;
```

Defines an event handler that is generated when a folder item is clicked.

Button is the mouse button that was use to click the folder. Shift is the keyboard state when the item was clicked. Index is the item index of the item that was clicked. Provide an event handler for OnFolderClick to determine when a folder button is clicked. Read the ActiveFolder property to determine the folder that the item clicked belongs to.

The following example changes pages in a notebook based on the item clicked. This example assumes the OvcLookOutBar contains a single folder.

```
  procedure TForm1.O32LookOutBar1ItemClick(
    Sender : TObject; Button : TMouseButton;
    Shift : TShiftState; Index : Integer);
  begin
    NoteBook1.PageIndex := Index;
  end;
```

See also: OnFolderClick

OnMouseOverItem event

```
property OnMouseOverItem : TO32MouseOverItemEvent
```

```
TO32MouseOverItemEvent = procedure(
  Sender : TObject; Item : TO32LookoutItem) of object;
```

Defines an event handler that is fired when the mouse cursor moves over a TO32LookOutBar item.

Sender is the object that generated the event. Item is a pointer to the item that the cursor is currently over. The following example shows how to display each item's description in a status bar:

```
  procedure TForm1.O32LookOutBar1MouseOverItem(
    Sender : TObject; Item : TO32LookoutItem);
  begin
    StatusBar1.SimpleText := Item.Description;
  end;
```

PlaySounds property

```
property PlaySounds : Boolean
```

Default: False

Determines whether or not a sound is played when the active folder changes.

Set PlaySounds to True to enable playing of a sound when the active folder changes. The sound played is determined by the SoundAlias property.

See also: SoundAlias

PreviousFolder — run-time, read-only property

```
property PreviousFolder : Integer
```

↳ The index of the folder that was previously selected.

Read PreviousFolder to determine the index of the folder that was previously selected. If no previous folder was selected PreviousFolder contains –1.

See also: ActiveFolder, Folders, FolderCollection, OnFolderClick, PreviousItem

PreviousItem — run-time, read-only property

```
property PreviousItem : Integer
```

↳ The index of the folder item that was previously clicked.

Read PreviousItem to determine the index number of the item that was previously clicked. If no item was previously clicked, PreviousItem contains –1. To determine the folder containing the previously clicked item, read the PreviousFolder or ActiveFolder properties.

See also: OnItemClicked, PreviousFolder, TO32LookoutFolder.Items

RemoveFolder — method

```
procedure RemoveFolder(AFolderIndex : Integer);
```

RemoveFolder removes the folder specified by AFolderIndex. All of the folder's contained items or components are destroyed.

RemoveItem — method

```
procedure RemoveItem(AFolderIndex, AItemIndex : Integer);
```

↳ Removes the specified item from the specified folder.

If AFolderIndex refers to a folder of type ftContainer then all calls to RemoveItem are ignored.

RenameFolder method

```
procedure RenameFolder(AFolderIndex : Integer);
```

Enables in-place editing of a folder's caption.

Call RenameFolder to invoke the OvcLookOutBar's in-place editor. Users can then type a new caption for the folder. Pressing <Esc> abandons the operation and returns the folder caption to its original state. Pressing the <Enter> changes the folder's caption to the new text. The Orpheus state components can be used to make folder caption changes persistent.

See also: InsertFolder, RemoveFolder, RenameItem

RenameItem method

```
procedure RenameItem(AFolderIndex, AItemIndex : Integer);
```

Enables in-place editing of an item's caption.

Call RenameItem to invoke the OvcLookOutBar's in-place editor. Users can then type a new name for the item. Pressing the <Esc> abandons the operation and returns the item caption to its original state. Pressing the <Enter> changes the item's caption to the new text. The Orpheus state components can be used to make folder item caption changes persistent.

Note: If AFolderIndex refers to a folder of type ftContainer then all calls to RenameItem are ignored.

See also: InsertItem, RemoveItem, RenameFolder

ScrollDelta property

```
property ScrollDelta : Integer
```

Default: 2

Determines the scrolling granularity when a folder changes.

When a folder button is clicked, the new folder scrolls into view. A lower ScrollDelta results in finer (and slower) scrolling. Set ScrollDelta to a higher number to increase the scroll speed.

```
property SoundAlias : string
```

The sound that will be played when the active folder changes.

SoundAlias can be either a filename or the name of a system sound. System sounds include Minimize, Maximize, MailBeep, and so on. System sound names are listed inthe registry under the HKEY_CURRENT_USER\AppEvents\Schemes registry key. Using system sounds is advantageous in that it allows you to incorporate the user's current sound scheme. You may, for example, want to associate a folder change with Windows' minimize sound. If the specified sound cannot be found, the default Windows sound is played (usually the "ding" sound). SoundAlias is ignored if the PlaySounds property is set to False.

See also: PlaySounds

TO32LookoutFolder Class

Hierarchy

TCollectionItem (VCL)

❶ TO32CollectionItem (OvcBase) 48

TO32LookoutFolder (O32Lkout)

Properties

❶ About	❶ DisplayText	ItemCollection
Caption	IconSize	Items
Container	ItemCount	❶ Name

Caption property

```
property Caption : string
```

↳ The text displayed on a folder’s button.

A folder’s caption can be changed at run time by assigning a new value to Caption. Additionally, you can call TO32LookOutBar.RenameFolder to invoke the O32LookOutBar’s in-place editor.

See also: TO32LookOutBar.RenameFolder

Container property

```
property Container : TO32FolderContainer
```

```
TO32FolderContainer = class(TScrollingWinControl)
protected{Private}
  FFolder: TO32LookoutFolder;
  FContainedList: TStringList;
  FBorderStyle: TBorderStyle;
  FCanvas: TControlCanvas;
  procedure WMNCHitTest(var Message: TMessage);
    message WM_NCHITTEST;
  procedure SetFolder(Value: TO32LookoutFolder);
public
  constructor Create(AOwner: TComponent); override;
  destructor Destroy; override;
  property Canvas: TControlCanvas
    read FCanvas;
  property Folder: TO32LookoutFolder
    read FFolder write SetFolder;
  property ContainedList: TStringList
    read FContainedList;
end;
```

↳ Provides access to the folder’s component container. Container is only available if the folder’s type is ftContainer. Otherwise, it is nil.

IconSize — property

```
property IconSize : TO32IconSize
```

```
TO32IconSize = (isLarge, isSmall);
```

Default: isLarge

Determines the display characteristics of folder items.

IconSize can be either isLarge or isSmall. When IconSize is set to isLarge, images are displayed centered on the OvcLookOutBar with the text below the image. When IconSize is set to isSmall the image is drawn on the left side of the OvcLookOutBar with the text to the right of the image. For large icons an image size of 32 by 32 pixels is recommended. For small icons a 16 by 16 pixel image is recommended. Using images larger than 16 by 16 with the isSmall setting may result in the item's text overlapping the image.

See also: TO32LookOutBar.Images

ItemCount — run-time, read-only property

```
property ItemCount : Integer
```

The number of items in the folder.

ItemCount returns 0 if the folder is of type ftContainer.

See also: Items, ItemCollection

ItemCollection — property

```
property ItemCollection : TO32Collection
```

The list of items in a folder.

A TO32LookOutBar contains one or more folders with one or more items in each folder. ItemCollection contains the list of items for a folder. Each item's characteristics are set at design time via the ItemCollection property editor or by run time through code. To enumerate a folder's items, use the Items property. To add items at run time call the OvcLookOutBar's InsertItem method. To remove items at run time, call the RemoveItem method.

See also: Items, ItemCount, TO32LookOutBar.InsertItem, TO32LookOutBar.RemoveItem

Items run-time, read-only property

```
property Items[Index : Integer] : TO32LookoutItem
```

is an indexed property that allows access to the items in a folder.

Use Items to enumerate the items in a folder, or to access a particular item by index number. The following example changes the caption of the second item in the first folder of an OvcLookOutBar:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  O32LookOutBar1.Folders[0].Items[1].Caption := 'Outbox';
end;
```

See also: ItemCount, ItemCollection

TO32LookoutBtnItem Class

The TO32ButtonItem class models the behavior of the default LookOutBar items. These are the items contained in folders of type ftDefault. These items cannot be used in folders of type ftContainer.

Hierarchy

TCollectionItem (VCL)

    ❶ TO32CollectionItem (OvcBase) 48

        TO32LookoutBtnItem (O32LkOut)

Properties

❶	About	❶	DisplayText		LabelRect
	Caption		IconIndex	❶	Name
	Description		IconRect		

Reference Section

Caption property

```
property Caption : string
```

↳ Caption is the display text for a folder item.

An item's caption can be changed at run time by assigning a new value to Caption. Additionally, you can call TO32LookOutBar.RenameItem to invoke the OvcLookOutBar's in-place editor.

See also: TO32LookOutBar.RenameItem

Description property

```
property Description : string
```

↳ Description is used to specify a text description of a TO32LookOutBar item.

Description can be used to store any additional text associated with a TO32LookOutBar item. For example, you can use Description to display hint text for a particular item in your application's status bar. See TO32LookOutBar.OnMouseOverItem for an example.

See also: TO32LookOutBar.OnMouseOverItem

IconIndex property

```
property IconIndex : Integer
```

↳ IconIndex is the index in the image list used to draw a folder item.

Set each folder item's IconIndex property to the index corresponding to the image to display for that item. Images are contained in the OvcLookOutBar's Images property.

See also: TO32LookOutBar.Images

IconRect run-time, read-only property

```
property IconRect : TRect
```

↳ IconRect contains the size and location of a folder item's display icon.

See also: LabelRect

LabelRect run-time, read-only property

```
property LabelRect : TRect
```

LabelRect contains the size and location of a folder’s label.

See also: IconRect

Chapter 4: ReportView Components

A ReportView is a special kind of list organized in columns and rows with built-in capabilities for sorting data in columns by clicking the column headers, grouping data on arbitrary columns, and computing aggregates (totals, counts, and so on) for each column both globally and at the group level.

Orpheus contains three ReportView components that descend from a common class, TOvcCustomReportView:

- TOvcReportView is an event driven ReportView similar to a virtual list box. The form that contains the ReportView manages the data and supplies that data when the appropriate event fires.
- TOvcDataReportView implements an Items property, and thus manages the data of the ReportView internally.
- TOvcDbReportView extracts the data to be displayed from a standard data source.

A combo box designed specifically for Orpheus' ReportView components, TOvcViewComboBox automatically displays a list of the defined views for a specified ReportView. This combo box reflects changes in the list of identified views available to the ReportView and allows a user to select a current view from the list of valid views.

Overview

The ReportView component creates different views of the same information. A view is some subset of all the available fields from the underlying data.

Before you can define views with TOvcReportView or TOvcDataReportView you must define the fields that are available. TOvcDbReportView defines available fields automatically by extracting the field definitions from the associated data source.

The Fields property, a collection of TOvcRvField objects, defines the fields of the data items. These components are stored in the form file as child components of the ReportView itself and do not have separate component icons.

Clicking the ellipses button to the right of the Fields property in the Object Inspector opens a standard Orpheus collection editor for the Fields collection of the ReportView. Add field definitions by clicking the <+> button or by pressing <Ins>.

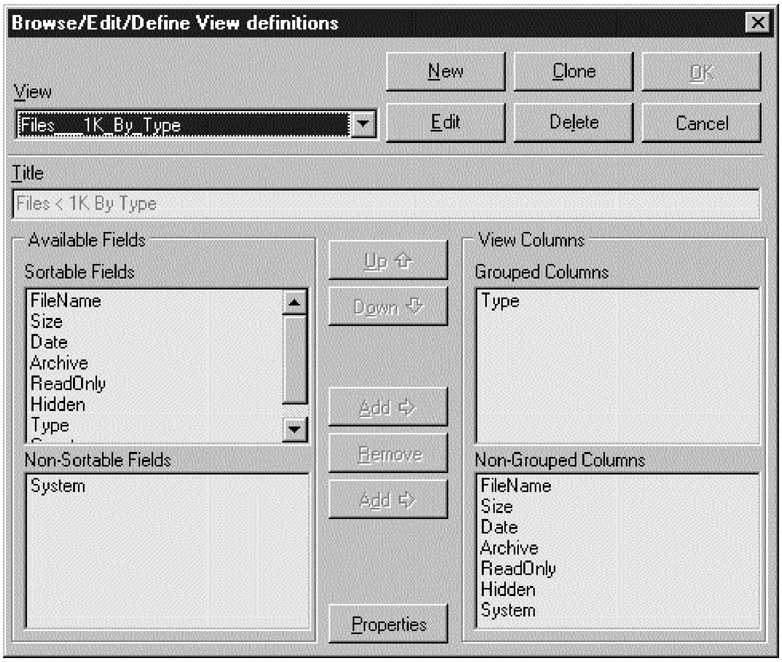
Once you have defined the collection of fields that define the data, which can be displayed in the report view, you can define multiple view layouts to display data in the ReportView. Views are stored in the Views property, which is a collection similar to the Fields collection just described, except that the component type for the items in this collection is TOvcRvView.

The view components, like the field components, are stored within the ReportView definition in the form file. Even though the views are components, they do not appear as separate icons on the form. This is similar in concept to the field components for a database table component.

Each of the view components contains a number of properties for the view along with a list of view fields that define the horizontal layout of the view.

The view fields are references to fields in the global Fields property of the ReportView with extra properties that are local to the current view. View fields are of the type TOvcRvViewField, which, again, are components stored in a collection property: ViewFields.

View definitions can be conveniently defined and edited at design time with the component editor for the ReportView component as shown in Figure 4.1. The view editor is activated from the context menu for the ReportView component.



View: Files___1K_By_Type

Title: Files < 1K By Type

Sortable Fields: FileName, Size, Date, Archive, ReadOnly, Hidden, Type

Non-Sortable Fields: System

Grouped Columns: Type

Non-Grouped Columns: FileName, Size, Date, Archive, ReadOnly, Hidden, System

Figure 4.1: View editor

There is also a run-time view layout editor, which lets the user edit and define new view layouts. In addition, there is an editor, which allows the end user to define entirely new calculated fields. These fields may then be used as normal columns in user-defined views.

TOvcRvField Class

The TOvcRvField class is a field definition in a ReportView. It never exists as a stand-alone component on the form, but is always a member of the ReportView's Fields collection.

Hierarchy

TComponent (VCL)

- ❶ TOvcComponent (OvcBase) 21
  - ❷ TOvcCollectible (OvcBase) 41
    - TOvcAbstractRvField (OvcRvIdx)
      - TOvcRvField (OvcRptVw)

Properties

❶	About		DefaultOwnerDraw	❷	Index
	Alignment		DefaultWidth		Hint
	CanSort		DefaultPrintWidth		Name
	Caption	❷	DisplayText		NoDesign
❷	Collection		Expression		OwnerReport
	DataType		ImageIndex		

Reference Section

Alignment property

```
property Alignment : TAlignment
```

Default: taLeft

↳ Specifies whether the field should be displayed left justified, centered, or right justified.

CanSort property

```
property CanSort : Boolean
```

Default: True

↳ Determines whether the user can click the column header for this field to sort the column, and also determines whether a view can be grouped on this field.

Caption property

```
property Caption : string
```

↳ Determines the text displayed in the header for the column and the group header for grouped columns.

DataType property

```
property DataType : TOvcDRDataType
```

```
TOvcDRDataType = (dtString, dtFloat, dtInteger, dtDateTime,
 dtBoolean, dtDWord, dtCustom);
```

Default: dtString

↳ Defines the type of data represented by the field.

For TOvcReportView (the event driven report view type), the data type property is used only for validating expressions (i.e., filter expressions or calculated fields).

For TOvcDataReportView (the report view which maintains its data internally), the data type defines the actual storage layout for the data in memory as well as the type of data used for expressions.

For TOvcDbReportView (the database-aware report view), the data type is defined automatically based on the fields of the data source being used.

Field of data types, which do not map to simple SQL data types, are set to dtCustom. This excludes them from participating in expressions.

See also: TOvcRvField.Expression, TOvcRvView.Filter

DefaultOwnerDraw property

```
property DefaultOwnerDraw : Boolean
```

Default: False

Determines the default value for the OwnerDraw property of a view column when a new view is defined.

The value of the OwnerDraw flag can be changed individually at the view level.

See also: TOvcRvViewField.OwnerDraw

DefaultPrintWidth property

```
property DefaultPrintWidth : Integer
```

Default: 1440

Determines the default width used while printing views that use this field.

The print width can be changed individually at the view level.

See also: TOvcRvViewField.PrintWidth

DefaultWidth property

```
property DefaultWidth : Integer
```

Default: 50

Determines the default display width used in views that use this field.

The width can be changed individually at the view level.

See also: TOvcRvViewField.Width

Expression property

```
property Expression : string
```

Defines a calculated field.

When Expression is non-blank, the report view does not access the report view data directly to evaluate the field value but calls the evaluation method for expressions instead.

The Expression syntax is a subset of SQL.

Hint property

```
property Hint : string
```

Defines a pop-up hint string for the field.

When present, the hint string is displayed when the mouse pointer hovers over the relevant report view header section. The hint string is also displayed in the run-time view designer when the mouse cursor hovers over the screen component which represents the field.

ImageIndex property

```
property ImageIndex : Integer
```

Default: -1

Used as an index into the image list optionally associated with the report view through its HeaderImages property.

If the index is valid for the image list, the image is displayed in the report view header for the column to the left of the caption. The caption may be blank.

See also: TOvcCustomReportView.HeaderImages

Name property

```
property Name : string
```

The name of the field component.

The name is used by the view definition to reference the field.

See also: TOvcRvViewField.FieldName

NoDesign property

```
property NoDesign : Boolean
```

Default: False

↳ Determines whether the field shows up in the various designers at run time.

NoDesign is automatically set to True, for fields that have a type, which is not supported by normal SQL expressions.

OwnerReport run-time, read-only property

```
property OwnerReport : TOvcCustomReportView
```

↳ A reference to the ReportView component that owns this field.

TOvcRvView Class

The TOvcRVView class holds a view definition within the ReportView component. It contains global properties for the view and also contains a collection of view fields that make up the horizontal layout of the view.

Hierarchy

TComponent (VCL)

TOvcAbstractRvView (OvcRvIdx)

TOvcRVView (OvcRptVw)

Properties

❶	About		Filter	ShowGroupTotals
❷	Collection		FilterIndex	ShowFooter
	DefaultSortColumn		Hidden	ShowHeader
	DefaultSortDescending	❷	Index	Title
	Dirty	❷	Name	ViewField
❷	DisplayText		ShowGroupCounts	ViewFields

DefaultSortColumn property

```
property DefaultSortColumn : Integer
```

Default: 0

↳ Defines the view column used for sorting when the view is activated. DefaultSortColumn is zero based.

See also: DefaultSortDescending

DefaultSortDescending property

```
property DefaultSortDescending : Boolean
```

Default: False

↳ Determines whether the view should be sorted in ascending or descending order when first activated.

The column used for sorting is determined by the DefaultSortColumn property.

See also: DefaultSortColumn

Dirty run-time property

```
property Dirty: Boolean
```

↳ Indicates whether the view has been changed by the user (e.g., when a column is resized or moved).

Dirty is initially set to True for views that are created by the run-time view editor.

If the owning report view has an associated storage component in its CustomViewStore property, the report view will save a dirty view automatically when the report view is destroyed and load custom views when the report view is created.

See also: LoadFromStorage, SaveToStorage

Filter property

```
property Filter : string
```

Defines a Boolean expression which is evaluated for each row in the view. Only rows where the filter expression evaluates to True are included in the view. See “Report view expression syntax” on page 217 for more information.

See also: FilterIndex

FilterIndex property

```
property FilterIndex : Integer
```

Default: -1

Specifies an index value passed to the OnFilter event for the view, which allows the report views to implement custom filtering.

Though still fully functional, the use of FilterIndex and the OnFilter event has been deprecated in favor of filter expressions. A non-blank Filter expression takes precedence over the FilterIndex value.

See also: Filter, TOvcCustomReportView.OnFilter

Hidden property

```
property Hidden : Boolean
```

Default: False

Determines whether the view shows up in the view combo box.

This is convenient if the application uses a view internally to access data in a certain order, say, and makes specific assumptions about the structure of a view definition. Making the view hidden prevents the user from activating and thus modifying the view.

ShowFooter property

```
property ShowFooter : Boolean
```

Default: False

Determines whether the view should display a footer.

Footers are typically used to display totals.

ShowGroupCounts property

```
property ShowGroupCounts : Boolean
```

Default: False

↳ Determines whether the report view displays the number of items in each group following the group header for each group.

The property has no effect on non-grouped views.

ShowGroupTotals property

```
property ShowGroupTotals : Boolean
```

Default: False

↳ Determines whether totals should be displayed at the group level for the view fields that have their ComputeTotals property set to True.

This property has no effect unless the view has grouped columns.

See also: TOvcRvViewField.ComputeTotals, TOvcReportView.OnGetGroupString

ShowHeader property

```
property ShowHeader : Boolean
```

Default: True

↳ Determines whether the view should display the column header.

Title property

```
property Title : string
```

↳ The user-friendly name for the view.

Views have both a Name (inherited from TComponent) and a Title property. The name is used internally for streaming and for referencing the view. As opposed to the Name property, Title can contain spaces and special characters.

See also: TOvcCustomReportView.ActiveViewByTitle

ViewField — run-time, read-only property

```
property ViewField[Index : Integer] : TOvcRvViewField
```

↳ An indexed property that returns a reference to the relevant field in the view.

ViewFields — property

```
property ViewFields : TOvcRvViewFields
```

↳ A collection of TOvcRvViewField components that make up the view.

The properties of views can be edited at design time by selecting the relevant view component in the collection editor for the Views property of the ReportView. Views can also be created or edited using the custom property editor for the ReportView.

TOvcRvViewField Class

The TOvcRvViewField class represents a field definition within a view definition in a ReportView. This is also referred to as a column. Components of type TOvcRvViewField never exist as stand-alone components on the form: they are always members of the View definition's ViewFields collection.

Hierarchy

TComponent (VCL)

- ❶ TOvcComponent (OvcBase) 21
  - ❷ TOvcCollectible (OvcBase) 41
    - TOvcAbstractRvViewField (OvcRvIdx)
      - TOvcRVViewField (OvcRptVw)

Properties

❶	About		Field	OwnerReport
	Aggregate		FieldName	OwnerView
	AllowResize		GroupBy	PrintWidth
❷	Collection	❷	Index	ShowHint
	ComputeTotals	❷	Name	Visible
❷	DisplayText		OwnerDraw	Width

Reference Section

Aggregate property

```
property Aggregate : string
```

Defines an optional aggregate expression for the view column. The value of the expression is displayed on group lines and in the footer for the column.

Note that for the Aggregate expression to be evaluated, the ComputeTotals property must be set to True. Setting ComputeTotals to True without specifying an explicit Aggregate expression is equivalent to having an aggregate expression of the form as shown in the following:

SUM(Name)

Name is to be construed as the actual value of the Name property (e.g., if you had a view field which referred to the field Size in the report views Fields property, then the implicit aggregate expression when setting ComputeTotals to true would be "SUM(Size)").

The Aggregate expression does not limit you to simple aggregates like this, though. For example, you may specify (potentially meaningless) expressions like "AVG(FieldA + FieldB) – SUM(FieldC)". Here, the fields FieldA, FieldB, and FieldC are all assumed to be numeric.

Note that, as in full SQL, you may not refer directly to a field in aggregate expressions. Otherwise, all features of the expression syntax may be freely used in aggregate expressions. Field references, however, must be enclosed in one of the specific aggregate functions, SUM, MAX, MIN, or AVG, as outlined in the example in the previous paragraph. There is a fifth aggregate function, COUNT, which returns the number of rows in the group or view. SQL allows COUNT to be used either with an asterisk as the sole argument, or with a field name optionally prefixed with either DISTINCT or ALL. The Orpheus report view expression syntax only allows the asterisk form and disallows the DISTINCT and ALL modifiers. See "Report view expression syntax" on page 217 for more information.

AllowResize property

```
property AllowResize : Boolean
```

Default: True

Determines whether the width of the column may be resized at run time using the mouse.

ComputeTotals property

```
property ComputeTotals : Boolean
```

Default: False

Instructs the ReportView to calculate totals for the current view column.

For event driven views (views of type TOvcReportView), the OnGetFieldAsFloat event must be implemented if you set ComputeTotals to True for any view field. The other two types of ReportViews implement OnGetFieldAsFloat internally, but only for numeric fields.

See also: Aggregate, TOvcReportView.OnGetFieldAsFloat

Field run-time, read-only property

```
property Field : TOvcRvField
```

Returns the global field definition to which this column refers.

The following example shows how to set the caption of a field in code. The code snippet sets the Caption property of the global field definition, which is being used in column 3 of view 0, to "Field 3."

```
  ReportView1.View[0].ViewField[3].Field.Caption := 'Field 3';
```

FieldName property

```
property FieldName : string
```

Returns the name of the global field definition to which this column refers.

The view uses this value to look up the global field definition for this column.

GroupBy property

```
property GroupBy : Boolean
```

↳ Determines whether the view is grouped (or categorized) by the current column.

When you have grouped columns in a view, the layout of the view changes. Each group is shown with a group header that shows the caption and the common element for the field.

When the group is expanded, each individual item in the group is shown as usual with the ordinary (non-grouped) columns, but the group field is left blank. Typically, you will want to resize the columns that have grouping enabled so that they are just wide enough to display the expand/collapse button.

Note, that grouping must always occur from left to right. You cannot group by a column if the column to the left of it is not grouped.

OwnerDraw property

```
property OwnerDraw : Boolean
```

Default: False

↳ Specifies whether the column will be drawn by the client application.

When OwnerDraw is True, the client application will draw the column. The ReportView fires an OnDrawViewField event whenever the column needs to be drawn. When OwnerDraw is False, the OnDrawViewField event is not generated and the field is drawn with the default drawing attributes.

See also: ShowHint, TOvcRvField.DefaultOwnerDraw,
TOvcCustomReportView.OnDrawViewField

OwnerReport — run-time, read-only property

```
property OwnerReport : TOvcCustomReportView
```

↳ A reference to the report view component that owns this view field definition.

OwnerView — run-time, read-only property

```
property OwnerView : TOvcCustomReportView
```

↳ A reference to the view that owns this field definition.

PrintWidth — property

```
property PrintWidth : Integer
```

Default: -1

↳ Determines the width of the column when the view is printed using the ReportView's Print method.

The value is in twips. One twip equals 1/1440 inch.

When the value of this property is –1, the default print width for the field is used instead.

See also: TOvcRvField.DefaultPrintWidth

ShowHint — property

```
property ShowHint : Boolean
```

Default: True

↳ Determines whether the report view should display a pop-up hint with the full text of the current cell when the mouse pointer hovers over it and the contents of the cell has been truncated to fit the width of the column.

Normally, ShowHint is left at True, as it is by default (the typical exception being owner) draw fields where the textual contents of the field may not be present or in a form suitable for display.

See also: OwnerDraw

Visible property

```
property Visible : Boolean
```

Default: True

Determines whether the current column is shown on screen or while printing.

Hidden columns are typically used for views where the user should not change the sort order. This is accomplished by having the only column (which is based on data) that can be sorted, be hidden.

Width property

```
property Width : Integer
```

Default: -1

The size of the view column on screen in pixels.

In addition to changing this value in the Object Inspector at design time, this value can be changed by directly sizing the columns of the view using the pointing device.

When this value is –1, the default width for the field is used instead.

See also: TOvcRvField.DefaultWidth

TOvcCustomReportView Class

TOvcCustomReportView class serves as the ancestor class for the TOvcReportView, TOvcDataReportView and TOvcDbReportView components. The TOvcCustomReportView class implements most of the functionality of the three ReportViews.

Hierarchy

TCustomControl (VCL)

 ❶ TOvcCustomControl (OvcBase). 16

  ❷ TOvcCustomControlEx (OvcBase) 19

   TOvcAbstractReportView (OvcRvIdx)

    TOvcCustomReportView (OvcRptVw)

Properties

❶	About		FieldWidthStore	Options
	ActiveView		GridLines	PageCount
	ActiveViewByTitle		GroupColor	PageNumber
❶	AttachedLabel		HaveSelection	PrinterProperties
	AutoCenter		HeaderImages	PrintDetailView
	AutoReset		HideSelection	Presorted
	ColumnResize		IsEmpty	Printing
❷	Controller		IsGrouped	SelectionCount
	CurrentGroup		ItemData	SmoothScroll
	CurrentItem		KeySearch	SortColumn
	CurrentView		KeyTimeout	SortDescending
	CustomViewStore	❶	LabelInfo	View
	Field		MultiSelect	Views
	Fields		OffsetOfData	

Methods

BeginUpdate	EndUpdate	Navigate
CenterCurrentLine	Enumerate	Print
ColumnFromOffset	EnumerateEx	PrintPreview
Count	EnumerateSelected	RebuildIndexes
EditCalculatedFields	ExpandAll	ScaleColumnWidths
EditCopyOfCurrentView	GetGroupElement	StretchDrawImageListImage
EditCurrentView	ItemAtPos	Total
EditNewView	MakeVisible	ViewNameByTitle

Events

❶ AfterEnter	OnEnumerate	OnPrintStatus
❶ AfterExit	OnExtern	OnSelectionChanged
OnDetailPrint	OnFilter	OnSignalBusy
OnDrawViewField	OnGetPrintHeaderFooter	OnSortingChanged
OnDrawViewFieldEx	❶ OnMouseWheel	OnViewSelect

Reference Section

ActiveView property

```
property ActiveView : string
```

Determines the currently active view by name.

The following example demonstrates how to switch to the view named “View2”:

```
{change view}
View.ActiveView := 'View2';
```

ActiveViewByTitle property

```
property ActiveViewByTitle : string
```

Determines the currently active view by title.

The following example shows how to set the active view using the view’s title. The example sets the view with the title “Sales Per Month” as the active view.

```
{change view}
View.ActiveViewByTitle := 'Sales Per Month';
```

AutoCenter property

```
property AutoCenter : Boolean
```

Default: False

Determines whether the report view scrolls the list so that the row representing the currently selected item is centered vertically within the list.

See also: CenterCurrentLine

AutoReset property

```
property AutoReset : Boolean
```

Default: False

Determines whether the report view automatically scrolls to the top whenever the contents change.

BeginUpdate method

```
procedure BeginUpdate;
```

Tells the ReportView that several data items will be added, removed, or changed, and that the display should not be updated until EndUpdate is called.

Using BeginUpdate and EndUpdate speeds up the process of changing data. You should normally use a try-finally block to ensure that the corresponding EndUpdate method is called.

The following example shows how to enclose multiple calls to AddData within a BeginUpdate/EndUpdate pair to speed up the process and prevent screen flicker during updates:

```
ReportView.BeginUpdate;
try
  for I := 0 to 99 do
    ReportView.AddData([Item[I]]);
finally
  ReportView.EndUpdate;
end;
```

See also: EndUpdate

CenterCurrentLine method

```
procedure CenterCurrentLine;
```

Centers the currently selected line vertically on the screen.

If no line is currently selected, CenterCurrentLine does nothing.

See also: AutoCenter

ColumnFromOffset method

```
function ColumnFromOffset(XOffset : Integer) : TOvcRvField;
```

Returns the report view field at a particular horizontal pixel offset within the report view.

If the offset is out of range or if no view is currently active, nil is returned. ColumnFromOffset can be used to implement context sensitive pop-up help for the report view.

ColumnResize property

```
property ColumnResize : Boolean
```

Default: True

↳ Determines whether the user is allowed to resize the width of the columns in views.

Count method

```
function Count(GroupRef : TOvcRvIndexGroup) : Integer;
```

↳ Returns the number of items in a group.

The GroupRef argument must be one passed from the ReportView in an event.

See also: Total, TOvcReportView.OnGetGroupString

CurrentGroup run-time property

```
property CurrentGroup : TOvcRvIndexGroup
```

↳ Returns the currently selected group.

The value is nil if no group is currently selected.

CurrentItem run-time property

```
property CurrentItem : Pointer
```

↳ Returns or sets the currently selected item.

The value is nil if no item is current.

The following example demonstrates how to use the current item in one ReportView to initialize another ReportView. The example implements a click handler for the first ReportView (MainView) which clears the current selection in the other ReportView (SubView). Then, if there is a currently selected item in MainView, it uses that item's data to fill the SubView with new data.

```
procedure TfrmMain.MainViewClick(Sender : TObject);
begin
  SubView.CurrentItem := nil; {clear current selection}
  SubView.Clear;
  if MainView.CurrentItem = nil then
    Exit;
  SubView.BeginUpdate;
  SubView.AddData(TMyData(MainView.CurrentItem).SubData);
  SubView.EndUpdate;
end;
```

CurrentView run-time, read-only property

```
property CurrentView : TOvcRVView
```

↳ Returns a reference to the currently selected view object.

The following example demonstrates how to use the CurrentView property to read information from the currently active view. In this case, the number of groups in the view are read and then displayed in the form's caption.

```
if ReportView.CurrentView <> nil then
  Form1.Caption :=
    'Groups:' + IntToStr(ReportView.CurrentView.GroupCount)
else
Form1.Caption := 'No view selected';
```

CustomViewStore property

```
property CustomViewStore : TOvcAbstractStore
```

↳ Connects a report view to one of the storage containers.

When a valid storage container is assigned to CustomViewStore, the report view will automatically store and reload customized view layouts as well as definitions for calculated fields. See "TOvcAbstractStore Class" on page 268 for more information on storage containers.

Note that since column widths are part of the overall view definition saved and loaded through the CustomViewStore storage container, the use of the FieldWidthStore property is redundant when CustomViewStore is used.

EditCalculatedFields method

```
function EditCalculatedFields : Boolean;
```

Displays the run-time editor for calculated fields.

If the user terminates the editor by pressing OK, True is returned. Otherwise, EditCalculatedFields returns False.

See also: EditCurrentView, TOvcRvField.Expression

EditCopyOfCurrentView method

```
function EditCopyOfCurrentView : Boolean;
```

Displays the run-time view editor with a copy of the current view.

If the user terminates the editor by pressing OK, True is returned and the new view is created. Otherwise, EditCopyOfCurrentView returns False and the new view is destroyed.

Note that for custom views to be persistent, a storage container must be assigned to the CustomViewStore property.

See also: CustomViewStore, EditCalculatedFields, EditCurrentView, EditNewView

EditCurrentView method

```
function EditCurrentView : Boolean;
```

Displays the current view in the run-time view editor.

If the user terminates the editor by pressing OK, True is returned and the view definition is updated. Otherwise, EditCurrentView returns False and the view definition is left unchanged.

See also: EditCalculatedFields, EditCopyOfCurrentView, EditNewView

EditNewView method

```
function EditNewView(const Title: string) : Boolean;
```

Displays the run-time view editor with a new blank view definition.

Pass the default title for the new view in the Title argument. The title may be customized by the user.

If the user terminates the editor by pressing OK, True is returned and the new view is made current, otherwise, EditCurrentView returns False, and the new view definition is discarded.

Note there is no method for deleting a view. A view is destroyed simply by freeing it (e.g., ReportView.CurrentView.Free).

See also: EditCalculatedFields, EditCopyOfCurrentView, EditCurrentView.

EndUpdate method

```
procedure EndUpdate;
```

Terminates the update mode initiated with BeginUpdate.

See also: BeginUpdate

Enumerate method

```
procedure Enumerate(UserData : Pointer);
```

Enumerates all items in the view, and calls OnEnumerate for each item.

UserData is a user-defined argument, which is passed on to the OnEnumerate event.

See also: EnumerateEx, EnumerateSelected, OnEnumerate

EnumerateEx — method

```
procedure EnumerateEx(Backwards, SelectedOnly : Boolean;
  StartAfter: Pointer; UserData : Pointer);
```

Enumerates items in the view and calls OnEnumerate for each item.

The Backwards flag, when True, instructs EnumerateEx to return items in the reverse order they appear on screen.

If SelectedOnly is True, only selected items are passed to OnEnumerate, otherwise all items are returned. Note that passing True for SelectedOnly is only allowed for report views that have the MultiSelect property set to True.

If StartAfter is nil, all items are passed to OnEnumerate. Pass a data value in StartAfter to have only the data items reported that appear after the StartAfter value has been seen.

UserData is a user-defined argument, which is passed on to the OnEnumerate event.

See also: Enumerate, EnumerateSelected, OnEnumerate

EnumerateSelected — method

```
procedure EnumerateSelected(UserData : Pointer);
```

Cycles through all items in the view that are currently selected and calls OnEnumSelected for each selected item.

Calling this method is only allowed when the MultiSelect property is set to True. UserData is a user-defined argument, which is passed on to the OnEnumerate event.

See also: EnumerateEx, OnEnumerate

ExpandAll — method

```
procedure ExpandAll(Expand : Boolean);
```

Expands or collapses all groups in the view.

When the Expand argument is True, ExpandAll will expand all groups in the view. When the Expand argument is False, ExpandAll will collapse all groups. This method affects views with grouped columns only.

Field property

```
property Field[Index : Integer] : TOvcRvField
```

An indexed property that returns a reference to the field definition at offset Index in the global field definition.

Fields property

```
property Fields : TOvcRvFields
```

The collection that holds the global field definition for the ReportView.

FieldWidthStore property

```
property FieldWidthStore : TOvcAbstractStore
```

Connects a report view to one of the storage containers.

When a valid storage container is assigned to FieldWidthStore, the report view will automatically store and reload column widths that are changed from their default by the user. See “TOvcAbstractStore Class” on page 268 for more information on storage containers.

See also: CustomViewStore

GetGroupElement method

```
function GetGroupElement(G: TOvcRvIndexGroup) : Pointer;
```

Returns an arbitrary data element for a particular group.

Combined with information from the current view definition about what columns are grouped, this property may be used to discover the grouping values.

The following example code is an excerpt from the ExRvDir example:

```
procedure TForm1.mnShowGroupInfoClick(Sender: TObject);
var
  GroupedFields: set of byte;
  i: Integer;
  Data: Pointer;
  S: string;
begin
  {determine what fields we are grouped on}
  GroupedFields := [];
  for i := 0 to OvcReportView1.CurrentView.ViewFields.Count - 1
```

```
do
    if OvcReportView1.CurrentView.
      ViewField[i].GroupBy then
      Include(GroupedFields,
OvcReportView1.CurrentView.ViewField[i].Field.Index)
    else
      break;
  {get a data instance from our group}
  Data :=
OvcReportView1.GetGroupElement(OvcReportView1.CurrentGroup);
  {build a string representing the grouped fields}
  with PFileRec(Data)^ do begin
    if fiFileName in GroupedFields then
      S := 'Name:' + SearchRec.Name
    else
      S := '';
    if fiSize in GroupedFields then
      S := S + ' Size:' + IntToStr(SearchRec.Size);
    if fiDate in GroupedFields then
      S := S + ' date/time:' + DateTimeToStr(
        FileDateToDateTime(SearchRec.Time));
    if fiPath in GroupedFields then
      S := S + ' path:' + Path;
    if fiType in GroupedFields then
      S := S + ' type:' + Ext;
  end;
  ShowMessage(S);
end;
```

GridLines — property

```
property GridLines : TOvcRVGridLines
```

```
TOvcRVGridLines = (glNone, glVertical, glHorizontal, glBoth);
```

Determines whether horizontal and/or vertical lines should be displayed to separate columns and items.

GroupColor — property

```
property GroupColor : TColor
```

Determines the background color used to paint grouped views.

HaveSelection — run-time, read-only property

```
property HaveSelection : Boolean
```

Returns True if any items are currently selected.

HaveSelection is only valid if MultiSelect is True. If MultiSelect is False, check the CurrentItem property for nil to determine if an item is selected.

HeaderImages — property

```
property HeaderImages : TImageList
```

Points to an image list used for displaying glyphs in the report view header to the left of the caption.

See also: TOvcRvField.ImageIndex.

HideSelection — property

```
property HideSelection : Boolean
```

Default: False

Determines whether the report view repaints the currently selected item with the normal drawing attributes when the report view loses the input focus.

IsEmpty run-time, read-only property

```
property IsEmpty : Boolean
```

↳ Returns True if the report view contains no data.

Note that IsEmpty refers to the report view as a whole, not the currently active view. The current view may be empty even if IsEmpty returns False. This occurs if the current view is filtered and no data meets the filtering criteria.

IsGrouped run-time, read-only property

```
property IsGrouped : Boolean
```

↳ Returns True if the currently active view contains grouped columns.

ItemAtPos method

```
function ItemAtPos(Pos : TPoint) : Pointer;
```

↳ Returns the data value of the item for a specific point within the report view control. This is typically used with pop-up menus to perform some action on the item under the mouse cursor.

If no item is at the current cursor position, ItemAtPos returns nil.

ItemData run-time, read-only property

```
property ItemData[Index: Integer] : Pointer
```

↳ Returns the data item at a particular vertical offset within the list.

See also: OffsetOfData

KeySearch property

```
property KeySearch : Boolean
```

↳ Determines whether the user should be allowed to search items by typing when the ReportView has focus.

The OnKeySearch event must be implemented for searching to work. See the ExRvDir example program for more information.

See also: GotoNearest, KeyTimeout, OnKeySearch

KeyTimeout property

```
property KeyTimeout : LongInt
```

Default: 1000

↳ Defines the time, in milliseconds, between keystrokes during incremental search before the report view considers the next key as the start of a new search phrase.

See also: KeySearch

MakeVisible method

```
procedure MakeVisible(Data : Pointer);
```

↳ Expands the group hierarchy so that the item is expanded if the data item represented by the Data argument is in a collapsed group.

MultiSelect property

```
property MultiSelect : Boolean
```

↳ Determines whether the user can select more than one item at a time.

When MultiSelect is True, selecting a group also selects every item contained within the group. The reverse is True for deselecting a group.

Navigate method

```
procedure Navigate(NewPosition : TRvCurrentPosition);
```

```
TRvCurrentPosition = (rvpMoveToFirst, rvpMoveToLast, rvpMoveToNext,
  rvpMoveToPrevious, rvpScrollToTop, rvpScrollToBottom);
```

↳ Allows the application to scrolls the list via code.

The various navigational codes have the following effect:

Code	**Meaning**
rvpMoveToFirst	Sets the first item in the list as the current item.
rvpMoveToLast	Sets the last item in the list as the current item.
rvpMoveToNext	Sets the item following the current item as current. If no item was current, this code has no effect.

Code	Meaning
rvpMoveToPrevious	Sets the item prior to the current item as current. If no item was current, this code has no effect.
rvpScrollToTop	Scrolls the embedded list so that the first item becomes visible. Does not affect the state of the currently selected item.
rvpScrollToBottom	Scrolls the embedded list so that the last item becomes visible. Does not affect the state of the currently selected item.

OffsetOfData — run-time, read-only property

```
property OffsetOfData[DataValue: Pointer]: Integer
```

Returns the offset of a given item in the current view. If the item is not currently visible, -1 is returned.

See also: ItemData, MakeVisible

OnDetailPrint — event

```
property OnDetailPrint: TNotifyEvent
```

Defines an event handler that is called immediately after each item has been printed when detail printing is enabled. This gives the program an opportunity to set the data in the detail view to the detail corresponding to the item just printed.

The following example is taken from the ExDbRv example:

```
procedure TForm1.OvcDbReportView1DetailPrint(Sender: TObject);
begin
  Table2.SetRange([Table1CustNo.Value], [Table1CustNo.Value]);
end;
```

See also: PrintDetailView

```
property OnDrawViewField : TOvcDrawViewFieldEvent
```

```
TOvcDrawViewFieldEvent = procedure(
  Sender : TObject; Canvas : TCanvas; Data : Pointer;
  ViewFieldIndex : Integer; Rect : TRect;
  const S : string) of object;
```

Defines an event handler that is called for view fields that have their OwnerDraw property set to True.

Canvas is the canvas of the ReportView. Data is a pointer to the data item which is to be drawn. ViewFieldIndex is the offset into the view definition of the current column. Rect is the rectangle of the canvas where the image should be drawn. S is a string containing the text of the field. When the OnDrawViewField event is invoked, the canvas has previously been initialized with the default drawing properties.

The following example demonstrates how to display a graphical percentage bar in a ReportView column. The data in the ReportView is assumed to be of type TMyData, which in turn is assumed to have fields called ValueA and ValueB. The percent value is calculated as round (ValueA * 100 / ValueB).

```
  procedure TfMain.ViewDrawSection(Sender : TObject;
    Canvas : TCanvas; Data : Pointer; ViewFieldIndex : Integer;
    Rect : TRect; const S : String);
  var
    P, H : Integer;
  begin
    with TMyData(Data) do begin
      if ValueB <> 0 then
        P := Round(ValueA * 100.0 / ValueB)
      else
        Exit;
    end;
    Canvas.Brush.Color := clWhite;
    Canvas.FillRect(Rect);
    H := Rect.Bottom - Rect.Top;
    H := H div 4;
    Canvas.Brush.Color := PColor[P];
    Canvas.Rectangle(Rect.Left + 2,Rect.Top + H,Rect.Left + 3 +
      Round((P * 0.9 * (Rect.Right - Rect.Left) / 100)),
      Rect.Bottom - H);
    Canvas.Brush.Color := clWhite;
  end;
```

Note that the report view supports another, more flexible, owner-draw event handler, OnDrawViewFieldEx. Note also, that if the report view defines handlers for both OnDrawViewField and OnDrawViewFieldEx, only the OnDrawViewFieldEx handler is called.

See also: OnDrawViewFieldEx, TOvcRvViewField.OwnerDraw

OnDrawViewFieldEx event

```
property OnDrawViewFieldEx : TOvcDrawViewFieldExEvent

TOvcDrawViewFieldExEvent = procedure(
  Sender : TObject; Canvas : TCanvas; Field : TOvcRvField;
  ViewField : TOvcRvViewField; var TextAlign : Integer;
  IsSelected, IsGroupLine : Boolean; Data : Pointer;
  Rect : TRect; const Text, TruncText : string;
  var DefaultDrawing : Boolean) of object;
```

↳ Defines an event handler that is called for view fields that have their OwnerDraw property set to True.

Canvas is the canvas of the ReportView.

Field is a reference to the report field being drawn.

ViewField is a reference to the view column being drawn.

TextAlign is a var argument, which may be set by the event handler. If the event handler leaves the DefaultDrawing argument at True (see below), the value of TextAlign is passed on to the DrawText Windows API function as the uFormat argument when the actual text is drawn.

IsSelected is a flag indicating whether the item being drawn is currently selected.

IsGroupLine is a flag indicating whether the line being drawn is a grouping line.

Data is a pointer to the data item that is to be drawn.

Rect is the rectangle of the canvas where the image should be drawn.

Text is a string containing the text of the field.

TruncText is a string containing the text truncated to fit the width of the drawing cell using the currently selected canvas font. If the cell is wide enough to hold the untruncated text, the value of TruncText is equal to that of Text.

DefaultDrawing is a flag, True by default, which may be set to False to suppress the report view's default drawing. Leave DefaultDrawing at True if all your drawing handler does is modify properties of the canvas, like font, background color, or alignment. Set DefaultDrawing to False if you are doing the full cell drawing yourself.

When the OnDrawViewFieldEx event is generated, the canvas has previously been initialized with the default drawing properties.

The following example demonstrates how to display a particular column in bold text:

```
procedure TForm1.OvcDbReportView1DrawViewFieldEx(
  Sender: TObject; Canvas: TCanvas; Field: TOvcRvField;
  ViewField: TOvcRvViewField; var TextAlign: Integer;
    IsSelected, IsGroupLine: Boolean;
  Data: Pointer; Rect: TRect; const Text, TruncText: String; var
DefaultDrawing: Boolean);
begin
  if Field.Index = 2 then
    Canvas.Font.Style := [fsBold];
end;
```

Note that the owner-draw event will only be generated for the views where the column has been marked as owner-draw.

See also: TOvcRvViewField.OwnerDraw

OnEnumerate event

```
property OnEnumerate : TOvcRVEnumEvent
```

```
TOvcRVEnumEvent = procedure(
  Sender : TObject; Data : Pointer; var Stop : Boolean;
  UserData : Pointer) of object;
```

Defines an event handler that is called for each item when the Enumerate method is called.

Data is a pointer to the item data for the current item. Stop determines if the enumeration should continue. Assign True to Stop to terminate the enumeration loop. UserData is a user-defined pointer passed to the Enumerate method.

The following example assumes that you have a Boolean flag, Include, in your data and shown in the current view. The code sets the Include flag to True or False, depending on the value of the UserData argument, and notifies the ReportView that the data has changed.

```
procedure TfMain.ViewEnum(Sender : TObject; Data : Pointer;
  var Stop : Boolean; UserData : Pointer);
begin
  case Integer(UserData) of
    1 :
      if not TDataDef(Data).Include then begin
        TUnitDef(Data).Include := True;
        View.ChangeData([Data]);
      end;
    2 :
      if TDataDef(Data).Include then begin
        TUnitDef(Data).Include := False;
        View.ChangeData([Data]);
      end;
  end;
  Application.ProcessMessages;
  if UserAbort then
    Stop := True;
end;
```

See also: Enumerate

OnExtern event

```
property OnExtern: TOvcRVExternEvent
```

```
TOvcRVExternEvent = procedure(
  Sender : TOvcAbstractReportView; const ArgList: Variant;
  var Result: Variant) of object;
```

↳ Defines an event handler that is called for evaluating application-defined extensions to the report view expression language called via the EXTERN function.

ArgList is a variant array containing the arguments passed to the EXTERN function. Result is a placeholder for the result of the function call.

The following example implements two functions, ProperCase, and PI, callable from within report view expressions via the generic EXTERN function. The generic EXTERN function takes a list of variants as arguments and returns a variant result. This example adopts the convention that the custom function to be called is specified as a string in the first argument.

To get the value of the field LastName with proper casing, you would use the expression “EXTERN(‘ProperCase’, LastName)”. To get the value of PI, you would use the expression “EXTERN(‘PI’)”.

```
procedure TForm1.OvcReportView1Extern(Sender:
TOvcAbstractReportView;
  const ArgList: Variant; var Result: Variant);
var
  ArgCount: Integer;
begin
  ArgCount := VarArrayHighBound(ArgList, 1) + 1;
  if UpperCase(ArgList[0]) = 'PROPERCASE' then begin
    if ArgCount > 2 then
      raise Exception.Create(
       'Too many arguments for ProperCase');
    if ArgCount = 1 then
      Result := 'Dummy'
    else
      Result := ProperCase(ArgList[1])
  end else
  if UpperCase(ArgList[0]) = 'PI' then begin
    Result := PI;
  end else
    raise Exception.Create(
      'Unsupported Extern function referenced:' + ArgList[0]);
end;
```

The actual ProperCase function is not shown.

See also: Report View Expression Syntax

OnFilter event

```
property OnFilter : TOrRVFilterEvent
```

```
TOrRVFilterEvent = procedure(
  Sender : TObject; Data : Pointer; FilterIndex : Integer;
  var Include : Boolean) of object;
```

Defines an event handler that is called for each item while building views to determine whether the items should be included in the view.

The event is passed the value of the FilterIndex property specified for the view. Data is a pointer to the data for this item. FilterIndex is the type of filtering requested by the ReportView’s indexer. The value comes from the FilterIndex property for the view definitions. Include is used to indicate whether this item should be included in the index for the current view. The default for the Include argument is True.

The following example shows how to implement a filter event. The example assumes that you have two numeric fields, Ordered and InStock, in your data. This code defines three views: FilterIndex 0 that shows only the data items with a value in Ordered greater than zero, FilterIndex 1 that shows only the data having Ordered equal to InStock, and FilterIndex 2 that shows data items where the value for InStock is less than the value for Ordered. The user can define the view simply by setting the FilterIndex property on the view definition.

```
procedure TfMain.ViewFilter(Sender : TObject; Data : Pointer;
FilterIndex : Integer; var Include : Boolean);
begin
  with TDataDef(Data)^ do
    case FilterIndex of
      0 : Include := Ordered > 0;
      1 : Include := Ordered = InStock;
      2 : Include := InStock < Ordered;
    end;
end;
```

See also: TOvcRvView.FilterIndex

OnGetPrintHeaderFooter property

```
property OnGetPrintHeaderFooter : TOvcGetPrintHeaderFooterEvent
```

```
TOvcGetPrintHeaderFooterEvent = procedure(
  Sender : TObject; IsHeader : Boolean ; Line : Integer;
  var LeftString, CenterString, RightString : string;
  var LeftAttr, CenterAttr, RightAttr : TFontStyles) of object;
```

↳ Defines an event handler that specifies headers and footers for the printed report.

IsHeader indicates whether the ReportView is about to draw a header or a footer. Line is the line number within the header or footer. LeftString, CenterString, and RightString are strings that are drawn in the header or footer to the left, centered, and to the right. To have only a centered line of text, leave the LeftString and RightString arguments blank. LeftAttr, CenterAttr, and RightAttr are the font attributes used to draw the header and footer strings. The default attribute is normal.

The following example demonstrates how to generate headers and footers when a ReportView is printed. The code prints a title, “Main Report,” in bold face on the upper left of the page, the current date on the upper right of the page, a user-name in italics to the lower left on the page, and finally the page number to the lower right. Note that this example assumes that you have both the PrintHeaderLines and PrintFooterLines properties in the PrinterProperties property set to 1.

```
procedure TfMain.ViewGetPrintHeaderFooter(
  Sender : TObject; IsHeader : Boolean; Line : Integer;
  var LeftString, CenterString, RightString : string;
  var LeftAttr, CenterAttr, RightAttr : TFontStyles);
begin
  if IsHeader then begin
    LeftString := 'Main Report';
    LeftAttr := [fsBold];
    RightString := DateTimeToStr(Now);
  end else begin
    LeftString := UserName;
    LeftAttr := [fsItalic];
    RightString := 'Page '+IntToStr(View.PageNumber);
  end;
end;
```

See also: PrinterProperties

OnPrintStatus event

```
property OnPrintStatus : TOvcPrintStatusEvent

TOvcPrintStatusEvent = procedure(
  Sender : TObject; Page : Integer; var Abort : Boolean) of object;
```

Defines an event handler that is called while the ReportView is printed.

This event can be used to implement a print status dialog. Page is the number of the currently printing page. Abort is a flag that can be set to True to abort the printing process. The default for Abort is False.

The following example shows how to update a print status dialog, which displays the currently printing page number and has an Abort button for cancelling the print process:

```
procedure var Abort : Boolean);
  TfMain.ViewPrintStatus(
Sender : TObject; Page : Integer;
begin
  PrintDlg.PageLabel.Caption := IntToStr(Page);
  PrintDlg.Update;
  Application.ProcessMessages;
  Abort := PrintDlg.AbortRequested;
end;
```

See also: Print

OnSelectionChanged event

```
property OnSelectionChanged : TNotifyEvent
```

Defines an event handler that is called if the selected state of items or groups in the report view changes.

Note that OnSelectionChanged may be triggered during mouse activity even though the final selection state did in fact not change. On the other hand, the user cannot change the selected state without having OnSelectionChanged triggered. You should interpret the OnSelectionChanged event to mean that selection may have changed.

See also: EnumerateSelected

OnSignalBusy property

```
property OnSignalBusy;
```

```
TOvcRvSignalBusyEvent = procedure(
  Sender : TObject; BusyFlag : Boolean) of object;
```

Defines an event handler that is called by the ReportView before and after it starts a potentially lengthy operation.

BusyFlag indicates whether the ReportView is entering or leaving busy mode.

The following example demonstrates how to change the mouse cursor to an hourglass when the ReportView starts a lengthy process and change it back to the default when the process is done:

```
procedure TfMain.ViewSignalBusy(
  Sender : TObject; BusyFlag : Boolean);
begin
  if BusyFlag then
    Screen.Cursor := crHourGlass
  else
    Screen.Cursor := crDefault;
end;
```

OnSortingChanged event

```
property OnSortingChanged : TNotifyEvent
```

↳ Defines an event handler that is called whenever the sort column or direction is changed. Use the SortColumn and SortDescending properties to determine the new sort order.

See also: SortColumn, SortDescending

OnViewSelect event

```
property OnViewSelect : TNotifyEvent
```

↳ Defines an event handler that is called whenever a new view is activated. Use the ActiveView or CurrentView properties to determine what view was activated.

See also: ActiveView, CurrentView

Options property

```
property Options : TOvcRVOptions

TOvcRVOptions = class(TPersistent)
  property HeaderAllowDragRearrange : Boolean
  property HeaderDrawingStyle : TOvcBHDrawingStyle
  property FooterDrawingStyle : TOvcBHDrawingStyle
  property HeaderAutoHeight : Boolean
  property HeaderHeight : Integer
  property HeaderLines : Integer
  property HeaderTextMargin : Integer
  property HeaderWordWrap : Boolean
  property FooterAutoHeight : Boolean
  property FooterHeight : Integer
  property FooterTextMargin : Integer
  property ListAutoRowHeight : Boolean
  property ListRowHeight : Integer
  property ListBorderStyle : TBorderStyle
  property ListColor : TColor
  property ListSelectColor : TovcColors
  property WheelDelta : Integer
end;
```

↳ Encapsulates a number of properties that affect the behavior and visual appearance of the three visual sub-components that make up a report view, namely the header, the footer and the list box.

For a definition of these properties, see “TOvcButtonHeader Component” on page 628 and “TOvcVirtualListBox Component” on page 689.

PageCount run-time, read-only property

```
property PageCount : Integer
```

↳ Returns the total number of print pages while the report view is being printed.

PageCount can be used from within OnGetPrintHeaderFooter to show the number of pages along with the current page number in the header or footer.

See also: OnGetPrintHeaderFooter, PageNumber, Print

PageNumber — run-time, read-only property

```
property PageNumber : Integer
```

Returns the currently printing page number while the ReportView is being printed.

PageNumber is typically used from within OnGetPrintHeaderFooter to show the page number in a header or footer.

See also: OnGetPrintHeaderFooter, PageCount, Print

PreSorted — run-time property

```
property PreSorted : Boolean
```

Optimizes the loading process.

If you are loading a large collection of data into a report view, which may only be sorted on one column and which already is sorted, you can optimize the loading process by setting the PreSorted property to True prior to loading the data. As long as PreSorted remains True, the report view does not attempt to sort the data.

Print — method

```
procedure Print(PrintMode : TRVPrintMode; SelectedOnly : Boolean);
```

```
TRVPrintMode = (pmCurrent, pmExpandAll, pmCollapseAll);
```

Prints the current view on the default printer.

PrintMode determines whether groups should be expanded in the case of a grouped view. SelectedOnly determines whether all items or only the currently selected items should be printed.

The possible values of PrintMode are shown in the following table:

Value	**Meaning**
pmCurrent	Print the view with the currently expanded/collapsed groups.
pmExpandAll	Print the view as if all groups were expanded.
pmCollapseAll	Print the view as if all groups were collapsed.

See also: OnGetPrinterHeaderFooter, OnPrintStatus, PrintDetailView, PrintPreview, PrinterProperties

PrintDetailView run-time property

`property PrintDetailView : TOvcCustomReportView`

Prints the current report view.

When the PrintDetailView property is set, the report view will generate an OnDetailPrint event for each item in the view and then proceed to print the body of the detail view as part of the current report. After the detail block is sent to the printer or preview, printing of the master view proceeds on the following line.

Use the OnDetailPrint event handler to load the detail data appropriate for the current item into the detail view.

See the ExDbRv sample application in the Examples directory for a fully working example of master/detail printing.

See also: OnDetailPrint, PrinterProperties

PrinterProperties property

`property PrinterProperties : TOvcRvPrintProps`

A persistent object containing properties pertaining to printing.

TOvcRvPrintProps has the properties shown in the following table:

Property Definition	**Default**	**Purpose**
AutoScaleColumns : Boolean	False	When true, AutoScaleColumns causes the report view to size the print width of each column to its actual contents. Note that the report view does not attempt to auto-size columns that are marked owner-draw.
MarginBottom : Integer	0	Specifies the print page's bottom margin in twips.
MarginLeft : Integer	0	Specifies the print page's left margin in twips.
MarginRight : Integer	0	Specifies the print page's right margin in twips
MarginTop : Integer	0	Specifies the print pages's top margin in twips.

Property Definition	Default	Purpose
DetailIndent : Integer	0	Indicates the amount of indentation (in twips), which should be applied to the detail information when PrintDetailView is assigned.
PrintColumnMargin : Integer	72	PrintColumnMargin indicates the space between columns while printing in twips (1 twip = 1/1440 inch).
PrintFont : TFont		PrintFont determines the font used for printing the ReportView.
PrintFooterLines : Integer	0	PrintFooterLines determines the number of footer lines used. The content of the footer is set in the OnGetPrintHeaderFooter event.
PrintHeaderLines : Integer	0	PrintHeaderLines determines the number of header lines used. The content of the header is set in the OnGetPrintHeaderFooter event.
PrintLineWidth : Integer	12	PrintLineWidth determines the width of printed lines in twips (1 twip = 1/1440 inch).

See also: OnGetPrinterHeaderFooter, Print

Printing run-time, read-only property

```
property Printing : Boolean
```

Returns True when the report view is rendering data for printing and False if it is not.

This property may be used in owner-draw handlers to discriminate between printing and screen painting.

The following example from the ExRvDir sample application uses the Printing flag to determine when to stretch an image list entry to the paper resolution:

```
procedure TForm1.OvcReportView1DrawViewFieldEx(
  Sender: TObject; Canvas: TCanvas; Field: TOvcRvField;
  ViewField: TOvcRvViewField; var TextAlign: Integer;
  IsSelected, IsGroupLine: Boolean; Data: Pointer;
  Rect: TRect; const Text, TruncText: String;
  var DefaultDrawing: Boolean);
var
  DrawCheck: Boolean;
begin
  DrawCheck := False;
  with PFileRec(Data).SearchRec do
    case Field.Index of
    fiArchive :
      DrawCheck := Attr and faArchive <> 0;
    fiReadOnly :
      DrawCheck := Attr and faReadOnly <> 0;
    fiHidden :
      DrawCheck := Attr and faHidden <> 0;
    fiSystem :
      DrawCheck := Attr and faSysFile <> 0;
    end;
  if DrawCheck then begin
    if not OvcReportView1.Printing then
      ImageList1.Draw(Canvas, Rect.Left, Rect.Top, iiCheck)
  else
    OvcReportView1.StretchDrawImageListImage(
      Canvas, ImageList1, Rect, iiCheck, True);
  DefaultDrawing := False;
  end;
end;
```

PrintPreview method

```
procedure PrintPreview(
  PrintMode : TRVPrintMode; SelectedOnly : Boolean);
```

Works like the Print method, but brings up the print preview dialog rather than printing directly.

See the Print method on page 179 for the definition and a discussion of the PrintMode and SelectedOnly arguments.

See also: OnGetPrinterHeaderFooter, OnPrintStatus, Print, PrintDetailView, PrinterProperties

RebuildIndexes method

```
procedure RebuildIndexes;
```

↳ Forces an unconditional rebuild of the view indexes.

Specifically, all view indexes are deleted and the index for the currently selected view is then created. Indexes for the remaining views are created when or if they are needed.

Note that calling RebuildIndexes will temporarily change the active view and delete and re-create any active groups. RebuildIndexes should not be called from within any of the report view's own event handlers.

ScaleColumnWidths method

```
procedure ScaleColumnWidths;
```

↳ Adjusts the column widths of the current view according to contents. Columns that have the owner-draw attribute set are not scaled.

Note that end-users may scale each column to its current content by double-clicking over the column's right-hand border in the view header.

See also: AutoScaleColumns, PrinterProperties

SelectionCount run-time, read-only property

```
property SelectionCount : Integer
```

↳ Returns the number of items currently selected in a multi-select report view.

If MultiSelect is False, SelectionCount always returns zero.

SmoothScroll property

```
property SmoothScroll : Boolean
```

↳ Determines the granularity used when scrolling the window area of the ReportView.

Setting SmoothScroll to True will cause the scroll logic to use a finer granularity for scrolling operations, resulting in a smoother scroll rate, but occupying more time.

SortColumn — run-time property

```
property SortColumn : Integer
```

↳ The column currently selected for sorting.

The following code example shows how to change the current sort column for the ReportView named “View” to 5. This is equivalent to the user clicking the sixth column with the mouse (the column number is zero based).

```
  View.SortColumn := 5; {change sort column}
```

See also: SortDescending

SortDescending — run-time property

```
property SortDescending : Boolean
```

↳ Indicates whether the current sort order is ascending or descending.

The following code example shows how to toggle the current sort order:

```
  View.SortDescending := not View.SortDescending;
```

See also: SortColumn

StretchDrawImageListImage — class method

```
class procedure StretchDrawImageListImage(
  Canvas: TCanvas; ImageList: TImageList; TargetRect: TRect;
  ImageIndex: Integer; PreserveAspect: Boolean);
```

↳ A utility method for stretch-drawing an image from an image list.

The report view uses this method internally when printing images in column headers.

Total — method

```
function Total(
  GroupRef : TOvcRvIndexGroup; Field : Integer) : Double;
```

↳ Returns the computed total value for fields that have ComputeTotals set to True.

Total is typically used from within the OnGetGroupString event to generate a total string for the relevant columns.

See also: OnGetGroupString, Count, TOvcRvViewField.Aggregate

View <span>run-time property</span>

```
property View[Index : Integer] : TOvcRVView
```

An indexed property returning the view object by index.

The following example shows how to use the View property to iterate through the view definitions, populating a list box with the view titles and a reference to each view definition:

```
var
  I : Integer;
begin
  for I := 0 to Pred(ReportView.Views.Count) do
    ListBox1.Items.AddObject (ReportView.View[I].Title,
      ReportView.View[I]);
end;
```

ViewNameByTitle method

```
function ViewNameByTitle(const Value : string) : string;
```

Returns the name of a view with a given title.

If no view with the Title specified in the Value argument exists, a blank string is returned. This function is mainly for use in component and property editors.

The following example shows how to look up a view name in a ReportView from its title:

```
S := ReportView.ViewNameByTitle('Main View');
if S = '' then
  ShowMessage('There is no view with the title "Main View"')
else
  ShowMessage('The name of the view with the title "Main View"
    is ' + S);
```

Views property

```
property Views : TOvcRvViews
```

The collection which contains the view definitions for the ReportView.

TOvcReportView Component

TOvcReportView implements a columnar list box with built-in functionality for sorting and grouping columns and for computing totals for numeric columns. TOvcReportView inherits all its functionality from TOvcCustomReportView.

TOvcReportView does not implement data management for the data displayed in the control (except for sorting indexes). In order to use this component, you must implement event handlers for at least the OnGetFieldAsString and the OnCompareFields events. Several other inherited events are published to allow you to implement features like total display and key searching.

Hierarchy

TCustomControl (VCL)

- ❶ TOvcCustomControl (OvcBase) 16
  - ❷ TOvcCustomControlEx (OvcBase) 19
    - TOvcAbstractReportView (OvcRvIdx)
      - ❸ TOvcCustomReportView (OvcRptVw). 154
        - TOvcReportView (OvcRptVw)

Properties

❶	About	❸	FieldWidthStore	❸	Options
❸	ActiveView	❸	GridLines	❸	PageCount
❸	ActiveViewByTitle	❸	GroupColor	❸	PageNumber
❶	AttachedLabel	❸	HaveSelection	❸	PrintDetailView
❸	AutoCenter	❸	HeaderImages	❸	PrinterProperties
❸	AutoReset	❸	HideSelection	❸	Presorted
❸	ColumnResize	❸	IsEmpty	❸	PrintDetailView
❷	Controller	❸	IsGrouped	❸	Printing
❸	CurrentGroup	❸	ItemData	❸	SelectionCount
❸	CurrentItem	❸	KeySearch	❸	SmoothScroll
❸	CurrentView	❸	KeyTimeout	❸	SortColumn
❸	CustomViewStore	❶	LabelInfo	❸	SortDescending
❸	Field	❸	MultiSelect	❸	View
❸	Fields	❸	OffsetOfData	❸	Views

Methods

	AddData	❸	EditNewView	❸	Navigate
	BeginUpdate	❸	EndUpdate	❸	Print
	CenterCurrentLine	❸	Enumerate	❸	PrintPreview
	ChangeData	❸	EnumerateEx	❸	RebuildIndexes
	Clear	❸	EnumerateSelected		RemoveData
❸	ColumnFromOffset	❸	ExpandAll	❸	ScaleColumnWidths
❸	Count	❸	GetGroupElement	❸	StretchDrawImageListImage
❸	EditCalculatedFields		GotoNearest	❸	Total
❸	EditCopyOfCurrentView	❸	ItemAtPos	❸	ViewNameByTitle
❸	EditCurrentView	❸	MakeVisible		

Events

❶	AfterEnter	❸	OnExtern		OnKeySearch
❶	AfterExit	❸	OnFilter	❶	OnMouseWheel
	OnCompareFields		OnGetFieldAsFloat	❸	OnPrintStatus
❸	OnDetailPrint		OnGetFieldAsString	❸	OnSelectionChanged
❸	OnDrawViewField		OnGetFieldValue	❸	OnSignalBusy
❸	OnDrawViewFieldEx		OnGetGroupString	❸	OnSortingChanged
❸	OnEnumerate	❸	OnGetPrintHeaderFooter	❸	OnViewSelect

Reference Section

AddData method

```
procedure AddData(const Data : array of Pointer);
```

Adds a list of data references to the ReportView.

The data references are transparent as far as the ReportView is concerned. They are passed to various events when the ReportView needs to compare fields or convert them to their textual or numeric representation.

The following two examples demonstrate how to add data to a ReportView.

The first example assumes that the data is stored in an array of object instances. The array can be passed directly to the AddData method.

```
var
  Data : array[0..9] of TMyDataClass;
begin
  View.BeginUpdate;
  try
    View.AddData(Data);
  finally
    View.EndUpdate;
  end;
end;
```

The second example assumes that the data is stored in an array of records. Since AddData expects an array of pointers, the code uses a for loop to pass the addresses of each data record one by one.

```
var
  Data : array[0..999] of TMyDataRecord;
  Count, I : Integer;
begin
  ...
  {Count initialize}
  ...
  View.BeginUpdate;
  try
    for I := 0 to Pred(Count) do
      View.AddData([@Data[I]]);
  finally
    View.EndUpdate;
  end;
end;
```

ChangeData — method

```
procedure ChangeData(const Data : array of Pointer);
```

Tells the ReportView that the items passed in the Data arguments have changed and that the entries for the view indexes should be updated accordingly.

An attempt to pass items to ChangeData that have not already been added with AddData or have since been removed with RemoveData will raise an EItemNotFound exception.

The following example shows how to invalidate all data references stored in Data, which is assumed to be an array of pointers. This will cause the ReportView to rebuild all indexes.

```
begin
  View.BeginUpdate;
  try
    View.ChangeData(Data);
  finally
    View.EndUpdate;
  end;
end;
```

See also: AddData, RemoveData

Clear — method

```
procedure Clear;
```

Removes all data references from all views.

Use Clear when you want to replace all data in the view.

See also: AddData, RemoveData

GotoNearest — method

```
procedure GotoNearest(DataRef : Pointer);
```

Changes the currently selected item in the ReportView to the one closest to the value in DataRef with regards to the active view and the currently selected sort order.

This method is mainly for use in response to an OnKeySearch event.

GotoNearest does not use the pointer passed in DataRef directly, but passes it to the OnCompareFields event during its search for the nearest item.

See also: OnKeySearch, TOvcCustomReportView.KeySearch

```
property OnCompareFields : TOvcRVCompareFieldsEvent
```

```
TOvcRVCompareFieldsEvent = procedure(
  Sender : TObject; Data1, Data2 : Pointer;
  FieldIndex : Integer; var Res : Integer) of object;
```

Defines an event handler that is called whenever the ReportView needs to compare two items.

The ReportView uses OnCompareField to sort when the headers are clicked, and for grouping views. FieldIndex is the index into the Fields collection of the ReportView. One of the following values should be returned in the Res argument:

Value	Meaning
-1	If the field at FieldIndex in Data1 < the same field in Data2.
0	If the field at FieldIndex in Data1 = the same field in Data2.
1	If the field at FieldIndex in Data1 > the same field in Data2.

Following is an example of a compare fields event. The code uses a case statement to determine what field from the global field list is being compared, then does the comparison with the compare function appropriate for the field type.

```
procedure
  TfMain.ViewCompareFields(
    Sender : TObject; Data1, Data2 : Pointer;
    FieldIndex : Integer; var Res : Integer);
begin
  case FieldIndex of
    0 :
      Res := CompareText(TMyData(Data1).Name,
        TMyData(Data2).Name);
    1 :
      Res := CompareFloat(TMyData(Data1).Value,
        TMyData(Data2).Value);
  end;
end;
```

OnGetFieldAsFloat event

```
property OnGetFieldAsFloat : TOvcRVGetFieldAsFloatEvent
```

```
TOvcRVGetFieldAsFloatEvent = procedure(
  Sender : TObject; Data : Pointer; FieldIndex : Integer;
  var Value : Double) of object;
```

Defines an event handler that is called for every item in columns that have their ComputeTotals property set to True.

The Value is used by the ReportView to compute totals for columns and groups. Data is the current data item. FieldIndex is the index into the global Field property for the current field definition. Value is the numeric value of the field to be returned by the event.

The following example returns the numerical value of a field. In this case, it is assumed that the data in the ReportView only has one column, namely column 1, which can display totals.

```
  procedure TfMain.ViewGetColumnAsFloat(Sender : TObject;
    Data : Pointer; FieldIndex : Integer; var Value : Double);
  begin
    case FieldIndex of 1 : Value := TMyData(Data)^.Value;
    end;
  end;
```

OnGetFieldAsString event

```
property OnGetFieldAsString : TOvcRVGetFieldAsStringEvent
```

```
TOvcRVGetFieldAsStringEvent = procedure(
  Sender : TObject; Data : Pointer; FieldIndex : Integer;
  var Str : string) of object;
```

Defines an event handler that is called by the ReportView to render a string representation of the item passed as an argument.

Data is the current data item. FieldIndex is the index into the global Field property for the current field definition. Str is the text rendition of the field to be returned by the event.

The following is an example of a GetFieldAsString event used to return a text representation of a field's data. The code uses the FieldIndex argument to determine which field is being requested, then uses the appropriate formatting function to return the string value.

```
procedure
  TfMain.ViewGetFieldAsString(Sender : TObject;
    Data : Pointer; FieldIndex : Integer; var Str : String);
begin
  with TMyData(Data)^ do
    case FieldIndex of
      0 : Str := Name;
      1 : Str := Format('%6.3f', [Value]);
    end;
end;
```

OnGetFieldValue event

```
property OnGetFieldValue: TOvcRVGetFieldValueEvent
```

```
TOvcRVGetFieldValueEvent = procedure(
  Sender : TObject; Data : Pointer; FieldIndex : Integer;
  var Value : Variant) of object;
```

Defines an event handler, which must be able to return the field value as a variant.

OnGetFieldValue is generated by the report view's expression logic. Report view expressions may be used for calculated fields, for filter expressions, and for view column aggregate expressions. If your application does not use any of these features you do not have to implement the OnGetFieldValue handler.

The Data argument is the current data item. FieldIndex is the index into the global Field property for the current field definition. Value is the variant value of the field to be returned by the handler.

The following example is an excerpt of the OnGetFieldValue handler for the ExRvDir sample application:

```
procedure TForm1.OvcReportView1GetFieldValue(Sender: TObject;
  Data: Pointer; FieldIndex: Integer; var Value: Variant);
begin
  with PFileRec(Data).SearchRec do
    case FieldIndex of
    fiFileName :
      Value := Name;
    fiSize :
      if Attr and faDirectory <> 0 then
        Value := 0
      else
        Value := Size;
    fiDate :
      Value := FileDateToDateTime(Time);
    fiPath :
      Value := PFileRec(Data).Path;
    end;
end;
```

OnGetGroupString event

```
property OnGetGroupString : TOvcRVGetGroupStringEvent
```

```
TOvcRVGetGroupStringEvent = procedure(
  Sender : TObject; FieldIndex : Integer;
  GroupRef : TOvcRvIndexGroup; var Str : string) of object;
```

↳ Defines an event handler that renders a string representation of group totals only when the ShowTotals property is True.

Data is the current data item. FieldIndex is the index into the global Field property for the current field definition. GroupRef is an opaque reference to the ReportView's internal group representation. It can be used as an argument to the Total and Count functions to retrieve the total or the item count for a group. Str is the text rendition of the field to be returned by the event.

The following example demonstrates how to implement a GetGroupString event. The example displays the formatted total for the data in column two. The actual value is retrieved from the ReportView by calling the Total function. In column 3, the view displays the average of the data from column two. This is accomplished by calling Total for column two and dividing with the value returned from the Count function.

```
procedure
  TfMain.ViewGetGroupString(Sender : TObject;
  FieldIndex : Integer; GroupRef : Pointer;
  var Str : String);
begin
  case FieldIndex of
    2 : Str := Format('%6.3f', [View.Total(GroupRef,2)]);
    3 :
      {average}
      if View.Count(GroupRef) <> 0 then
        Str := Format('%6.3f', [View.Total(GroupRef,2)/
        View.Count(GroupRef)])
      else
        Str := 'n/a';
  end;
end;
```

OnKeySearch event

```
property OnKeySearch : TOvcKeySearchEvent
```

```
TOvcKeySearchEvent = procedure(
  Sender : TObject; SortColumn : Integer;
  const KeyString : string) of object;
```

Defines an event handler that implements incremental searching.

The event is generated by the ReportView when the KeySearch property is True and when the user types while the ReportView has focus.

SortColumn is the column number that is currently used for sorting. KeyString is the string entered by the user.

The following example shows how to implement key searching in an event driven ReportView. Since the ReportView does not know the format of the data it displays, the event must build a temporary data item with the value of the key string converted to the appropriate data type. GotoNearest in turn calls the OnCompareFields event to locate the nearest item for the search string. This example assumes that key searching only makes sense for the first column in the view.

```
procedure TfMain.ViewKeySearch(
  Sender : TObject; SortColumn : Integer;
  const KeyString : string);
var
  MyData : TMyData;
begin
  MyData := TMyData.Create;
  try
    if SortColumn = 0 then begin
      MyData.Name := KeyString;
      View.GotoNearest(MyData);
    end;
  end;
  finally
    MyData.Free;
  end;
end;
```

RemoveData method

```
procedure RemoveData(const Data : array of Pointer);
```

Tells the ReportView that the items passed in the Data argument are to be removed from the views.

The following example shows how to remove the data stored as an array of pointers from the ReportView:

```
begin
  View.BeginUpdate;
  try
    View.ChangeData(Data);
  finally
    View.EndUpdate;
  end;
end;
```

See also: AddData, ChangeData, Clear

TOvcDataReportView Component

TOvcDataReportView has the same visual appearance—and mostly the same properties and events—as the TOvcReportView component. The difference is that TOvcDataReportView implements the Items property, which is a collection of data items displayed in the view.

Because TOvcDataReportView implements all data management and all necessary events to show and manipulate data in views, it is much easier to use than the lower-level TOvcReportView component. The downside is that you don't have quite as much control over how data is formatted, sorted, and so on.

In order to be able to manage and sort the data correctly, the ReportView needs to know what type of data each column in the view holds and how the data should be formatted for display. Therefore, TOvcDataReportView defines a custom field definition component, TOvcDataRvField. The Fields property of TOvcDataReportView uses this derived type as its item type.

Hierarchy

TCustomControl (VCL)

- ❶ TOvcCustomControl (OvcBase) 16
  - ❷ TOvcCustomControlEx (OvcBase) 19
    - TOvcAbstractReportView (OvcRvIdx)
      - ❸ TOvcCustomReportView (OvcRptVw). 154
        - TOvcDataReportView (OvcDRpVw)

Properties

❶ About	❸ CurrentView	❸ IsEmpty
❸ ActiveView	❸ CustomViewStore	❸ IsGrouped
❸ ActiveViewByTitle	Field	❸ ItemData
❶ AttachedLabel	❸ Fields	Items
❸ AutoCenter	❸ FieldWidthStore	❸ KeySearch
❸ AutoReset	❸ GridLines	❸ KeyTimeout
❸ ColumnResize	❸ GroupColor	❶ LabelInfo
❷ Controller	❸ HaveSelection	❸ MultiSelect
❸ CurrentGroup	❸ HeaderImages	❸ OffsetOfData
CurrentItem	❸ HideSelection	❸ Options

- ❸ PageCount
- ❸ PageNumber
- ❸ PrintDetailView
- ❸ PrinterProperties
- ❸ Presorted
- ❸ PrintDetailView
- ❸ Printing
- ❸ SelectionCount
- ❸ SmoothScroll
- ❸ SortColumn
- ❸ SortDescending
- ❸ View
- ❸ Views

Methods

- AddData
- BeginUpdate
- CenterCurrentLine
- ChangeData
- Clear
- ❸ ColumnFromOffset
- ❸ Count
- ❸ EditCalculatedFields
- ❸ EditCopyOfCurrentView
- ❸ EditCurrentView
- ❸ EditNewView
- ❸ EndUpdate
- ❸ Enumerate
- ❸ EnumerateEx
- ❸ EnumerateSelected
- ❸ ExpandAll
- ❸ GetGroupElement
- GotoNearest
- ❸ ItemAtPos
- ❸ MakeVisible
- ❸ Navigate
- ❸ Print
- ❸ PrintPreview
- ❸ RebuildIndexes
- RemoveData
- ❸ ScaleColumnWidths
- ❸ StretchDrawImageListImage
- ❸ Total
- ❸ ViewNameByTitle

Events

- ❶ AfterEnter
- ❶ AfterExit
- OnCompareCustom
- OnCompareFields
- ❸ OnDetailPrint
- OnDrawViewField
- OnDrawViewFieldEx
- OnEnumerate
- ❸ OnExtern
- OnFilter
- OnFormatFloat
- OnGetCustomAsString
- OnGetFieldAsFloat
- OnGetFieldAsString
- OnGetFieldValue
- OnGetGroupString
- ❸ OnGetPrintHeaderFooter
- OnKeySearch
- ❶ OnMouseWheel
- ❸ OnPrintStatus
- ❸ OnSelectionChanged
- ❸ OnSignalBusy
- ❸ OnSortingChanged
- ❸ OnViewSelect

Reference Section

CurrentItem property

```
property CurrentItem : TOvcDataRvItem
```

↳ Returns the currently selected item or nil if no item is selected.

Field property

```
property Field[Index : Integer] : TOvcDataRvField
```

↳ An indexed property that returns a reference to the field definition at offset Index in the global field definition.

Items property

```
property Items : TOvcDataRvItems
```

↳ The container for the data items displayed in the TOvcDataReportView component.

See “TOvcDataRvItems Class” on page 205 for more information.

OnCompareCustom event

```
property OnCompareCustom : TOvcDRVCompareCustomEvent
```

```
TOvcDRVCompareCustomEvent = procedure(
  Sender : TObject; FieldIndex: Integer; Data1, Data2: Pointer;
  var Result: Integer) of object;
```

↳ Defines an event handler called by the data report view to compare the values of fields with the dtCustom field type.

Data is a pointer to the buffer containing the raw field data for the field. FieldIndex is the index into the global Field property for the current field definition. Result should be set to the result of the comparison: zero if the fields are equal, a value less than zero if the field represented by the Data1 argument is less than that of Data2, and a value greater than zero otherwise.

If the data for all the dtCustom fields is actually PChars, the corresponding OnCompareCustom handler might look like that shown in the following example:

```
procedure TfrmCANote.rvProfileCompareCustom(Sender: TObject;
  FieldIndex: Integer; Data1, Data2: Pointer;
  var Result: Integer);
begin
  {All custom fields are assumed to be PChars }
  if PChar(Data1^) = PChar(Data2^) then
    Result := 0
  else
    Result := lStrCmp(PChar(Data1^), PChar(Data2^));
end;
```

See the entry for OnGetCustomAsString on page 201 for further background information on this example.

See also: TOvcDataRvField.DataType

OnDrawViewField event

```
property OnDrawViewField : TOvcDrawDViewFieldEvent

TOvcDrawDViewFieldEvent = procedure(
  Sender : TObject; Canvas : TCanvas; Data : TOvcDataRvItem;
  ViewFieldIndex: Integer; Rect : TRect;
  const S : string) of   object;
```

Identical to the OnDrawViewField event of the ancestor component, TOvcCustomReportView except that here, the Data argument is of type TOvcDataRvItem.

See also: TOvcCustomReportView.OnDrawViewField

OnDrawViewFieldEx event

```
property OnDrawViewFieldEx : TOvcDrawDViewFieldExEvent

TOvcDrawDViewFieldExEvent = procedure(
  Sender : TObject; Canvas : TCanvas; Field : TOvcRvField;
  ViewField : TOvcRvViewField; var TextAlign : Integer;
  IsSelected, IsGroupLine : Boolean; Data : TOvcDataRvItem;
  Rect : TRect; const Text, TruncText : string;
  var DefaultDrawing : Boolean) of object;
```

Identical to the OnDrawViewFieldEx event of the ancestor component, TOvcCustomReportView except that here, the Data argument is of type TOvcDataRvItem.

See also: TOvcCustomReportView.OnDrawViewFieldEx

OnEnumerate event

```
property OnEnumerate : TOvcDRVEnumEvent
```

```
TOvcDRVEnumEvent = procedure(
  Sender : TObject; Data : TOvcDataRvItem;
  var Stop : Boolean; UserData : Pointer) of object;
```

Identical to the OnEnumerate event of the ancestor component, TOvcCustomReportView except that here, the Data argument is of type TOvcDataRvItem.

See also: TOvcCustomReportView.OnEnumerate

OnFilter event

```
property OnFilter: TOvcDRVFilterEvent
```

```
TOvcDRVFilterEvent = procedure(
  Sender: TObject; Data: TOvcDataRvItem; FilterIndex: Integer;
  var Include: Boolean) of object;
```

Identical to the OnFilter event of the ancestor component, TOvcCustomReportView except that here, the Data argument is of type TOvcDataRvItem.

See also: TOvcCustomReportView.OnFilter

OnFormatFloat event

```
property OnFormatFloat : TOvcDRVFormatFloatEvent
```

```
TOvcDRVFormatFloatEvent = procedure(
  Sender: TObject; Index: Integer; const Value: double;
  var Result: string) of object;
```

Generated for fields that have a data type of dtFloat and also have the CustomFormat property set to True.

OnFormatFloat allows for completely customized formatting of floating point fields.

See also: See also: TOvcDataRvField.DataType, TOvcDataRvField.CustomFormat, TOvcDataRvField.Format

```
property OnGetCustomAsString : TOvcDRVGetCustomAsStringEvent

TOvcDRVGetCustomAsStringEvent = procedure(
  Sender : TObject; FieldIndex: Integer; Data: Pointer;
  var Result: string) of object;
```

Defines an event handler called by the data report view to render string representations of fields with the dtCustom field type.

Data is a pointer into the data report view’s buffer containing the raw field data for the field. FieldIndex is the index into the global Field property for the current field definition. Result is the text rendition of the field to be returned by the event.

When you have a string field that you wish to represent in a report view and the contents of the field are already stored in some sort of pool structure, it would be wasteful to store the field as a normal dtString field because that would force all the strings to be copied from the pool into the data report view’s data container.

Using a dtCustom type field instead and just storing a PChar pointer into the pool’s data in the data report view saves the overhead of coping the strings.

Since the data report view does not know how to convert fields of dtCustom to strings, you have to implement OnGetCustomAsString to provide that service to the data report view as shown in the following example:

```
  procedure TfrmSample.DataReportViewGetCustomAsString(
    Sender: TObject; FieldIndex: Integer; Data: Pointer;
    var Result: String);
  begin
    {Here, all custom fields are assumed to be PChars (we ignore the
    FieldIndex argument)}
    Result := string(PChar(Data^));
  end;
```

If the field has the CanSort property set to true, the OnCompareCustom handler needs to be implemented for the field.

See also: TOvcDataRvField.DataType

TOvcDataRvField Class

TOvcDataRvField is the field definition component used by TOvcDataReportView. TOvcDataRvField inherits from TOvcRvField, but extends it with properties that are needed by TOvcDataReportView for managing the data.

Hierarchy

TComponent (VCL)

- ❶ TOvcComponent (OvcBase) 21
  - ❷ TOvcCollectible (OvcBase) 41
    - TOvcAbstractRvField (OvcRvIdx)
      - ❸ TOvcRvField (OvcRptVw). 138
        - TOvcDataRvField (OvcDRpVw)

Properties

❶	About	❸	DefaultOwnerDraw	❸	ImageIndex
❸	Alignment	❸	DefaultPrintWidth	❷	Index
❸	CanSort	❸	DefaultWidth	❷	Name
❸	Caption	❷	DisplayText	❸	NoDesign
❷	Collection	❸	Expression	❸	OwnerReport
	CustomFormat		Format		
	DataType	❸	Hint		

Reference Section

CustomFormat property

```
property CustomFormat : Boolean
```

Default: False

Generates the OnCustomFormat event for the field when the field needs to be converted to a string when set to True.

Note that CustomFormat has no effect unless the field is of type dtFloat.

See also: DataType, TOvcDataReportView.OnCustomFormat

DataType property

```
property DataType : TOvcDRDataType
```

```
TOvcDRDataType = (
  dtString, dtFloat, dtInteger, dtDateTime,
  dtBoolean, dtDWord, dtCustom);
```

Defines how the data of the field is stored in the item.

The following table shows the DataTypes and how they are stored:

DataType	**Value of Field Stored As**
dtString	String
dtFloat	Extended floating-point value
dtInteger	LongInt value
dtDateTime	Date/time value
dtBoolean	Boolean value
dtDWord	DWord value
dtCustom	User-defined storage format

If any fields use the dtCustom DataType, the client must implement the data report view's handler for at least the OnGetCustomAsString event, which allows the data report view to render a text representation of the custom field. If any of the dtCustom fields also have their CanSort property set to true, the client must implement the data report view's OnCompareCustom handler as well.

Note that the dtDWord type is only supported for compilers that have native support for unsigned 32-bit integers.

Format property

```
property Format : string
```

Determines how the content of the field is formatted for display in the case of floating-point, integer, and date and time fields.

For floating-point and integer fields, the value is formatted using the Format function. For date/time fields, the FormatDateTime function is used. No special formatting is done for string and Boolean fields.

See the VCL on-line documentation on FormatFloat and FormatDateTime for the correct syntax for format strings.

TOvcDataRvItems Class

TOvcDataRvItems is the container for the data in a TOvcDataReportView.

Hierarchy

TObject (VCL)

    TOvcDataRvItems (OvcDRpVw)

Properties

Item

Methods

Add	Clear	Count

Reference Section

Add property

```
function Add : TOvcDataRvItem;
```

Appends a new item to the ReportView and returns a pointer to the new instance.

The following example shows how to add a new data record to the ReportView:

```
var
  NewNode : TOvcOutlineNode;
begin
  NewNode := DataView.Nodes.Add;
  NewNode.AsString[0] := 'Astrid';
  NewNode.AsFloat[1] := 35000;
end;
```

Clear method

```
procedure Clear;
```

Removes all data from the data container and clears the ReportView.

Count property

```
property Count : Integer
```

Returns the number of items currently in the Items container.

Item property

```
property Item[Index : Integer] : TOvcDataRvItem
```

An indexed property for reading and writing the items of the data container.

This example shows how to set the value of field 0 of item 123 to “Kurt.” It is assumed that field 0 is of type string and that you currently have more than 123 data items in the view.

```
DataView.Item[123].AsString[0] := 'Kurt';
```

TOvcDataRvItem Class

TOvcDataRvItem is the data type of individual data items in the TOvcDataReportView.

Hierarchy

TObject (VCL)

TOvcDataRvItem (OvcDRpVw)

Properties

AsBoolean	AsFloat	Data
AsDateTime	AsInteger	Selected
AsDWord	AsString	Value

Methods

SetCustom

4

AsBoolean property

```
property AsBoolean[Index : Integer] : Boolean
```

Gets or sets the field at Index in the current item in Boolean format.

Setting or reading the field with AsBoolean is only allowed when the field definition is of type dtBoolean.

The following example shows how to set the value of field 2 of item 123 to True. It is assumed that field 2 is of type Boolean and that you currently have more than 123 data items in the view.

```
DataView.Item[123].AsBoolean[2] := True;
```

The following example shows how to use the value of a Boolean field in a data view item in an expression:

```
if DataView.Item[0].AsBoolean[3] then
{do something}
```

AsDateTime property

```
property AsDateTime[Index : Integer] : TDateTime
```

Gets or sets the field at Index in the current item in the VCL's date/time format.

Setting or reading the field with AsDateTime is only allowed when the field definition is of the dtDateTime type.

The following example shows how to set the value of field 4 of item 123 to the current date. It is assumed that field 4 is of type date/time, and that you currently have more than 123 data items in the view.

```
DataView.Item[123].AsDateTime[4] := Date;
```

AsDWord property

```
property AsDWord[Index : Integer] : DWord
```

↳ Gets or set the field at Index in the current item in DWord (unsigned 32-bit integer) format.

Setting or reading the field with AsDWord is only allowed when the field definition is of the dtDWord type.

Note that the dtDWord field type is only supported on compilers that have native support for the unsigned 32-bit integer type.

The following example shows how to set the value of field 5 of item 123 to 7. It is assumed that field 5 is of numeric type and that you currently have more than 123 data items in the view.

```
  DataView.Item[123].AsDWord[5] := 7;
```

AsFloat property

```
property AsFloat[Index : Integer] : Extended
```

↳ Gets or sets the field at Index in the current item in extended format.

Setting or reading the field with AsFloat is only allowed when the field definition is of the dtFloat type.

The following example shows how to set the value of field 1 of item 123 to 123.54. It is assumed that field 1 is of type float and that you currently have more than 123 data items in the view.

```
  DataView.Item[123].AsFloat[1] := 123.54;
```

The following example shows how to get the value of field 6 of the first item (item zero) in the Items property:

```
  Value := DataView.Item[0].AsFloat[6];
```

AsInteger property

```
property AsInteger[Index : Integer] : LongInt
```

↳ Gets or sets the field at Index in the current item in LongInt format.

Setting or reading the field with AsInteger is only allowed when the field definition is of the dtInteger type.

The following example shows how to set the value of field 5 of item 123 to 7. It is assumed that field 5 is of numeric type and that you currently have more than 123 data items in the view.

```
DataView.Item[123].AsInteger[5] := 7;
```

AsString property

```
property AsString[Index : Integer] : string
```

↳ Gets or sets the field at Index in the current item in string format.

AsString can be used to set or read the field regardless of the data type defined in the field definition. AsString will convert the string to or from the native storage format of the field.

The following example shows how to set the value of field 0 of item 123 to “Kurt.” It is assumed that field 0 is of type string and that you currently have more than 123 data items in the view.

```
DataView.Item[123].AsString[0] := 'Kurt';
```

Data property

```
property Data : Pointer
```

↳ An application-defined pointer, which may be used to store a reference to data represented by the data report view item.

Beyond allocating space for it and initializing it to nil when the item is created, the data report view never reads or sets the Data property.

Selected property

```
property Selected : Boolean
```

↳ Gets or sets the selected state for the current item.

Accessing this property is only allowed when MultiSelect is True for the data ReportView.

SetCustom method

```
procedure SetCustom(Index : Integer; const Value; Size : DWord);
```

Sets the value of a field with the dtCustom field type.

Index is the index (in the Fields collection) of the field being set.

Value is a buffer holding the contents of the field.

Size is the size of the buffer to be transferred into the data report view storage for the field.

The following example creates a new data item and then stores the PChar pointer argument in its first field, which is assumed to be of type dtCustom:

```
procedure TForm1.AddPCharItem(P: PChar);
var
  Item: TOvcDataRvItem;
begin
  Item := OvcDataReportView1.Items.Add;
  Item.SetCustom(0, P, 4);
end;
```

Note that the example stores the PChar pointer itself, not the data it points to. The application is assumed to managed memory for the PChar points elsewhere.

Value property

```
property Value[Index : Integer] : Variant
```

Gets or sets the field at Index in the current item in variant format.

Value can be used to set or read the field regardless of the data type defined in the field definition (i.e., Value will convert the string to or from the native storage format of the field).

The Value property is mainly used internally by the report view's expression evaluator to read field values when evaluating expressions for filters, calculated fields, and aggregates.

TOvcDbReportView Component

TOvcDbReportView is a data-aware version of the ReportView component. TOvcDbReportView inherits most of its functionality from TOvcCustomReportView. The difference between the data-aware ReportView and the regular ReportView is that the data-aware version extracts its list of fields and their data from the associated data source. Binding of view fields to global fields is postponed until the underlying table or query is opened. Otherwise, defining views is done in the same way as for the non-data-aware ReportView.

Note: Since the content of the Fields collection is built at run-time when the data source is opened, any change to the properties of the Fields collection is lost between sessions. To make persistent changes to, say, the display format of fields in the db aware report view, you should instead define persistent fields for the data source and customize the display settings for the field there.

The data-aware ReportView is a convenient way to show a grouped view or a report with totals from a database. You should be aware, however, that the ReportView is essentially a memory- based component. The database back-ends and the VCL database interface do not support dynamic total calculation or grouping. The implication of this is that the data-aware report view must read all records from the source before it can display totals and groups. This is not a problem with small to moderate table sizes, but you should be aware of this fact if you are writing an application which will be deployed using large or server-based tables.

Hierarchy

TCustomControl (VCL)

- ❶ TOvcCustomControl (OvcBase) 16
  - ❷ TOvcCustomControlEx (OvcBase) 19
    - TOvcAbstractReportView (OvcRvIdx)
      - ❸ TOvcCustomReportView (OvcRptVw). 186
        - TOvcDbReportView (OvcDbRpV)

Properties

- ❸ ActiveViewByTitle
- ❶ AttachedLabel
- ❸ AutoCenter
- ❸ AutoReset
- ❸ ColumnResize
- ❷ Controller
- ❸ CurrentGroup
- ❸ CurrentItem
- ❸ CurrentView
- ❸ CustomViewStore
- DataSource
- ❸ Field
- ❸ Fields
- ❸ FieldWidthStore
- ❸ GridLines
- ❸ GroupColor
- ❸ HaveSelection
- ❸ HeaderImages
- ❸ HideSelection
- ❸ IsEmpty
- ❸ IsGrouped
- ❸ ItemData
- ❸ KeySearch
- ❸ KeyTimeout
- ❶ LabelInfo
- ❸ MultiSelect
- ❸ OffsetOfData
- ❸ Options
- ❸ PageCount
- ❸ PageNumber
- ❸ PrinterProperties
- ❸ PrintDetailView
- ❸ Presorted
- ❸ Printing
- RefreshOnMove
- ❸ SelectionCount
- ❸ SmoothScroll
- ❸ SortColumn
- ❸ SortDescending
- SyncOnOwnerDraw
- UseRecordCount
- ❸ View
- ❸ Views

Methods

- ❸ BeginUpdate
- ❸ CenterCurrentLine
- ❸ ColumnFromOffset
- ❸ Count
- ❸ EditCalculatedFields
- ❸ EditCopyOfCurrentView
- ❸ EditCurrentView
- ❸ EditNewView
- ❸ EndUpdate
- ❸ Enumerate
- ❸ EnumerateEx
- ❸ EnumerateSelected
- ❸ ExpandAll
- ❸ GetGroupElement
- ❸ ItemAtPos
- ❸ MakeVisible
- ❸ Navigate
- ❸ Print
- ❸ PrintPreview
- ❸ RebuildIndexes
- ❸ ScaleColumnWidths
- ❸ StretchDrawImageListImage
- ❸ Total
- ❸ ViewNameByTitle

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❸ OnDetailPrint
- ❸ OnDrawViewField
- ❸ OnDrawViewFieldEx
- OnEnumerate
- ❸ OnExtern
- ❸ OnFilter
- ❸ OnGetPrintHeaderFooter
- ❶ OnMouseWheel
- ❸ OnPrintStatus
- ❸ OnSelectionChanged
- ❸ OnSignalBusy
- ❸ OnSortingChanged
- ❸ OnViewSelect

Reference Section

DataSource property

```
property DataSource : TDataSource
```

↳ Defines the source of the data displayed in the ReportView.

OnEnumerate event

```
property OnEnumerate : TOvcDbRVEnumEvent
```

```
TOvcDbRVEnumEvent = procedure(
  Sender : TObject; var Stop : Boolean;
  UserData : Pointer) of object;
```

↳ Identical to the OnEnumerate event of the ancestor component, TOvcCustomReportView except that the enumerate event for the data-aware report view has no Data argument. Instead, the data-aware report view sets the current record of the underlying data source before generating each event.

See also: TOvcCustomReportView.Enumerate, TOvcCustomReportView.EnumerateEx, TOvcCustomReportView.EnumerateSelected, TOvcCustomReportView.OnEnumerate

RefreshOnMove property

```
property RefreshOnMove : Boolean
```

Default: True

↳ The underlying data source is synchronized to the report view so that when you click an item in the report view, enumerate, and so on, the current record of the data source follows that of the report view.

If you do not have other data controls attached to the report view, you may optimize the performance somewhat by setting RefreshOnMove to False. When RefreshOnMove is False, changing the currently selected in the data-aware report view does not change the current record on the data source.

SyncOnOwnerDraw property

```
property SyncOnOwnerDraw : Boolean
```

Default: False

Causes the data-aware report view to adjust the current record of the data source to the item being painted before each owner-draw event is generated when set to True.

This is necessary if your owner-draw logic relies on being able to read field data directly from the data source. For example, you might want to paint a field with colors that depend on another field in the same record. The reason this would not work without setting SyncOnOwnerDraw to True is that the report view caches field data internally and, by default, just passes the cached field string to the owner-draw handler.

Setting the current record pointer of the data source for each owner-draw event incurs a certain overhead, which is why SyncOnOwnerDraw is False by default.

UseRecordCount property

```
property UseRecordCount : Boolean
```

Default: False

Determines the number of records in the data source when it does a full refresh of its contents.

When UseRecordCount to False, as it is by default, the report view counts the records by iterating over them. If you know your database back-end maintains a record count, you can set UseRecordCount to True and get a significant performance benefit. As a general rule, most single-user database back-ends maintain a record count whereas many client-server database back-ends do not.

TOvcViewComboBox Component

The TOvcViewComboBox is a direct descendant of TOvcBaseComboBox (see page 534) and inherits many of its properties and events. The View Combo Box automatically displays a list of the defined views for a specified ReportView. This combo box reflects changes in the list of identified views available to the ReportView and allows a user to select a current view from the list of valid views.

When the user selects a view from the list in the combo box, that view is automatically selected as the current view in the ReportView. Conversely, changes in the ReportView, such as the creation or destruction of views, are reflected in the list displayed by the View Combo Box.

Hierarchy

TCustomComboBox (VCL)

❶ TOvcBaseComboBox (OvcCmBx). 534

TOvcViewComboBox (OvcRvCbx)

Properties

❶ About	❶ HotTrack	❶ ListIndex
❶ AttachedLabel	❶ ItemHeight	❶ MRUListColor
❶ AutoSearch	❶ KeyDelay	❶ MRUListCount
❶ Controller	❶ LabelInfo	ReportView
❶ DroppedWidth	❶ List	❶ Style

Methods

❶ AddItem	❶ ClearMRUList	❶ SelectionChanged
❶ AssignItems	❶ InsertItem	
❶ ClearItems	❶ RemoveItem	

Reference Section

ReportView property

```
property ReportView : TOvcCustomReportView
```

Attaches a ReportView to this combo box.

Report view expression syntax

The report view expression syntax is based on the expression syntax from Structured Query Language (SQL). The report view expression syntax does not quite implement the full SQL-92 syntax in all areas. On the other hand, it extends it others – most notably, it adds functions for formatting numbers, date-times, and integers.

The following section describes the report view expression syntax in detail.

The formal grammar for the report view expression syntax

The grammar for the report view expression syntax is as follows in EBNF-like format.

Syntax Conventions

Syntax	**Meaning**
Text in double-quotes	Denotes keywords and symbols. In actuality, the report view expression syntax is case-insensitive, so you do not have to type keywords in all upper caps.
Italic	Identifies user-supplied parameters, such as field names, or literal constants, that are not broken down further within the syntax.
[]	Denotes optional syntax.
{}	denotes repetition. The portion between brackets may appear zero or more times.
|	Denotes alternatives. One of the alternatives may appear.
()	Denotes grouping.

Syntax

ConditionalExpressionList = ConditionalExpression { “,” ConditionalExpression }

ConditionalExpression = ConditionalTerm { “OR” ConditionalTerm }

ConditionalTerm = ConditionalFactor { “AND” ConditionalFactor }

```
ConditionalFactor = [“NOT”] ConditionalPrimary
ConditionalPrimary = SimpleExpression
     [
      (  ( “=” | “<=” | “<“ | “>” | “>=” | “<>” )
SimpleExpression )
| BetweenClause
| InClause
| LikeClause
| "NOT"
        (
BetweenClause
        | LikeClause
        )
| IsTest
     ]
IsTest = “IS” [“NOT”] (“NULL” | “TRUE” | “FALSE” | “UNKNOWN”)
BetweenClause = “BETWEEN” SimpleExpression “AND” SimpleExpression
LikeClause = “LIKE” SimpleExpression [“ESCAPE” SimpleExpression ]
InClause = “IN” “(“ SimpleExpressionList ”)”
SimpleExpressionList = SimpleExpression { “,” SimpleExpression }
SimpleExpression = Term { (“+” | “-” | “||”) Term }
Term = Factor { (“*” | “/” ) Factor }
Factor = [ “-” ]
    (
    “(“ (CondExpression) ”)”
    | Field Reference
    | Literal
    | Aggregate
```

```
      | ScalarFunction
      )
Aggregate =
"COUNT(*)" |  ( "MIN" | "MAX" | "SUM" | "AVG" )  "(" SimpleExpression ")"
ScalarFunction = (
        "CASE"    CaseExpression
|  (  "CHARACTER_LENGTH"
      | "CHAR_LENGTH" )  "(" SimpleExpression ")"
      | "LOWER" "(" SimpleExpression ")"
      | "UPPER" "(" SimpleExpression ")"
      | "POSITION" "(" SimpleExpression "," SimpleExpression ")"
| "SUBSTRING" "(" SimpleExpression "FROM" SimpleExpression
[ "FOR" SimpleExpression ] ")"
     | "TRIM" "(" [ "LEADING" | "TRAILING" | "BOTH" ]
[SimpleExpression [ "FROM" SimpleExpression ] ")"
    | "FORMATNUMBER" "(" SimpleExpression "," SimpleExpression
[ "," CondExpression ] ")"
    | "FORMATDATETIME" "(" SimpleExpression "," SimpleExpression ")"
    | "INTTOHEX" "(" SimpleExpression ["," SimpleExpression ] ")"
    | "EXTERN" "(" ConditionalExpressionList ")"
  )
CaseExpression = WhenClauseList ["ELSE" ("NULL" | SimpleExpression ) ] "END"
WhenClauseList = WhenClause { WhenClause }
WhenClause = "WHEN" ConditionalExpression "THEN" ("NULL" | SimpleExpression )
Literal = ( floating point number | integer | StringLiteral | DateLiteral | TimeLiteral |
TimestampLiteral | BooleanLiteral )
BooleanLiteral =  "TRUE"  | "FALSE"
StringLiteral = SQLString
DateLiteral = "DATE" SQLString
```

TimeLiteral = “TIME” SQLString

TimestampLiteral = “TIMESTAMP” SQLString

SQLString = “‘” { NoQuote | “‘’” } “‘”

NoQuote = any character other than “‘” or end-of-line

Operators

The report view expression syntax supports the following operators in order of precedence (highest first):

Operator	**Definition**
`()`	`Parenthesis`
	`Unary minus`
`* /`	`Multiplicative operators`
`+ - ||`	`Additive operators`
`= >= <= < <> BETWEEN IN LIKE IS`	`Comparison operators`
`NOT`	`Unary logical NOT`
`AND`	`Logical AND`
`OR`	`Logical OR`

The || operator

The || operator is the official SQL string concatenation operator. The report view syntax treats || exactly the same as +.

The BETWEEN operator

The BETWEEN operator checks if the first argument is in the interval defined by the values on each side of AND – both ends of the interval included.

The expression

The following expressions are equivalent:

<Exp1> BETWEEN <Exp2> AND <Exp3>

<Exp1> >= <Exp2> AND <Exp1> <= <Exp3>

The following example always evaluated to True:

```
3 BETWEEN 2 AND 7
```

The IN operator

Definition: SimpleExpression “IN” “(“ SimpleExpressionList ”)”

The IN operator searches a list of alternatives for a match and returns True if a match was found.

The following example always evaluates to True:

```
'Two' IN ('One', 'Two', 'Three')
```

The LIKE operator

Definition: SimpleExpression "LIKE" SimpleExpression ["ESCAPE" SimpleExpression]

The LIKE operator scans the first operand for a match on the second operand according to the following rules:

The second operand is a string expression to search for, and may optionally one or more occurrences of the following:

‘%’ (match zero or more characters of any kind), and/or

‘_’ (match exactly one character of any kind)

If the escape expression is specified, it defines a character to prefix ‘%’ or ‘_’ with to indicate a literal ‘%’ or ‘_’, respectively, in the search phrase.

The search performed by the LIKE operator is case sensitive.

The following example evaluates to True if the string “Berkeley” exists anywhere in the first operand:

```
TextField LIKE '%Berkeley%'
```

The following example evaluates to True if the first operand is a character string exactly thee characters long and starts with an upper-case ‘S’:

```
TextField LIKE 'S__'
```

The following example evaluates to True if the length of the first operand is at least 4 characters and last but three is ‘c’:

```
TextField LIKE '%c___'
```

The following example evaluates to True if the first operand begins with an underscore:

```
TextField LIKE '=_%' ESCAPE '='
```

Supported functions

This section describes the built-in functions supported by the report view expressions.

AVG

Definition: AVG SimpleExpression

AVG calculates the average value of an expression for a group or an entire view.

AVG is an aggregate function and may only be used in the Aggregate definition on view fields. It may not be used in calculated fields or filter expressions.

Example:

```
AVG(Size)
```

See also: TOvcRvViewField.Aggregate

CASE

Definition: CASE WhenClause { WhenClause } [ELSE (NULL | SimpleExpression)] END

WhenClause = WHEN ConditionalExpression THEN (NULL | SimpleExpression)

Conditional expression. Checks each WHEN clause for a match. If a match is found, the result becomes that of the corresponding THEN clause. If no match is found, the result becomes that of the ELSE clause, or NULL if no ELSE clause is specified.

Examples:

```
CASE WHEN BooleanField THEN 'Yes' ELSE 'No' END
CASE WHEN IntegerField = 1 THEN "First"
WHEN IntegerField = 2 THEN "Second"
ELSE 'None of these' END
```

CHARACTER_LENGTH

Definition: CHARACTER_LENGTH(SimpleExpression)

May be abbreviated CHAR_LENGTH.

Returns the number of characters in the argument.

COUNT

Definition: COUNT(*)

Returns the number of rows in a group or an entire view.

COUNT(*) is an aggregate expression and may only be used in the Aggregate definition on view fields. In may not be used in calculated fields or filter expressions.

Example:

```
COUNT(*)
```

Note that the report view expression syntax only supports the asterisk syntax – it does not support counts on specific fields. Nor does it support the SQL DISTINCT modifier.

See also: TOvcRvViewField.Aggregate

EXTERN

Definition: EXTERN (ConditionalExpression { "," ConditionalExpression })

The EXTERN function forms a bridge between the report view expression evaluator and the client application. It provides a simple alternative to modifying the report view expression syntax parser and evaluation logic directly.

When the user invokes the EXTERN function from an expression, the report view generates the OnExtern event. The handler gets the parameter list and is responsible for evaluating a result and returning it.

See also: TOvcCustomReportView.OnExtern

FORMATDATETIME

Definition: FORMATDATETIME (SimpleExpression , SimpleExpression)

FormatDateTime formats the date, time, or date-time value in the first argument according to the mask specified as the second argument. The second argument is assumed to be of type text.

FormatDateTime maps directly to the VCL function FormatDatetime except that the arguments are reversed. Please see the VCL documentation for a description of the various formatting options.

FORMATNUMBER

Definition: FORMATNUMBER (SimpleExpression, SimpleExpression [, CondExpression])

FormatNumber formats the number in the first argument with the number of decimals indicated by the second argument. If the third optional, Boolean argument evaluates to True, FormatNumber will insert thousand separators.

The following example will typically yield the string “123,456.8”, subject to your local settings:

```
  FormatNumber(123456.78, 1, true)
```

INTTOHEX

Definition: INTTOHEX(SimpleExpression [, SimpleExpression])

IntToHex converts the first argument, which is assumed to represent an integer value, to a hexadecimal string with the number of places specified by the second, optional argument. If the second argument is omitted or has the value of zero, the width of the result is determined by the number of significant digits in the first argument.

Example:

```
IntToHex(Field1, 8)
```

LOWER

Definition: LOWER(SimpleExpression)

Lower converts the argument, which is assumed to be of text type, to lower case.

Example:

```
Lower('DO NOT SHOUT')
```

MIN

Definition: MIN SimpleExpression

MIN determines the smallest value of an expression for a group or an entire view.

MIN is an aggregate function and may only be used as part of the Aggregate definition on view fields. It cannot participate in calculated field expressions or filter expressions.

The following example returns the smallest value for Size in the group or view where it is used:

```
MIN(Size)
```

See also: TOvcRvViewField.Aggregate

MAX

Definition: MAX SimpleExpression

MAX determines the largest value of an expression for a group or an entire view.

MAX is an aggregate function and may only be used as part of the Aggregate definition on view fields. It cannot participate in calculated field expressions or filter expressions.

The following example returns the largest value for Size in the group or view where it is used:

```
MAX(Size)
```

See also: TOvcRvViewField.Aggregate

POSITION

Definition: POSITION(SimpleExpression , SimpleExpression)

Returns the position of the first argument in the second argument. Both arguments are assumed to be of type text. If the first argument is a blank string, one is returned. Otherwise, if the first argument is not found as part of the second, zero is returned. The search performed by Position is case sensitive.

The result of the following expression is 6:

```
POSITION("Power", "TurboPower")
```

SUBSTRING

Definition: SUBSTRING(SimpleExpression FROM SimpleExpression [FOR SimpleExpression])

Extracts a sub-string from the first argument, starting at the position indicated by the second argument and returns it. If the FOR phrase is present, the third argument indicates the number of characters to copy. If the FOR phrase is absent, the entire string starting with the position indicated by the FROM phrase is returned.

The following example returns “Power”:

```
SUBSTRING('TurboPower' FROM 5)
```

The following example returns “Pow”:

```
SUBSTRING('TurboPower' FROM 5 FOR 3)
```

SUM

Definition: SUM SimpleExpression

SUM determines the sum of the values for each row of an expression for a group or an entire view.

SUM is an aggregate function and may only be used as part of the Aggregate definition on view fields. It cannot participate in calculated field expressions or filter expressions.

The following example returns the sum of all sizes in the group or view where it is used:

```
SUM(Size)
```

See also: TOvcRvViewField.Aggregate

TRIM

Definition: TRIM([LEADING | TRAILING | BOTH] [SimpleExpression [FROM SimpleExpression])

TRIM trims either blank space or a specified trim character from the argument. If neither LEADING, TRAILING, or BOTH is specified, BOTH is assumed. If FROM is present, the first expression defines the character to trim and the second expression defines the input to be trimmed. If FROM is absent, the first expression defines the input to be trimmed.

The following example returns “TurboPower”:

```
TRIM(LEADING '  TurboPower')
```

The following example returns “TurboPowe”:

```
TRIM(TRAILING 'r' FROM 'TurboPower')
```

The following example returns “roomy”:

```
TRIM('  roomy ')
```

UPPER

Definition: UPPER(SimpleExpression)

Upper converts the argument, which is assumed to be of text type, to upper case.

Example:

```
Upper('this is loud')
```

Extending or modifying the report view expression syntax

As with everything else in Orpheus, it is possible to modify and extend the report view expression evaluator. Unfortunately, however, it isn’t a simple matter of inheriting from the existing parser. Before you decide to go ahead and change the expression evaluation syntax and logic directly, you should consider using the EXTERN function combined with OnExtern handler instead. That way, adopting future maintenance releases of Orpheus should be much less troublesome than if you customize the syntax directly.

If you do wish to modify the expression syntax, or if you are simply interested to learn about how it was made, the following section explains the details.

Generating the parser

The report view expression parser was built with Coco/r for Delphi. Coco/r is a compiler similar to lex/yacc. Coco/r was made by Michael Reith of Tetzel Inscriptions (www.tetzel.com) and released to the public domain. We include the version of Coco/r used to prepare Orpheus on the installation CD, but you can always download the latest version from the URL just mentioned.

The grammar for the report view expression language can be found in OvcRvExp.ATG. This is a Coco/r file that loads into the Coco/r environment.

When you compile the OvcRvExp grammar in Coco/r, the file OvcRvExp.pas will be generated. This is the actual parser. Naturally, a pre-generated OvcRvExp.pas file is provided as part of Orpheus for those who do not wish to customize the expression syntax.

The OvcRvExp.pas file that ships with Orpheus has been modified in two ways from the output generated by Coco/R: The default parser ancestor, cocobase.pas, has been copied to the Orpheus source tree and renamed to OvcCoco.pas, and the Uses list in OvcRvExp.pas has been changed to refer to the Orpheus specific copy of this file. This was done to isolate you from problems occurring if you install a different version of Coco/r than the one used to prepare Orpheus. The other manual change was to comment out a few unnecessary variable assignments generated by Coco/r that caused the Delphi compiler to (correctly) emit hints. It will be obvious to you what the changes are if you regenerate and then compile OvcRvExp.pas.

When the parser parses an expression, it builds a hierarchy of nodes that represent the expression in memory. This node hierarchy is then able to evaluate the result of the expression. This entire node hierarchy is defined in the OvcRvExpDef.pas unit.

4

Chapter 5: Outline Component

TOvcOutline represents a window that displays a hierarchical list of nodes, such as the files and directories on a disk, the master/detail relation, and data from a database or the headings in a document.

TOvcOutline makes it easy to have nodes load their child nodes on demand and to display nodes with radio buttons or check boxes next to them.

TOvcCustomOutline Class

The TOvcCustomOutline class is the immediate ancestor of the TOvcOutline component. It implements all of the methods and properties used by the TOvcOutline component and is identical to the TOvcOutline component except that no properties are published.

TOvcCustomOutline is provided to facilitate your creation of descendent Outline components. For property and method descriptions, see the “TOvcOutline Component” on page 231.

Hierarchy

TCustomControl (VCL)

    TOvcCustomControl (OvcBase) 16

        TOvcCustomControlEx (OvcBase) 19

            TOvcCustomVirtualListBox (OvcVLB) 688

                TOvcCustomOutline (OvcOutln)

TOvcOutline Component

The Outline component displays and manipulates an outline—sometimes referred to as a tree view. The Orpheus outline has built-in support for displaying outline nodes with check boxes or radio boxes, and for dynamically loading and freeing data as nodes are expanded and collapsed.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase). 16

❷ TOvcCustomControlEx (OvcBase) 19

TOvcCustomVirtualListBox (OvcVLB) 688

TOvcCustomOutline (OvcOutln) 230

TOvcOutline (OvcOutln)

Properties

	AbsNode		CurrentKey	Nodes
	AbsNodes		HideSelection	ShowButtons
❶	About		Images	ShowImages
	ActiveNode		Keys	ShowLines
❶	AttachedLabel	❶	LabelInfo	
❷	Controller		Node	

Methods

BeginUpdate	ExpandAll	LoadFromText
Clear	FindNode	SaveAsText
CollapseAll	LoadFromFile	SaveToFile
EndUpdate	LoadFromStream	SaveToStream

Events

❶ AfterEnter	OnCompareNodes	❶ OnMouseWheel
❶ AfterExit	OnDrawCheck	OnNodeClick
OnActiveChange	OnDrawText	OnNodeDestroy
OnCollapse	OnDynamicLoad	
OnClickHeader	OnExpand	

Reference Section

AbsNode — run-time property

```
property AbsNode[Index : LongInt] : TOvcOutlineNode
```

Defines all the nodes in the outline as a flat array sorted by the currently selected key.

Use the AbsNodes property to get the total number of nodes currently in the outline.

AbsNodes — run-time property

```
property AbsNodes : LongInt
```

Returns the total number of nodes currently in the outline, regardless of what level the nodes are at.

See also: AbsNode

ActiveNode — run-time property

```
property ActiveNode : TOvcOutlineNode
```

Reads or sets the currently selected Node in the outline. See “TOvcOutlineNode Class” on page 250 for more information.

BeginUpdate — method

```
procedure BeginUpdate;
```

Disables repainting.

BeginUpdate is used in combination with EndUpdate when multiple Nodes are added to or removed from the outline to prevent the control from repainting for each Node.

The following example adds three new nodes to an outline in a BeginUpdate/EndUpdate block:

```
procedure TForm1.Button1Click(Sender : TObject);
begin
  OvcOutline1.BeginUpdate;
  try
    OvcOutline1.Nodes.Add('Node 1');
    OvcOutline1.Nodes.Add('Node 2');
    OvcOutline1.Nodes.Add('Node 3');
  finally
    OvcOutline1.EndUpdate;
  end;
end;
```

See also: EndUpdate

Clear method

```
procedure Clear;
```

Removes all data in the outline.

CollapseAll method

```
procedure CollapseAll;
```

Causes the outline to collapse all expanded nodes.

See also: ExpandAll

CurrentKey property

```
property CurrentKey : Integer
```

Determines the current sort order for the outline nodes.

By default, the nodes are sorted by their text property, but additional keys can be defined by implementing an OnCompareNodes event handler and setting the Keys property. Note that the sort order does not affect the parent/child relationship between outline nodes, only the order in which children appear under their parent.

See also: Keys, OnCompareNodes

EndUpdate method

```
procedure EndUpdate;
```

Enables repainting.

EndUpdate is used in combination with BeginUpdate when multiple nodes are added to or removed from the outline to prevent the control from repainting with each change to the outline.

See also: BeginUpdate

ExpandAll method

```
procedure ExpandAll;
```

Causes the outline to expand all collapsed nodes, including dynamic nodes and their children.

FindNode method

```
function FindNode(const Text : string) : TOvcOutlineNode;
```

Searches the outline for a node with a particular value in its Text property.

FindNode returns the absolute node with a Text value greater or equal to the search phrase. If no node exists which meets the search criteria, nil is returned.

HideSelection property

```
property HideSelection : Boolean
```

Default: False

When HideSelection is true, the currently selected item is drawn with the select color only when the outline has focus.

Images property

```
property Images : TImageList
```

Points to an optional image list which holds images to display with each node.

The ImageIndex property (see page 252) determines what image from the Images list is drawn for each node.

Keys property

```
property Keys : Integer
```

Default: 1

↳ Specifies the number of sort orders defined for the Nodes in the outline.

Key zero is built into the outline, and provides sorting by the text property of the Nodes. Additional keys require that you implement an OnCompareNodes event handler.

See also: CurrentKey, OnCompareNodes, TOvcOutlineNode.AddIndex

LoadFromFile method

```
procedure LoadFromFile(const FileName : string);
```

↳ Loads data into the outline from a binary file.

The format of this file is proprietary to the Orpheus outline control. LoadFromFile is used to load data, which has previously been stored with SaveToFile. To load the contents of an outline from a text file, use LoadFromText.

See also: LoadFromText, SaveToFile

LoadFromStream method

```
procedure LoadFromStream(Stream : TStream);
```

↳ Loads data into the outline from a stream.

The format of the outline stream is proprietary to the Orpheus outline control. LoadFromStream is typically used to load data, which has previously been stored with SaveToStream.

See also: SaveToStream

LoadFromText method

```
procedure LoadFromText(const FileName : string);
```

↳ Loads the contents of an outline from a text file.

The indentation level is assumed to be represented by tab characters in the text file.

Node property

```
property Node[Index : LongInt] : TOvcOutlineNode
```

↳ Returns a specific node in the outline.

Index is the visible offset in the outline. Note that this is different from accessing the indexed Node sub-property of the Nodes property. For example, OvcOutline1.Node[5] would return the Node displayed at line 5 in the outline, whereas OvcOutline1.Nodes.Node[5] returns the fifth Node at the top level in the data. In other words, the Node property returns the visual elements by line.

Nodes property

```
property Nodes : TOvcOutlineNodes
```

↳ Holds the nodes shown in the outline.

Click on the ellipsis button to the right of the Nodes property in the Object Inspector to add Nodes at design time. See the BeginUpdate method on page 233 for an example.

See “TOvcOutlineNodes Class” on page 245 for more information.

OnActiveChange event

```
property OnActiveChange : TOvcOlActiveChangeEvent
```

```
TOvcOlActiveChangeEvent = procedure(
  Sender : TOvcCustomOutline;
  OldNode, NewNode : TOvcOutlineNode)of object;
```

↳ Defines an event handler that is called when the user selects an Node in the outline.

OldNode contains the value of the previously selected node. NewNode contains the value of the node which is about to be selected. If no node was previously selected, OldNode is a null pointer. Likewise, if no new node is selected but the previously selected node is deselected, NewNode is a null pointer.

The following example changes an outline node's ImageIndex property to 1 when it gets selected, and changes it back to 0 when it gets deselected:

```
procedure TForm1.OvcOutline1ActiveChange(
  Sender : TOvcCustomOutline;
  OldNode, NewNode : TOvcOutlineNode);
begin
  if OldNode <> nil then
    OldNode.ImageIndex := 0;
  if NewNode <> nil then
    NewNode.ImageIndex := 1;
end;
```

See page 252 for more information on the ImageIndex property.

OnCollapse event

```
property OnCollapse : TOvcOlNodeEvent

TOvcOlNodeEvent = procedure(
  Sender : TOvcCustomOutline; Node : TOvcOutlineNode) of object;
```

↳ Defines an event handler that is generated when a previously expanded node is collapsed.

OnCompareNodes event

```
property OnCompareNodes : TOvcOlCompareNodesEvent

TOvcOlCompareNodesEvent = procedure(
  Sender : TOvcCustomOutline; Key : Integer;
  Node1, Node2 : TOvcOutlineNode; var Result : Integer) of object;
```

↳ Defines an event handler that is called when nodes are added to the outline.

By default, nodes are sorted by their Text property. Setting the outline's Keys property to a value higher than 1 will cause the outline to generate OnCompareNodes events when Nodes are added to the outline. The event should compare the two nodes for the specified key. The Result value should be less than zero if the key in Node1 is lower than the key in Node2, zero if the two keys are equal, and greater than zero if the key in Node1 is greater than the key in Node2.

The following example assumes that each outline node has an associated Data property, which points do an instance of the application level defines class TMyData. The event provides sorting for the data's FloatField field.

```
type
  TMyData = class
    FloatField : Double;
  end;

procedure TForm1.OvcOutline1CompareNodes(
  Sender : TOvcCustomOutline; Key : Integer; Node1,
  Node2 : TOvcOutlineNode; var Result : Integer);
begin
  case Key of
    0 :; {0 is the built-in Text key}
    1 : if TMyData(Node1.Data).FloatField <
          TMyData(Node2.Data).FloatField then
          Result := -1
        else if TMyData(Node1.Data).FloatField =
          TMyData(Node2.Data).FloatField then
          Result := 0
        else
          Result := 1;
  end;
end;
```

See also: CurrentKey, Keys

OnDrawCheck event

```
property OnDrawCheck : TOvcOlDrawCheckEvent

TOvcOlDrawCheckEvent = procedure(
  Sender : TOvcCustomOutline; Canvas : TCanvas;
  Node: TOvcOutlineNode; Rect : TRect;
  Style : TOvcOlNodeStyle; Checked : Boolean;
  var DefaultDrawing : Boolean) of object;
```

Defines an event handler that implements owner drawing of the check boxes in the outline.

The following example is from the ExOutln2 example project (available in the examples directory) and illustrates how to draw the Checked state using images from an image list:

```
procedure TForm1.OvcOutline1DrawCheck(Sender: TOvcCustomOutline;
  Canvas: TCanvas; Node: TOvcOutlineNode; Rect: TRect;
  Style: TOvcOlNodeStyle; Checked: Boolean; var DefaultDrawing:
Boolean);
begin
  if OwnerDrawChecks then begin
    DefaultDrawing := False;
    case Style of
    osRadio :
      begin
        if Checked then
          ImageList2.Draw(Canvas, Rect.Left, 0, 0)
        else
          ImageList2.Draw(Canvas, Rect.Left, 0, 1);
      end;
    osCheck :
      begin
        if Checked then
          ImageList2.Draw(Canvas, Rect.Left, 0, 2)
        else
          ImageList2.Draw(Canvas, Rect.Left, 0, 3);
      end;
    end;
  end;
end;
```

OnDrawText event

```
property OnDrawText : TOvcOlDrawTextEvent
```

```
TOvcOlDrawTextEvent = procedure(
  Sender : TOvcCustomOutline; Canvas : TCanvas;
  Node : TOvcOutlineNode; const Text : string;
  Rect : TRect; var DefaultDrawing : Boolean) of object;
```

↳ OnDrawText defines an event handler that implements owner drawing of the text in the outline.

If all you want to do is change the color of font style of the text, you can do that simply by modifying the canvas argument in the event. The outline will initialize the canvas with the default drawing attributes before calling OnDrawText. If DefaultDrawing is left with its default True value, the outline will draw the text using the currently active properties of the canvas. Of course, you can also set DefaultDrawing to False in the event and handle all text drawing in code. The following example of an OnDrawText event handler changes the text color of root nodes to red:

```
procedure TForm1.OvcOutline1DrawText(
  Sender : TOvcCustomOutline; Canvas : TCanvas;
  Node : TOvcOutlineNode; const Text : String;
  Rect : TRect; var DefaultDrawing : Boolean);
begin
  if Node.Parent = nil then
    Canvas.Font.Color := clRed;
end;
```

OnDynamicLoad event

```
property OnDynamicLoad : TOvcOlNodeEvent
```

```
TOvcOlNodeEvent = procedure(
  Sender : TOvcCustomOutline; Node : TOvcOutlineNode) of object;
```

Defines an event handler that is generated when a node with the Mode property set to omDynamic or omDynamicLoad is expanded.

OnDynamicLoad adds child nodes to a node dynamically instead of having to load the entire outline up front. See page 253 for information about the Mode property.

The following example adds child nodes to an outline node in response to the user expanding the node, which is assumed to have its mode set to omDynamic or omDynamicLoad:

```
procedure
  TForm1.OvcOutline1DynamicLoad(
    Sender : TOvcCustomOutline; Node : TOvcOutlineNode);
begin
  OvcOutline1.BeginUpdate;
  try
    OvcOutline1.Nodes.AddChild(Node, 'Node 1');
    OvcOutline1.Nodes.AddChild(Node, 'Node 2');
    OvcOutline1.Nodes.AddChild(Node, 'Node 3');
  finally
    OvcOutline1.EndUpdate;
  end;
end;
```

OnExpand event

```
property OnExpand : TOvcOlNodeEvent
```

```
TOvcOlNodeEvent = procedure(
  Sender : TOvcCustomOutline; Node : TOvcOutlineNode) of object;
```

Defines an event handler that is generated when any collapsed node is expanded.

OnNodeClick event

```
property OnNodeClick : TOvcOlNodeEvent
```

```
TOvcOlNodeEvent = procedure(
  Sender : TOvcCustomOutline; Node : TOvcOutlineNode) of object;
```

Defines an event handler that is generated whenever the Checked state changes for outline nodes that have their Style set to osRadio or osCheck.

The following example displays the checked state of an outline node in the caption of a form when the checked state changes.

```
  procedure TForm1.OvcOutline1NodeClick(
    Sender : TOvcCustomOutline; Node : TOvcOutlineNode);
  begin
    if Node.Checked then
      Caption := 'Node is checked'
    else
      Caption := 'Node is not checked';
  end;
```

See also: Style

OnNodeDestroy event

```
property OnNodeDestroy : TOvcOlNodeEvent
```

Defines an event handler that is generated for each Node in the outline immediately before the Node is destroyed.

This gives the program a chance to destroy any custom data objects stored in the Node's Data property. For example, if the node's Data property is used to store pointers to objects, the OnNodeDestroy event can be used to free those objects when the nodes that own them are destroyed, as shown in the following example:

```
procedure
  TForm1.OvcOutline1NodeDestroy(
  Sender : TOvcCustomOutline; Node : TOvcOutlineNode);
begin
  TObject(Node.Data).Free;
end;
```

SaveAsText method

```
procedure SaveAsText(const FileName : string);
```

Saves the contents of the outline as a text file.

The text file is indented with TAB characters to reflect the parent/child relationship of nodes. Saving the contents of an outline as text does not preserve the Checked state and ImageIndex property of the node. See the "TOvcOutlineNode Class" on page 250 for more information on these properties. To preserve these properties as well, use SaveToFile.

See also: SaveToFile

SaveToFile method

```
procedure SaveToFile(const FileName : string);
```

Saves the contents of an outline in a binary file.

In addition to the text and the parent/child relationship among nodes, the values of the ImageIndex, Style, Checked and Mode properties are preserved. See the "TOvcOutlineNode Class" on page 250 for more information on these properties.

SaveToStream method

```
procedure SaveToStream(Stream : TStream);
```

Saves the contents of an outline to a binary stream.

In addition to the text and the parent/child relationship among nodes, the values of the ImageIndex, Style, Checked and Mode properties are preserved. See "TOvcOutlineNode Class" on page 250 for more information on these properties.

ShowButtons property

```
property ShowButtons : Boolean
```

↳ Determines whether the outline should display buttons for expanding and collapsing Nodes that have children.

Note that you can still expand and collapse nodes using the keyboard and by double-clicking even when buttons are not displayed.

ShowImages property

```
property ShowImages : Boolean
```

↳ Determines whether images should be displayed to the left of each Node.

To display images, you must also connect an image list to the Images property of the outline and set the ImageIndex property (see page 252) for the Node to a valid index in the image list.

See also: Images

ShowLines property

```
property ShowLines : Boolean
```

↳ Determines whether the outline should draw dotted lines to connect child nodes to parent nodes.

TOvcOutlineNodes Class

The TOvcOutlineNodes class defines a persistent object that contains the nodes of an outline control.

Hierarchy

TPersistent (VCL)

  TOvcOutlineNodes (OvcOutln)

Properties

Count	Node

Methods

Add	AddChildEx	AddObject
AddButtonChild	AddChildObject	AddObjectEx
AddButtonChildObject	AddChildObjectEx	Clear
AddChild	AddEx	

Reference Section

Add method

```
function Add(const S : string) : TOvcOutlineNode;
```

Creates a new top-level outline node, adds it to the outline, and returns the new instance.

S is the new node's text.

AddButtonChild method

```
function AddButtonChild(
  Node : TOvcOutlineNode; const S : string;
  InitStyle : TOvcOlNodeStyle;
  InitChecked : Boolean) : TOvcOutlineNode;
```

```
TOvcOlNodeStyle = (osPlain, osRadio, osCheck);
```

Adds a child node with a button style to an existing outline node.

The button style can be either check box or radio button. The InitChecked argument determines the initial state of the node's Checked property. Checked is documented in the TOvcOutlineNode Class on page 169.

AddButtonChildObject method

```
function AddButtonChildObject(
  Node : TOvcOutlineNode; const S : string;
  Ptr : Pointer; InitStyle : TOvcOlNodeStyle;
  InitChecked : Boolean) : TOvcOutlineNode;
```

Works like AddButtonChild, but lets you specify a value for the new node's Data property as well (the Ptr argument).

See also: AddButtonChild

AddChild method

```
function AddChild(
  Node : TOvcOutlineNode; const S : string) : TOvcOutlineNode;
```

Adds a child node to an existing outline node.

Node is the existing Node's parent. S is the text of the new node.

AddChildEx method

```
function AddChildEx(Node : TOvcOutlineNode;
  const S : string; InitImageIndex : Integer;
  InitMode : TOvcOlNodeMode) : TOvcOutlineNode;
```

Works the same as AddChild, but lets you specify values for the ImageIndex and Mode properties of the new node.

ImageIndex and Mode are documented in the “TOvcOutlineNode Class” on page 250.

See also: AddChild

AddChildObject method

```
function AddChildObject(
  Node : TOvcOutlineNode; const S : string;
  Ptr : Pointer) : TOvcOutlineNode;
```

Works the same as AddChild but lets you specify a value for the Data property of the new node.

S is the text of the new node. Ptr is the Data property of the new node.

See also: AddChild

AddChildObjectEx method

```
function AddChildObjectEx(
  Node : TOvcOutlineNode; const S : string;
  Ptr : Pointer; InitImageIndex : Integer;
  InitMode : TOvcOlNodeMode) : TOvcOutlineNode;
```

Works the same as AddChildObject but lets you specify values for the ImageIndex and Mode properties of the new node as well.

ImageIndex and Mode are documented in the “TOvcOutlineNode Class” on page 250.

See also: AddChildObject

5

AddEx method

```
function AddEx(
  const S : string; InitImageIndex : Integer;
  InitMode : TOvcOlNodeMode) : TOvcOutlineNode;
```

Works the same as Add but lets you specify values for the ImageIndex and Mode properties of the new node as well.

ImageIndex and Mode are documented in TOvcOulineNodeClass on page 169.

See also: Add

AddObject method

```
function AddObject(
  const S : string; Ptr : Pointer) : TOvcOutlineNode;
```

Works the same as Add but lets you specify the value for the Data property of the new node in the Ptr argument.

See also: Add

AddObjectEx method

```
function AddObjectEx(
  const S : string; Ptr : Pointer; InitImageIndex : Integer;
  InitMode : TOvcOlNodeMode) : TOvcOutlineNode;
```

Works the same as AddObject but lets you specify values for the ImageIndex and Mode properties of the new node as well.

ImageIndex and Mode are documented in the “TOvcOutlineNode Class” on page 250.

See also: AddObject

Clear method

```
procedure Clear;
```

Destroys all child nodes and refreshes the outline display.

Count property

```
property Count : LongInt
```

Returns the number of nodes.

Node property

```
property Node[Index : LongInt] : TOvcOutlineNode
```

Returns a specific node from the list of child nodes.

The Index is the offset of the child node with respect to the current key.

TOvcOutlineNode Class

The TOvcOutlineNode class is the class of Nodes in the Orpheus outline.

Hierarchy

TPersistent (VCL)

TOvcOutlineNodes (OvcOutln)

Properties

AddIndex	HasParent	Outline
Checked	ImageIndex	Parent
Count	Index	Style
Data	Level	Text
Expanded	Mode	Visible
HasChildren	Node	

Methods

Collapse	Invalidate	IsSibling
DeleteChildren	IsFirstSibling	
Expand	IsLastSibling	

Reference Section

AddIndex run-time property

```
property AddIndex : Integer
```

↳ The value of the AbsNodes property when the node was added.

The value of AddIndex can be used to sort on in an OnCompareNode handler to allow the outline nodes to be shown in the order they were added.

Checked property

```
property Checked : Boolean
```

↳ Determines whether the current node is checked.

Setting the Checked property to True for a node with the osRadio style will cause the current node to be selected and all siblings with the osRadio style to be de-selected. Setting Checked to True for a node with the osCheck style will cause the check box showing to the left of the Node's text to appear checked. For nodes with the osPlain style, setting the Checked property has no effect.

Collapse method

```
procedure Collapse(Recurse : Boolean)
```

↳ Causes a specific node to collapse and updates the display.

The Recurse argument determines whether child nodes (if any) should collapse as well.

Count property

```
property Count : LongInt
```

↳ Returns the number of child nodes attached to the current node.

Data property

```
property Data : Pointer
```

↳ Stores application specific data associated with each outline node.

DeleteChildren method

```
procedure DeleteChildren;
```

↳ Deletes all children of the node.

If any of the children have children of their own, these are deleted as well.

Expand method

```
procedure Expand(Recurse : Boolean);
```

↳ Expands the current node.

If the Recurse argument is True, any child nodes are expanded as well.

Expanded property

```
property Expanded : Boolean
```

↳ Expanded reads or sets the current expanded or collapsed state of a node.

Setting Expanded does not expand or collapse child nodes. To expand or collapse both a node and its children, use the Expand or Collapse methods.

See also: Collapse, Expand

HasChildren property

```
property HasChildren : Boolean
```

↳ Returns True if the node has children, False if it does not.

HasParent property

```
property HasParent : Boolean
```

↳ Returns True if the current node has a parent (i. e. it is not a top-level node).

ImageIndex property

```
property ImageIndex : Integer
```

Default: -1

↳ Determines what image, if any, should be displayed to the left of the text for the current node.

The value is used as an index into the image list specified in the outline's Images property. See page 235 for more information about the Images property.

Index property

```
property Index : LongInt
```

Returns the visual offset in the outline of the current node, if the current node is currently visible (the node's entire parent hierarchy is expanded). If the node is in a collapsed portion of the outline, Index returns –1.

IsFirstSibling method

```
function IsFirstSibling : Boolean
```

Returns true if the current node is the first of its parent's child nodes, with respect to the current sort order.

IsLastSibling method

```
function IsLastSibling : Boolean
```

Returns true if the current node is the last of its parent's child nodes, with respect to the current sort order.

IsSibling method

```
function IsSibling(Value : TOvcOutlineNode): Boolean;
```

Returns True if the current node is a sibling to the node passed in the Value argument.

Level property

```
property Level : Integer
```

Returns the indent level for the current node counted from the left.

Top-level nodes have a level of zero. Their children have a level of 1. Their children, in turn, have a level of 2, and so on.

Mode property

```
property Mode : TOvcOlNodeMode

TOvcOlNodeMode  = (omPreload, omDynamicLoad, omDynamic)
```

Determines how a node should respond to Expand and Collapse commands.

The most common mode is omPreload, which indicates that the node's children are already loaded. The two other modes, omDynamicLoad and omDynamic, both indicate that the node's children are to be determined only when the node is actually expanded by the user. When the node is expanded, the outline will generate an OnDynamicLoad event for the node, giving the application a chance to load data for the relevant child nodes and add them to the outline. When the node is in omDynamicLoad Mode, the node will keep the child nodes once they have been initialized. In contrast, omDynamic causes the child nodes to be destroyed again when the node is collapsed. In other words, with omDynamicLoad, the OnDynamicLoad will only fire once for each node. With omDynamic, OnDynamicLoad event will be generated each time the node is expanded and collapsed. See page 241 for documentation of the OnDynamicLoad event.

See also: Collapse, Expand

Node property

```
property Node[Index : LongInt] : TOvcOutlineNode
```

↳ An indexed property of a node's child nodes.

Use Node to access individual child nodes. Use the Count property to get the number of child nodes. The Index is relative to the current sort key.

See also: Count

Outline property

```
property Outline : TOvcCustomOutline
```

↳ Provides access to the owner of the outline node.

The owner is always a TOvcCustomOutline. See "TOvcCustomOutline Class" on page 230 for more information.

Parent property

```
property Parent : TOvcOutlineNode
```

↳ Provides access to the immediate parent of the outline node.

If the current node is at the top-most level, Parent is nil.

Style property

```
property Style : TOvcOlNodeStyle
```

```
TOvcOlNodeStyle = (osPlain, osRadio, osCheck)
```

Determines the node's appearance and behavior.

With osPlain style, the node is displayed simply with its text. With osRadio, the node is displayed with a radio button to the left of the text. When the user clicks the radio button or presses the space bar, the current Node is selected and any siblings at the same level, which also have the osRadio style will automatically be deselected. With osCheck, the node is displayed with a check box to the left of the text.

See also: Checked

Text property

```
property Text : string
```

Determines the display text for the node.

Visible property

```
property Visible : Boolean
```

Indicates whether all the node's ancestors are currently expanded.

Setting a node's Visible property to True will expand any ancestor nodes that are currently collapsed. Setting Visible to False has no effect. Note that a node may still be clipped by the drawing logic even if its Visible property is True—when the node is either above or below the current view on screen.

Chapter 6: Orpheus State Components

The Orpheus State series of non-visual components (TOvcFormState, TOvcComponentState, and TOvcPersistantState) allow you to write applications that can automatically store and recall the state (current status and property values) of forms, components, and other persistent classes. For forms, you can store the position, size, active control, and the form's state (minimized, maximized, or normal). For components (and any class derived from TPersistent) you can store and recall the value of nearly any published property.

TOvcFormState has only a few properties, that determine how it will perform. TOvcComponentState is a little more complex, but offers a powerful property editor that allows easy selection of only those properties you want to maintain. TOvcPersistantState is used at run time to store and recall all the published properties of an instance of any class descended from TPersistent. TOvcAbstractState is the ancestor class for two of the state components, TOvcFormState, and TOvcComponentState. It implements functionality common to both components.

TOvcFormState, TOvcComponentState, and TOvcPersistantState each have a Storage property that defines a component that performs the physical reading and writing of information. Orpheus provides four non-visual Store components: TOvcRegistryStore, TOvcIniFileStore, TOvcVirtualStore, and the new TO32XMLStore. The Store components read and write information for any of the State components. TOvcRegistryStore implements a mechanism for saving and loading information in the Windows Registry. TOvcIniFileStore uses a Windows INI file to store information. TO32XMLStore uses a standard XML format to store information. In addition, TOvcVirtualStore provides several event handlers that allow you to implement just about any storage scheme you can think of. TOvcAbstractStore defines basic functionality and is the ancestor of each of the store components.

TOvcAbstractState Class

TOvcAbstractState is the ancestor class for TOvcFormState and TOvcComponentState. It provides some common functionality for those components, but does not publish any properties. You will not normally create an instance of TOvcAbstractState; rather, you could use it as an ancestor for your own new State component.

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

TOvcAbstractState (OvcState)

Properties

❶	About	Section
	Active	Storage

Methods

RestoreState	SaveState

Events

OnRestoreState	OnSaveState

Reference Section

Active property

```
property Active : Boolean
```

Default: True

Active determines if the state information is saved or loaded.

If Active is True, the state information (defined by descendant classes) is loaded at the time the parent form is created. Conversely, if Active is True, state information is saved when the parent form is about to close or be destroyed. If Active is False, no action is taken.

The actual process of loading and saving state information is performed by the store component that is assigned to the Storage property.

See also: Storage

OnRestoreState event

```
property OnRestoreState : TNotifyEvent
```

OnRestoreState defines an event handler that is called after the state information has been restored.

See also: OnSaveState

OnSaveState event

```
property OnSaveState : TNotifyEvent
```

OnSaveState defines an event handler that is called after the state information has been saved.

See also: OnRestoreState

RestoreState method

```
procedure RestoreState;
```

RestoreState reads and restores all of the previously stored properties.

See also: SaveState

SaveState method

```
procedure SaveState;
```

SaveState save the values of the specified properties.

Section property

```
property Section : string
```

Default: The parent form's name

Section is the name of the location where state information is stored.

Section has a different meaning depending on the type of store component that is assigned to the Storage property. If a TOvcIniFileStore is being used, Section is the name of the INI file section. If a TOvcRegistryStore is being used, Section represents the node or lowest element of the key name. For TOvcVirtualStore, the interpretation of Section is entirely dependent on your implementation of the TOvcVirtualStore events.

See also: Storage, TOvcIniFileStore, TOvcRegistryStore, TOvcVirtualStore

Storage property

```
property Storage : TOvcAbstractStore
```

Storage defines the component that is used to load and save state information.

When a class is assigned to this property, the methods of the class are used to read and write state information. If Storage is left unassigned, no state information is saved or loaded, much like setting the Active property to False.

See also: Active

TOvcFormState Component

TOvcFormState is a descendant of the TOvcAbstractState class and implements a component that allows you to save and restore the state of any form. In addition to the capabilities provided by its ancestor, TOvcFormState adds an Option property that lets you determine which state information should be maintained.

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

❷ TOvcAbstractState (OvcState) 258

TOvcFormState (OvcState)

Properties

❶ About	Options	❷ Storage
❷ Active	❷ Section	

Methods

❷ RestoreState	❷ SaveState

Events

❷ OnSaveState	❷ OnRestoreState

Options property

```
property Options : TOvcFormStateOptions
```

```
TOvcFormStateOptions = set of TOvcFormStateOption;
```

```
TOvcFormStateOption = (
  fsState, fsPosition, fsActiveControl, {$IFDEF VERSION4},
  fsDefaultMonitor{$ENDIF});
```

Default: [fsState, fsPosition]

↳ Options determines which of the form's state items are saved and restored.

Possible options are any combination of the following:

Value	**Description**
fsState	The state of the window: minimized, maximized, or normal.
fsPosition	The position and size of the form.
fsActiveControl	The value of the form's ActiveControl property.
fsDefaultMonitor	The value of the DefaultMonitor property (Delphi/C++Builder 4 or later).

TOvcComponentState Component

TOvcComponentState descends from TOvcAbstractState and allows you to save and restore virtually any property of any component. It introduces one additional property, StoredProperties, that you use to specify which properties are maintained.

StoredProperties is simply a list of the component and property names that you want to save and restore values for. A sample list of strings for this property is shown below:

```
'OvcReportView1.ActiveView'
'OvcSlider1.Orientation'
'OvcPictureField1.DataType'
'OvcReportView1.Font'
```

Given this list of components and properties, TOvcComponentState (and its store component) handle everything necessary to save and restore the listed property values.

To make the process of selecting and maintaining each list of properties, Orpheus provides a sophisticated property editor that displays a list of components and a list of each property belonging to the selected component. Using this property editor, you can add, delete, and reorder the list of properties that you want maintained.

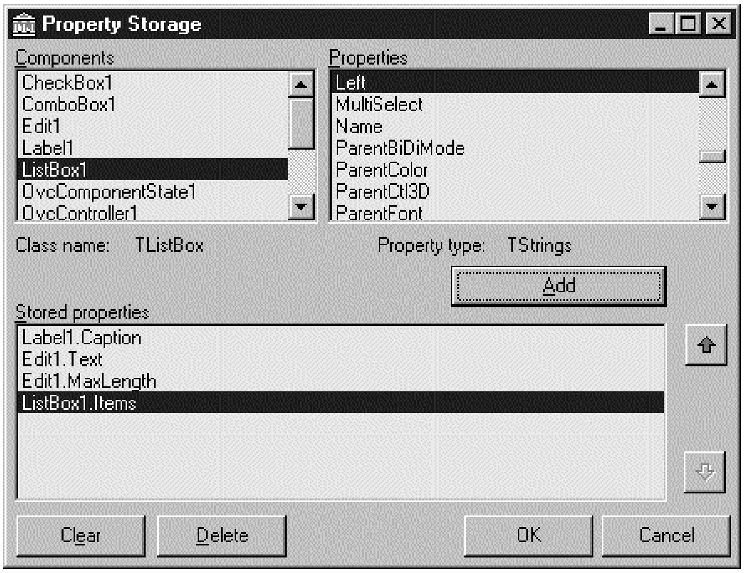


Figure 6.1: Property Storage property editor

Hierarchy

TComponent (VCL)

- ❶ TOvcComponent (OvcBase) 21
  - ❷ TOvcAbstractState (OvcState) 258
    - TOvcComponentState (OvcState)

Properties

❶ About	❷ Section	StoredProperties
❷ Active	❷ Storage	

Methods

❷ RestoreState	❷ SaveState

Events

❷ OnSaveState	❷ OnRestoreState

StoredProperties property

```
property StoredProperties : TStrings
```

StoredProperties determines which property values are saved and restored.

TOvcPersistantState Component

Unlike the previous two State components, TOvcPersistantState does not descend from TOvcAbstractState. It is a simple non-visual component that implements two basic capabilities:

1. The capability to save the published property values for any class that descends from TPersistent. (That covers all components and many classes that you use internal to your applications.)
2. The capability to read and restore saved property values.

Classes like TFont and TStringList can be saved and restored using just two method calls:

6

```
SaveState(TheObjectName, ASectionName)
RestoreState(TheObjectName, ASectionName)
```

In both methods, TheObjectName represents an instance of an object that has TPersistent as its ancestor. ASectionName represents the storage area where the property values are stored.

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

TOvcPersistantState (OvcState)

Properties

❶	About	Storage

Methods

RestoreState	SaveState

Reference Section

RestoreState method

```
procedure RestoreState(
  AnObject : TPersistent; const ASection : string);
```

RestoreState reads and restores all of the previously stored properties for the class specified by AnObject.

ASection defines where the properties are stored and is dependent on the type of store assigned to the Storage property. See“TOvcAbstractState Class” on page 258for additional information about the Section property.

See also: SaveState, Storage

SaveState method

```
procedure SaveState(
  AnObject : TPersistent; const ASection : string);
```

SaveState saves the values of all published properties for the specified object.

ASection defines where the properties are stored and is dependent on the type of store assigned to the Storage property. See “TOvcAbstractState Class” on page 258for additional information about the Section property.

See also: RestoreState, Storage

Storage property

```
property Storage : TOvcAbstractStore
```

Storage defines the component that is used to load and save state information.

When a class is assigned to this property, the methods of the class are used to read and write state information. If Storage is left unassigned, no state information is saved or loaded.

TOvcAbstractStore Class

TOvcAbstractStore defines an abstract ancestor class for the Orpheus store components. It implements the basic functionality needed to support writing and reading property values. Each of the state components (described earlier) require an instance of a descendant of TOvcAbstractStore so that they can read and write state information.

Of the virtual methods defined in this class, descendants need only implement the ReadString and WriteString methods.

Hierarchy

TComponent (VCL)

    ❶ TOvcComponent (OvcBase) 21

        TOvcAbstractStore (OvcFiler)

Properties

❶ About

Methods

Close	ReadBoolean	WriteBoolean
EraseSection	ReadInteger	WriteInteger
Open	ReadString	WriteString

Reference Section

Close — virtual method

```
procedure Close; virtual;
```

Close is called to indicate that all reading and writing is complete.

Close is called after reading and writing. The specifics of this method are implemented in descendants.

See also: Open

EraseSection — virtual method

```
procedure EraseSection(const Section : string); virtual;
```

EraseSection removes all values from the specified section.

The specifics of this method are implemented in descendants.

Open — virtual method

```
procedure Open; virtual;
```

Open prepares the storage location for future operations.

Open is called before reading or writing. The specifics of this method are implemented in descendants.

See also: Close

ReadBoolean — method

```
function ReadBoolean(
  const Section, Item : string DefaultValue : Boolean) : Boolean;
```

ReadBoolean reads a value for Item from the specific section.

If a value does not exist for Item, it is assigned the default value specified by DefaultValue.

See also: WriteBoolean

ReadInteger — method

```
function ReadInteger(
  const Section, Item : string; DefaultValue : Integer) : Integer;
```

↳ ReadInteger reads a value for Item from the specific section.

If a value does not exist for Item, it is assigned the default value specified by DefaultValue.

See also: WriteInteger

ReadString — virtual abstract method

```
function ReadString(
  const Section, Item, DefaultValue : string) : string;
  virtual; abstract;
```

↳ ReadString reads a value for Item from the specific section.

If a value does not exist for Item, it is assigned the default value specified by DefaultValue. The specifics of this method are implemented in descendants.

See also: WriteString

WriteBoolean — method

```
procedure WriteBoolean(
  const Section, Item : string; Value : Boolean);
```

↳ WriteBoolean saves the value for Item in the specified Section.

See also: ReadBoolean

WriteInteger — method

```
procedure WriteInteger(
  const Section, Item : string; Value : Integer);
```

↳ WriteInteger saves the value for Item in the specified Section.

See also: ReadInteger

WriteString virtual abstract method

```
procedure WriteString(
  const Section, Item, Value : string); virtual; abstract;
```

WriteString saves the value for Item in the specified Section.

The specifics of this method are implemented in descendants.

See also: ReadString

TOvcRegistryStore Component

TOvcRegistryStore implements a storage mechanism using the Windows Registry. It adds two properties to those provided by TOvcAbstractStore, KeyName and RegistryRoot, which allow specifying registry-specific information.

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

❷ TOvcAbstractStore (OvcFiler) 268

TOvcRegistryStore (OvcStore)

Properties

❶ About	KeyName	RegistryRoot

Methods

❷ Close	❷ ReadBoolean	❷ WriteBoolean
❷ EraseSection	❷ ReadInteger	❷ WriteInteger
❷ Open	❷ ReadString	❷ WriteString

Reference Section

KeyName property

```
property KeyName : string
```

Default: See below

KeyName is the name of the registry tree where information is stored.

If no value is specified for this property, a default value is constructed using the application's name minus the extension. For example, if the application name were "Project1.exe" the key would be "Software\Project1."

The root value for this key is determined by the RegistryRoot property.

See also: RegistryRoot

RegistryRoot property

```
property RegistryRoot : TOvcRegistryRoot
```

```
TOvcRegistryRoot = (rrCurrentUser, rrLocalMachine);
```

Default: rrCurrentUser

RegistryRoot determines the root registry key.

Possible values and their meaning are:

Value	**Meaning**
rrCurrentUser	HKEY_CURRENT_USER;
rrLocalMachine	HKEY_LOCAL_MACHINE;

See also: KeyName

TOvcIniFileStore Component

TOvcIniFileStore implements a storage mechanism using Windows INI files. It adds two properties to those provide by TOvcAbstractStore, IniFileName and UseExeDir, which allow specifying INI file-specific information.

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

❷ TOvcAbstractStore (OvcFiler) 268

TOvcINIFileStore (OvcStore)

Properties

❶ About	IniFileName	UseExeDir

Methods

❷ Close	❷ ReadBoolean	❷ WriteBoolean
❷ EraseSection	❷ ReadInteger	❷ WriteInteger
❷ Open	❷ ReadString	❷ WriteString

Reference Section

IniFileName property

```
property IniFileName : string
```

Default: See below

IniFileName determines the name of the INI file where property values are stored.

If no value is specified for IniFileName, a default value is constructed using the name of the application. For example, if the name of the application were “Project1.exe”, the file name would be “Project1.ini”.

UseExeDir property

```
property UseExeDir : Boolean
```

Default: False

UseExeDir determines if the INI file is created and maintained in the same directory as the application.

If UseExeDir is True, the INI file is stored in the application’s directory. Otherwise, it is stored in the Windows directory (the default).

TO32XMLStore Component

TO32XMLStore implements a storage mechanism using XML based files. It adds two properties to those provide by TOvcAbstractStore, XMLFileName and UseExeDir, which allow specifying XML file specific information.

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

❷ TOvcAbstractStore (OvcFiler) 268

TO32XMLStore (OvcStore)

Properties

❶ About	XMLFileName	UseExeDir

Methods

❷ Close	❷ ReadBoolean	❷ WriteBoolean
❷ EraseSection	❷ ReadInteger	❷ WriteInteger
❷ Open	❷ ReadString	❷ WriteString

Reference Section

Note: Read methods open the file in read-only, shared mode and then close it after they're finished. Write methods open and lock the file in write mode and close it when they're finished.

UseExeDir property

```
property UseExeDir : Boolean
```

Default: False

UseExeDir determines if the XML file is created and maintained in the same directory as the application.

If UseExeDir is True, the XML file is stored in the application's directory. Otherwise, it is stored in the Windows directory (default).

XMLFileName property

```
property IniFileName : string
```

Default: See below

XMLFileName determines the name of the XML file where property values are stored.

If no value is specified for XMLFileName, a default value is constructed using the name of the application. For example, if the name of the application were "Project1.exe", the file name would be "Project1.xml".

TOvcVirtualStore Component

TOvcVirtualStore implements a purely event-driven store. This component allows you to create special storage capabilities without having to create a descendant of TOvcAbstractStore (as was done for the TOvcRegistryStore and TOvcIniFileStore). You accomplish this by providing event handlers for several of the events published by this component.

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

❷ TOvcAbstractStore (OvcFiler) 268

TOvcVirtualStore (OvcStore)

Properties

❶ About

Methods

❷ Close	❷ ReadBoolean	❷ WriteBoolean
❷ EraseSection	❷ ReadInteger	❷ WriteInteger
❷ Open	❷ ReadString	❷ WriteString

Events

OnCloseStore	OnOpenStore	OnWriteString
OnEraseSection	OnReadString	

OnCloseStore event

```
property OnCloseStore : TNotifyEvent
```

OnCloseStore defines an event handler that is called to close the storage device after all reading and writing is complete.

This event handler is only required if you need to perform some special action to close the storage device. For example, setting the dataset's Active property to False if the storage device were a database file.

See also: OnOpenStore

OnEraseSection event

```
property OnEraseSection : TOvcEraseSectEvent
```

```
TOvcEraseSectEvent = procedure(
  const Section : string) of object;
```

OnEraseSection defines an event handler that is called to remove the Section (and all of its associated values) from the storage device.

Please note that this event handler is required.

OnOpenStore event

```
property OnOpenStore : TNotifyEvent
```

OnOpenStore defines an event handler that is called to open the storage device before any reading or writing is performed.

This event handler is only required if you need to perform some special action to open the storage device. For example, setting the dataset's Active property to True if the storage device were a database file.

See also: OnCloseStore

OnReadString event

```
property OnReadString : TOvcReadStrEvent
```

```
TOvcReadStrEvent = procedure(
  const Section, Item; var Value : string) of object;
```

OnReadString defines an event handler that is called to obtain a string for the specified Section.

Item is the name of the value to read and Value is the value of that item. By default, Value will contain a default value each time the event is fired.

Please note that this event handler is required.

See also: OnWriteString

OnWriteString event

```
property OnWriteString : TOvcWriteStrEvent
```

```
TOvcWriteStrEvent = procedure(
  const Section, Item, Value : string) of object;
```

OnWriteString defines an event handler that is called to save a string in the specified Section.

Item is the name of the value to write and Value is text that is written.

Please note that this event handler is required.

See also: OnReadString

Chapter 7: Tabbed Notebook

The Orpheus tabbed notebook implements properties similar to the standard TNotebook and TTabSet components. It is not a descendant of either of these components, but a completely new component class that provides all the benefits of the VCL components, plus the look and feel of a Microsoft-style tab sheet. Some special features of the tabbed notebook component are its ability to display tabs on any edge as well as the ability to rotate the text on the tabs.

The Orpheus tabbed notebook consists of one or more pages that serve as owners for other components (similar to the TPanel component). Selecting a page's tab displays the associated page (and any components on that page) and hides the previous page. When a page is hidden, you can optionally configure the notebook to destroy the window handles for the page to help conserve Windows resources (the ConserveResources property).

The first click on a tab moves the focus to the first component (in tab order) on that page. A second click on the same tab moves the focus to the tab itself. When the tab has the focus, you can use the arrow keys to navigate among the different tabs. Pressing the <Tab> key moves the focus to the first component on the selected page.

Keyboard selection of pages is also supported. If a page name contains an ampersand '&' character, the page name is displayed on the tab with the letter following the '&' underlined. Pressing <Alt> followed by the underlined letter selects the corresponding notebook page. If more than one page name has the same underlined character, successive entry of the <Alt> <letter> key sequence will result in the selection of the next page in a cyclic fashion (i.e., TabA, TabB, TabC, TabA, TabB, etc.).

Individual notebook pages can be disabled or hidden. The name for a disabled page is drawn using grayed-out text. A disabled page cannot be selected or made the active page until it is enabled. Hidden notebook pages are the same as disabled notebook pages except that the tab corresponding to a hidden page is not displayed. Hidden pages are a good way to hide unlicensed functionality or supervisor level activities.

TOvcNotebook Component

The TOvcNotebook component implements tabbed notebook pages that can be displayed on any edge of the notebook as shown in Figures 7.1 through 7.4.

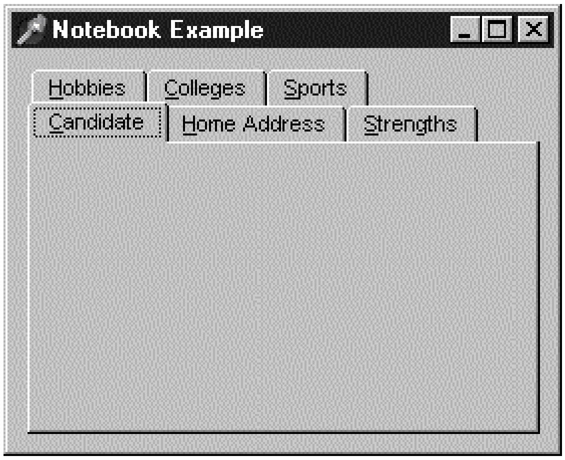


Figure 7.1: Tabs on the top

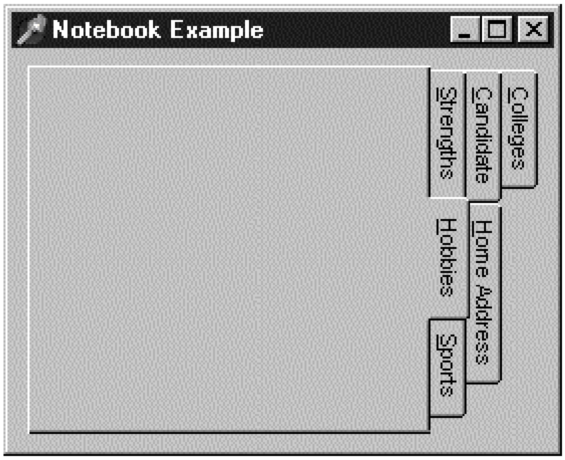


Figure 7.2: Tabs on the right

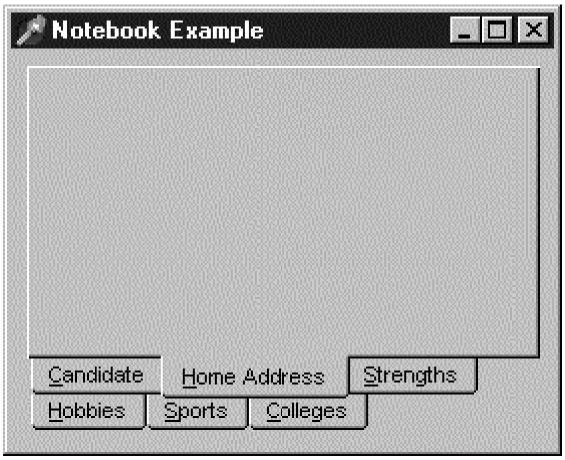


Figure 7.3: Tabs on the bottom

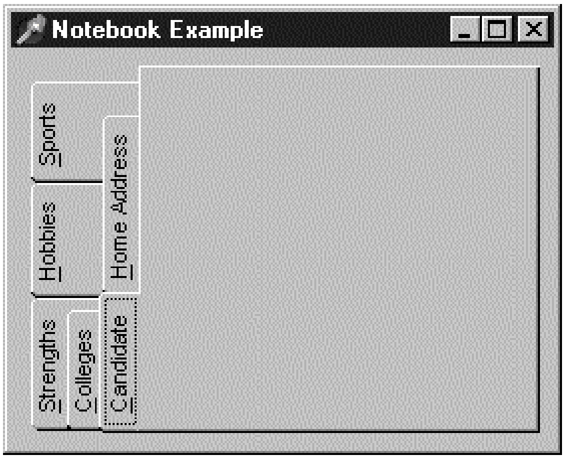


Figure 7.4: Tabs on the left

A notebook can have any number of tabs and any number of tab rows. By default, as tabs are added or removed (via the PageCollection property), the number of tab rows is automatically adjusted so that each row contains as many tabs as will fit based on the current tab caption. If the notebook width increases, tabs are re-arranged to minimize the number of rows.

Scrolling the tabs

If there are more tabs than can fit on the number of rows specified in TabRowCount, then navigating through the tabs using an <Alt> <letter> key sequence, the <Tab> key, or the arrow keys will cause the tab area to scroll to the visible tab.

If ShowScrollSpinner is True then a spinner button appears in the tab area. The spinner is used for scrolling among the tabs horizontally or vertically depending on the value of TabOrientation. Setting TabOrientation to “toTop”, TabRowCount to ‘1’, and ScrollSpinnerLocation to “slRightBottom” will cause the OvcNotebook to emulate the Delphi component palette’s tab area.

Notebook window handles

In some cases you may need to insure that all the notebook pages have window handles. Use the ForcePageHandles method to force each notebook page to create its window handle. This in turn, will force each control on the notebook pages to create their window handles.

Notebook commands

The commands in Table 7.1 are available in the Orpheus notebook. The commands and key assignments described here are available when you use the “Default” or “WordStar” command tables (several command tables can be used at the same time). See “TOvcCommandProcessor Class” on page 50 for information on how to customize the command tables.

Table 7.1: *Notebook commands*

Command	Default	WordStar	Description
ccDown	<Down>	<CtrlX>	Move the selection to the next tab.
ccLeft	<Left>	<CtrlS>	Move the selection to the previous tab.
ccRight	<Right>	<CtrlD>	Move the selection to the next tab.
ccUp	<Up>	<CtrlE>	Move the selection to the previous tab.

Example

This example shows how to construct a simple application that uses a TOvcNotebook component to create a multi-page entry form.

Create a new project and add a TOvcNotebook component. Select the Pages property from the Object Inspector and invoke the notebook page property editor. From the notebook page property editor, you can add new pages, change page information, and re-order the pages. Add or change the notebook pages to get a feel for using the property editor.

Add a few components to each of the notebook pages by selecting the component and then clicking on a notebook page (it doesn't matter which components you choose for this example).

Set the Enabled property for a page to False. This disables that notebook page and causes the tab text to be drawn using shadowed text. While designing, you can select and use a disabled page just like an enabled page. At run-time, disabled pages cannot be selected.

Run the project and experiment with the generated program. Notice that the disabled page tab cannot be selected.

When a notebook tab is first selected, the notebook attempts to give the focus to the first control on that page that has its TabStop property set to True. If there are no components (or none have their TabStop property set to True), the focus is placed on the page's tab. Clicking on the tab of the active page also gives the focus to the tab.

Once a tab has the focus, you can move from page to page using the arrow keys. Pressing the <Tab> key moves the focus from the tab to the first component on that page.

Hierarchy

TCustomControl (VCL)

    ❶ TOvcCustomControl (OvcBase) 16

        ❷ TOvcCustomControlEx (OvcBase) 19

            TOvcNotebook (OvcNbk)

Properties

❶	About	❶	LabelInfo	ShowScrollSpinner
	ActiveTabFont		OldStyle	TabAutoHeight
	AllowTabRowsRearrange		PageCount	TabHeight
❶	AttachedLabel		Pages	TabOrientation
	ClientHeight		PageCollection	TabRowCount
	ClientWidth		PageIndex	TabTextOrientation
	ConserveResources		PageUsesTabColor	TabUseDefaultColor
❷	Controller		ScrollSpinnerLocation	TextShadowColor
	DefaultPageIndex		ShadowColor	UseActivetabFont
	HighlightColor		ShadowedText	

Methods

BeginUpdate	InsertPage	PageNameToIndex
DeletePage	InvalidateTab	PrevPage
EndUpdate	IsValid	PrevValidIndex
ForcePageHandles	NextPage	TabsInRow
GetTabRect	NextValidIndex	

Events

❶	AfterEnter		OnMouseOverTab	OnPageChanged
❶	AfterExit	❶	OnMouseWheel	OnTabClick
	OnDrawTab		OnPageChange	OnUserCommand

Reference Section

ActiveTabFont — property

```
property ActiveTabFont : TFont
```

Default: Arial

↳ Determines the font used for the active tab.

This property can be used to cause the font characteristics of the active tab to be different than inactive tabs.

AllowTabRowsRearrange — property

```
property AllowTabRowsRearrange : Boolean
```

Default: true

↳ Determines whether the tab rows are allowed to re-arrange themselves to place the active tab on the bottom row.

The standard behavior is to allow the tab rows to re-arrange thus placing the active tab's row on the bottom. Some users find this annoying and would prefer that the tabs maintain proper order whether they are selected or not.

Setting this property to False forces the tab rows to maintain proper order. If by chance, the selected tab is on the bottom row (or the row closest to the tab page, depending on TabOrientation) then the appearance is unchanged. However, if the selected tab is in another row, then it will appear among the rest of the tabs as a raised button.

BeginUpdate — method

```
procedure BeginUpdate;
```

↳ Prevents the screen from being repainted while changes are made to the notebook pages.

After BeginUpdate is called, the notebook's window is not updated. EndUpdate must be called to re-enable screen repainting. It is probably best to use a try-finally block to ensure that EndUpdate is called.

The following example uses a try-finally block to ensure that EndUpdate is called:

```
BeginUpdate;
try
  {do something with the notebook pages}
finally
  EndUpdate;
end;
```

See also: EndUpdate

ClientHeight run-time, read-only property

```
property ClientHeight : Integer
```

↳ Returns the height of the notebook page area in pixels.

ClientWidth run-time, read-only property

```
property ClientWidth : Integer
```

↳ ClientWidth returns the width of the notebook page area in pixels.

ConserveResources property

```
property ConserveResources : Boolean
```

Default: False

↳ Determines if the window handles of inactive notebook pages are freed.

The ConserveResources property determines if the window handle associated with a notebook page is freed when the page is not the active page. Freeing the page window handle also frees the window handles of all components on that page.

If ConserveResources is set to True, Windows resources are not used for a notebook page until that page is made the active page. Selecting a different page as the active page frees the window handle of the current active page before making the newly selected page active.

Freeing window handles has no effect on non-visual components or components that have no window handle.

Use of this property should not have any negative effects on the operation of the notebook or its components as long as the components are designed to preserve their data across window re-creation. All the standard Orpheus components support this, but some of the standard VCL components (most notably the ComboBox) do not preserve data across window re-creation in all cases.

DefaultPageIndex property

```
property DefaultPageIndex : Integer
```

Default: 0

↳ Determines which notebook page is displayed when the notebook is first displayed.

DeletePage method

```
procedure DeletePage(Index : Integer);
```

↳ Deletes an existing notebook page.

DeletePage removes the notebook page indicated by Index. Deleting a notebook page also deletes all components on that page. Valid values for Index are 0 to the current number of notebook pages - 1.

EndUpdate method

```
procedure EndUpdate;
```

↳ Re-enables screen repainting.

If BeginUpdate is called, EndUpdate must be called to re-enable screen repainting. See BeginUpdate for more information and an example.

See also: BeginUpdate

ForcePageHandles method

```
function ForcePageHandles;
```

↳ Causes the window handles for all notebook pages to be created.

Use the ForcePageHandles method to force each notebook page to create its window handle. This, in turn, will force each control on the notebook pages to create their window handles.

GetTabRect method

```
function GetTabRect(Index : Integer) : TRect;
```

↳ Returns the bounding rectangle for the specified tab.

Index is the index of one of the notebook pages. The function result is a TRect structure that bounds the specified tab. It is intended to be used when using the OnDrawTab event, when drawing to the tab area. Valid values for Index are 0 to the current number of notebook pages -1.

HighlightColor property

```
property HighlightColor : TColor
```

Default: clBtnHighlight

↳ Determines the color used to display the highlighted edge of the notebook and tabs.

The highlighted edge is the top and left border of the notebook and tabs.

InsertPage method

```
function InsertPage(
  const Name : string; Index : Integer) : TOvcNotebookPage;
```

↳ Inserts a new notebook page into the notebook.

InsertPage creates a notebook page with Name as its name and inserts it into the notebook's collection of notebook pages at the position indicated by Index. Valid values for Index are 0 to the current number of notebook pages.

InvalidateTab method

```
procedure InvalidateTab(Index : Integer);
```

↳ Marks the specified tabs as requiring painting.

Use this method when you want to force a specific notebook tab to be redrawn. Valid values for Index are 0 to the current number of notebook pages - 1.

IsValid method

```
function IsValid(Index : Integer) : Boolean;
```

↳ Tells whether the specified page index is valid.

IsValid returns True if Index is a valid page index. The valid range for page indexes is 0 to the number of notebook pages - 1. An index that is in the valid range is also invalid if the page is disabled or hidden.

The following example tests the validity of an index before attempting to use it to change the active page:

```
  if IsValid(SomeIndex) then
    PageIndex := SomeIndex
  else
    {do something here}
```

The following example uses exceptions to perform the same operation:

```
try
  PageIndex := SomeIndex;
except
  on EInvalidPageIndex do
    {do something here and/or re-raise the exception}
end;
```

See also: PageIndex

NextPage method

```
procedure NextPage;
```

↳ Causes the next valid notebook page to become the active notebook page.

See also: PrevPage

NextValidIndex method

```
function NextValidIndex(Index : Integer) : Integer;
```

↳ Returns the index of the next visible and enabled notebook page.

The valid range for Index is 0 to the number of notebook pages -1.

OldStyle property

```
property OldStyle : Boolean
```

Default: False

↳ Determines if the notebook tabs are drawn using the style used in older version of Orpheus.

The major difference between the current drawing method and the older style is that the tab widths are not forced to occupy the entire width of the notebook.

OnDrawTab event

```
property OnDrawTab : TDrawTabEvent
```

```
TDrawTabEvent = procedure(Sender : TObject; Index : Integer;
  const Title : string; R : TRect; Enabled,
  Active : Boolean) of object;
```

Defines an event handler that is fired when the tab text area needs to be drawn.

Index is the 0-based tab index number. Title is the text associated with this tab (the title normally painted in the tab text area). R is the coordinates of the area that requires painting. Enabled indicates the tab's enabled status, and Active is True to indicate that this is a normal tab and requires painting or False to indicate that this tab is a filler tab and should not normally be painted.

The EXODT.DPR project demonstrates using this event to allow drawing a bitmap on the active tab and shows how to draw horizontal text on vertical tabs.

OnMouseOverTab event

```
property OnMouseOverTab : TMouseOverTabEvent
```

```
TMouseOverTabEvent = procedure(
  Sender : TObject; Index : Integer) of object;
```

Defines an event handler that is called when the mouse moves over one of the notebook's tabs.

This event is generated for each mouse movement over a notebook tab. If the mouse is moved within a tab, a continuous stream of events is generated. Index is the page index of the tab. Sender is the instance of the notebook.

The following example shows how to detect mouse movement over one of the notebook tabs. In this example, when the mouse is moved over the tab with an index value of 1, the computer generates a beep.

```
procedure TForm1.OvcNotebook1MouseOverTab(
  Sender : TObject; Index : Integer);
begin
  if Index = 1 then
    MessageBeep(0);
end;
...
OnMouseOverTab := Form1.OvcNotebook1MouseOverTab;
```

OnPageChange event

```
property OnPageChange : TPageChangeEvent
```

```
TPageChangeEvent = procedure(Sender : TObject;
  Index : Integer; var AllowChange : Boolean) of object;
```

Defines an event handler that is called when a notebook page is selected.

The method assigned to the OnPageChange event is called to provide notification of a notebook page change. Index is the index value of the new notebook page. You can disallow the page change by setting AllowChange to False.

The following example shows how to reject the selection of a page. In this example, if the mouse is clicked on the tab with a page index of 5, AllowChange is set to False. This informs the notebook object not to change the active tab.

```
  procedure TForm1.OvcNotebook1PageChange(
    Sender : TObject; Index : Integer; var AllowChange : Boolean);
  begin
    if Index = 5 then
      AllowChange := False;
  end;
  ...
  OnPageChange := Form1.OvcNotebook1PageChange;
```

See also: OnPageChanged

OnPageChanged event

```
property OnPageChanged : TPageChangedEvent
```

```
TPageChangedEvent = procedure(
  Sender : TObject; Index : Integer) of object;
```

Defines an event handler that is called after a new notebook page is selected.

The method assigned to the OnPageChanged event is called after a new notebook page is selected. Index is the index number of the new page.

The following example updates a TLabel component with the name of the new notebook page:

```
procedure TForm1.OvcNotebook1PageChanged(
  Sender : TObject; Index : Integer);
begin
  Label1.Caption := OvcNotebook1.Pages[Index];
end;
...
OnPageChanged := Form1.OvcNotebook1PageChanged;
```

See also: OnPageChange, Page

OnTabClick event

```
property OnTabClick : TTabClickEvent
```

```
TTabClickEvent = procedure(
  Sender : TObject; Index : Integer) of object;
```

7

↳ Defines an event handler that is called when a mouse click is made on a notebook tab.

The method assigned to the OnClick event is called to provide notification of a mouse click on a notebook tab. Index is the page index of the tab and can be the same as the active page index.

The following example updates a TLabel component with the page name for the tab that was clicked on:

```
procedure TForm1.OvcNotebook1TabClick(
  Sender : TObject; Index : Integer);
begin
  Label2.Caption := OvcNotebook1.Page[Index].Caption;
end;
```

See also: OnMouseOverTab, Page

OnUserCommand event

```
property OnUserCommand : TUserCommandEvent
```

```
TUserCommandEvent = procedure(
  Sender : TObject; Command : Word) of object;
```

↳ Defines an event handler that is called when a user-defined command is entered.

The method assigned to the OnUserCommand event is called when you enter the key sequence corresponding to one of the user-defined commands (ccUser1, ccUser2, etc.).

For example, suppose you add the <F9> key sequence to one of the active command tables and assign it to the ccUser1 command. When you press <F9>, the method assigned to the OnUserCommand event is called and ccUser1 is passed in Command.

See “TOvcCommandProcessor Class” on page 50 for additional information about user-defined commands and command tables.

PageCollection property

```
property PageCollection : TOvcCollection
```

↳ Provides access to the internal TOvcPageCollection objects.

PageCollection also provides a property editor that allows adding, moving, deleting, and setting the properties of the notebook pages.

See “TOvcCollection Class” on page 39 for additional information about Orpheus collections and the collection editor.

See also: Pages

PageCount run-time, read-only property

```
property PageCount : Integer
```

↳ Returns the number of notebook pages.

PageIndex property

```
property PageIndex : Integer
```

↳ The index of the active notebook page.

PageIndex allows access to the index of the active notebook page and allows you to change the active notebook page. If you set PageIndex to an invalid page index, an EInvalidPageIndex exception is raised. The valid range for page indexes is 0 to the number of notebook pages - 1. An index that is in the valid range is invalid at run-time if the page is disabled or hidden.

See also: IsValid, Page

PageNameToIndex method

```
function PageNameToIndex(const Name : string) : Integer;
```

7

↳ Returns the index of the specified page.

Given a notebook page name, PageNameToIndex returns its corresponding page index. If Name does not match any of the notebook page names, -1 is returned. The text comparison is not case sensitive.

Pages run-time property

```
property Pages[Index : Integer] : TOvcTabPage
```

↳ An indexed property that provides access to the individual notebook page objects.

Each notebook page is an instance of a TOvcTabPage and has the following properties that can be set either at run time or at design time using the collection editor provided by the PageCollection property:

Properties	**Description**
Caption	Determines the text displayed in the tab area.
Enabled	Determines if the tab can be selected.
TabColor	Determines the color of the tab area.
TextColor	Determines the color used when displaying the caption.
Visible	Determines if the tab and page are visible (true) or hidden (false).

See also: PageCollection.

PageUsesTabColor property

```
property PageUsesTabColor : Boolean
```

Default: True

↳ Determines if the notebook page area is drawn using the color of the associated tab.

PrevPage method

```
procedure PrevPage;
```

↳ Causes the previous valid notebook page to become the active notebook page.

See also: NextPage

PrevValidIndex method

```
function PrevValidIndex(Index : Integer) : Integer;
```

↳ Returns the index of the first enabled and visible notebook page prior to the specified Index.

Valid values of Index are from 0 to the number of notebook pages - 1.

ScrollSpinnerLocation property

```
property ScrollSpinnerLocation : TSpinnerLocation;
```

```
TSpinnerLocation = (
  slLeftBottom, slLeftTop, slRightBottom, slRightTop);
```

Default: slRightBottom

↳ Determines the location of the tab area where the scroll spinner will appear.

See also: ShowScrollSpinner

ShadowColor property

```
property ShadowColor : TColor
```

Default: clBtnShadow

↳ The color of the shadowed edge of the notebook and tabs.

The shadowed edge is the bottom and right border of the notebook and tabs.

See also: HighlightColor

ShadowedText property

```
property ShadowedText : Boolean
```

Default: False

↳ Determines if tab text is drawn using shadows.

If ShadowedText is True, the page names displayed on the tabs are drawn using shadowed text. Shadowing gives the text a 3D Look.

See also: TextShadowColor

ShowScrollSpinner property

```
property ShowScrollSpinner : Boolean
```

Default: True

↳ Determines whether a spinner button will be displayed in the event there are more tabs than can be displayed in the number of rows specified by TabRowCount.

See also: TabRowCount, ScrollSpinnerLocation

TabAutoHeight property

```
property TabAutoHeight : Boolean
```

Default: True

↳ Determines if the tab height is adjusted automatically when the tab text font changes.

If TabAutoHeight is True, the height of the notebook tabs are automatically adjusted based on the height of the current font.

See also: TabHeight

TabHeight property

```
property TabHeight : LongInt
```

Default: 20 (if TabAutoHeight is False)

↳ The height of the notebook tabs.

Assigning a value (in pixels) to TabHeight adjusts the height of all the notebook tabs and sets TabAutoHeight to False.

See also: TabAutoHeight

TabOrientation property

```
property TabOrientation : TTabOrientation
```

```
TTabOrientation = (toTop, toRight, toBottom, toLeft);
```

Default: toTop

↳ Determines where the tabs are drawn.

TabOrientation determines whether the notebook is drawn with tabs at the top, right, bottom or left edges of the notebook. If TabOrientation is set to toRight or toLeft and the font currently being used is not a True Type font, the font is automatically changed to Arial. The text drawn on the tabs must be rotated and only True Type fonts can be rotated.

See also: ActiveTabFont, TabTextOrientation, UseActiveTabFont

TabRowCount property

```
property TabRowCount : Integer
```

Default: 0

↳ Determines the number of notebook tab rows.

Setting TabRowCount to a value other than 0 forces the notebook to draw its tabs using the specified number of rows. Depending on the size of the notebook and/or form, and the value of the Align property, this may result in changes in the tab width. Setting TabRowCount to 0, tells the notebook to calculate the optimum number of rows based on notebook size.

See also: ShowScrollSpinner

TabsInRow method

```
function TabsInRow(Row : Integer) : Integer;
```

↳ Returns the number of tabs in the specified Row.

TabTextOrientation property

```
property TabTextOrientation : TTabTextOrientation
```

```
TTabTextOrientation = (
  ttoHorizonatl, ttoVerticalRight, ttoVerticalLeft);
```

Default: ttoHorizontal

↳ Determines how the text is oriented in the tab.

TabTextOrientation determines how the text in the tabs appear in relation to the tab's associated page area. The default setting is ttoHorizontal which is normal orientation. ttoVerticalRight displays the text vertically with the bottom of the text along the right side of the tab. ttoVerticalLeft displays the text vertically with the bottom of the text along the left side of the tab.

If TabTextOrientation is set to ttoVerticalRight or ttoVerticalLeft and the font currently being used is not a True Type font, then the font will be changed to Arial. The text drawn on the tabs must be rotated and only True Type fonts can be rotated. If you don't want to use Arial in your tabs, then make sure you select another True Type font prior to changing the value of TabTextOrientation.

See also: ActiveTabFont, TabOrientation, UseActiveTabFont

TabUseDefaultColor property

```
property TabUseDefaultColor : Boolean
```

Default: True

↳ Determines if the notebook tabs use the color value assigned to the notebook (True) or if the notebook tabs use individual color settings (False).

TextShadowColor property

```
property TextShadowColor : TColor
```

Default: clButtonShadow

↳ The color used to draw the text shadow on notebook tabs.

If ShadowedText is False, TextShadowColor is not used.

See also: ShadowedText

UseActiveTabFont property

```
property UseActiveTabFont : Boolean
```

Default: False

Determines if the active notebook tab uses the font defined as the ActiveTabFont or if it uses the font defined by the tab page.

See also: ActiveTabFont

7

Chapter 8: Most-Recently-Used Menu List

Orpheus provides a flexible MRU component for automatically adding a list of “most recently used” documents to your File menu or elsewhere in your application. The most common use for an MRU list is to maintain a list of recently used files in an application. Use of TOvcMenuMRU is not, however, limited to file names. Any list of strings can be maintained using TOvcMenuMRU. You have the option to add menu accelerators to the items in the MRU list, to show the full path of the file or just the file name, and to have the path and file name shortened if it exceeds a specified width.

TOvcMenuMRU supports two styles of MRU list. The first style is a traditional MRU list like you see in many Windows programs. This type of MRU list is a file name list that is typically displayed above the Exit menu item on the File menu. The second type of file MRU list is the Delphi/C++Builder-style MRU list. This type of MRU list is a split list. The Delphi/C++Builder Reopen menu item displays the project files in the top portion of the list and other files (.PAS, .CPP, .TXT, .RC, etc.) in the bottom portion of the list. By default, TOvcMenuMRU assumes that you are using a traditional MRU list but you can easily change to a Borland-style MRU list if you prefer.

TOvcMenuMRU Component

The TOvcMenuMRU component implements a most-recent-used list (MRU). Strings are added to the list in first-in, first-out fashion with the last string added at the top of the list. If the number of strings in the list exceeds the maximum number of items allowed, the last item in the list is dropped off to make room for the newest item.

The OnClick event is generated when an MRU menu item is clicked. The text value of the item clicked is passed to your OnClick handler. You can use the string to reopen the file that was clicked or, for other types of MRU lists, to perform some action based on the type of list you are implementing. OnClick is generated both for mouse clicks and keyboard menu selection.

TOvcMenuMRU does not provide built-in persistence. If you want the MRU list to be persistent between instances of your application, use a TOvcComponentState component in combination with one of the store components (TOvcIniFileStore, TOvcRegistryStore, TO32XMLStore, or TOvcVirtualStore). To make the MRU list persistent you only need to save the contents of TOvcMenuMRU's Items property. See the Orpheus State components on page 257 for more information on the persistent state components.

Hierarchy

TComponent (VCL)

  TOvcComponent (OvcBase). 21

    TOvcMenuMRU (OvcMRU)

Properties

AddPosition	Hint	Options
AnchorItem	Items	Style
Count	MaxItems	Visible
Enabled	MaxMenuWidth	
GroupIndex	MenuItem	

Methods

Add	Clear
AddSplit	Remove

Events

OnClick

Reference Section

Add method

```
procedure Add(const Value : string);
```

Places a string at the top of the MRU list.

Items are always added to the top of the list. If the number of items in the list exceeds MaxItems, the last (oldest) item in the list is removed. If the item being added already exists in the MRU list, it is moved to the top of the list. To add an item to a split menu, call AddSplit instead of Add. Calling Add will raise an exception if Style is set to msSplit. The following example adds a file name to an MRU list:

```
  if OpenDialog1.Execute then
    OvcMenuMRU1.Add(OpenDialog1.FileName);
```

See also: AddSplit, MaxItems, Remove, Style

AddPosition property

8

```
property Position : TOvcMRUAddPosition
```

```
TOvcMRUAddPosition = (apAnchor, apTop, apBottom);
```

Default: apAnchor

The position within the parent menu where the MRU list will appear.

Set AddPosition to determine where MRU items will appear within the parent menu. The parent menu is set via the MenuItem property. The following list includes the possible values and describes each:

Value	**Meaning**
apAnchor	The MRU list will be placed above the menu item specified by AnchorItem.
apTop	The MRU list will be placed at the top of the menu.
apBottom	The MRU list will be placed at the bottom of the menu.

For traditional MRU lists you will typically set AddPosition to apAnchor (the default). For pop-up MRU lists (MRU lists which are submenus of a given menu item) set AddPosition to either apTop or apBottom. For split MRU lists AddPosition is ignored.

See also: Add, AddSplit, AnchorItem, MenuItem

AddSplit method

```
procedure AddSplit(
  const Value : string; Position : TOvcMRUAddPosition);
```

```
TOvcMRUAddPosition = (apAnchor, apTop, apBottom);
```

Places a string at either the top or the bottom of a split MRU list.

Call AddSplit to add a string to the top or bottom portion of a split MRU list. Value is the string to be added. Position is the split menu position where the item will be added. Valid values for Position are apTop and apBottom.

An EMenuMRUError exception is raised if you call AddSplit with a position parameter of apAnchor, if the Style is msNormal or if no menu item is assigned.

The following example adds Delphi project files to the top of a split MRU list, and Delphi PAS files to the bottom:

```
  if OpenDialog1.Execute then begin
    if UpperCase(ExtractFileExt(OpenDialog1.FileName)) =
      '.DPR' then
      DelphiList.AddSplit(OpenDialog1.FileName, maTop);
    if UpperCase(ExtractFileExt(OpenDialog1.FileName)) =
      '.PAS' then
      DelphiList.AddSplit(OpenDialog1.FileName, maBottom);
  end;
```

See also: Add, MaxItems, Remove, Style

AnchorItem property

```
property AnchorItem : TMenuItem
```

The menu item above which the MRU list will be inserted.

Set AnchorItem to the item on a menu above which you want the MRU list to appear. By default TOvcMenuMRU assumes that you want the MRU list to appear on the File menu above the Exit menu item. The MRU list will be added to the top of the parent menu if AnchorItem is not assigned. AnchorItem is ignored when Style is set to msSplit.

See also: Add, AddPosition, MenuItem, Style

Clear method

```
procedure Clear;
```

↳ Removes all items from the MRU list.

Call Clear to empty the MRU list. Clear will empty the list and remove all entries from the parent menu (specified by MenuItem). You may also call Items.Clear if you prefer, although there is rarely any reason to do so.

See also: Add, AddSplit, Items, Remove

Count property

```
property Count : Integer;
```

↳ The number of items in the MRU list.

See also: Items

Enabled property

```
property Enabled : Boolean
```

Default: True

↳ Determines if the MRU items are enabled if the MRU list is associated with a menu.

If the MenuItem property is not assigned, Enabled will be ignored.

See also: MenuItem

GroupIndex property

```
property GroupIndex : Integer
```

Default: 0

↳ Controls how menus are merged if the MRU list is associated with a menu.

If the MenuItem property is not assigned, GroupIndex will be ignored.

See also: MenuItem

Hint property

```
property Hint : string
```

↳ The hint text that is sent to the Application object's OnHint event if the MRU list is associated with a menu.

If the MenuItem property is not assigned, Hint will be ignored.

See also: MenuItem

Items property

```
property Items : TStrings
```

↳ The list of strings in the MRU list.

Items is an instance of the VCL's TStringList class. Items is published primarily to facilitate persistence using Orpheus' persistent state components. While it is possible to add strings to the list at design time, it is not expected that you would ever find the need to do so.

To add items to the MRU list, call the Add or AddSplit methods. To clear the list, call the Clear method. To determine the number of items in the list use the Count property.

All of the usual properties and methods of TStringList apply to the Items property. As such, it is possible to call Delete, Insert, Clear, etc. to modify the string list directly. While allowable, you should never need to call these methods directly. In fact, you should avoid directly manipulating the string list whenever possible. In particular, you should never call Items.Add or Items.Insert from your applications. Always use the Add or AddSplit methods of TOvcMenuMRU instead. Calling TStringList's Delete and Clear methods will likely result in correct behavior of the MRU list, but that situation is best avoided if possible.

See also: Add, AddSplit, Clear, Count, Remove

MaxItems property

```
property MaxItems : Integer
```

Default: 4

↳ The maximum number of items in the MRU list.

Set MaxItems to the maximum number of items the MRU list should display. Items are added to the list in first-in, first out fashion. If the number of strings in the list exceeds MaxItems, the last item in the list is dropped off to make room for the newest item.

For split MRU lists, the MaxItems property applies to both the top and the bottom portions of the list individually. If, for example, you set MaxItems to 4 then both the top portion of the list and the bottom portion can each contain up to four items (giving a total of up to eight items in the list as a whole). It is not possible to set MaxItems for the top and bottom portions of a split MRU list individually.

See also: Add, AddSplit, Count, Style

MaxMenuWidth property

```
property MaxMenuWidth : Integer
```

Default: 0

Determines the maximum display width of the MRU list text, in pixels.

Set MaxMenuWidth to force MRU items to conform to a specific menu width. TOvcMenuMRU will truncate the display of strings to fit the specified width. When MaxMenuWidth is 0, no truncating of the MRU list text is performed.

The following table shows the effect of truncating the file name, “C:\program files\myfiles\my long file.doc” given a variety of values for MaxMenuWidth:

MaxMenuWidth	**Result**
200	`C:\program files\myfiles\my long file.doc`
175	`C:\program files\...\my long file.doc`
125	`C:\...\my long file.doc`
100	`C:\...\my long file.d...`
75	`C:\...\my long...`
0	`C:\program files\myfiles\my long file.doc`

This example assumes that the Options property does not include the moStripPath and moAddAccelerators options. Note that MaxMenuWidth only affects the display of the menu text. It has no bearing on the actual string stored in Items.

See also: Items, Options

MenuItem property

```
property MenuItem : TMenuItem
```

↳ The menu item that will contain the MRU list.

Set MenuItem to the menu item within which the MRU list will be placed. The menu specified by MenuItem will become the parent menu of the MRU list. For traditional MRU menus this will normally be a top-level menu item. For example, for a traditional MRU list, set MenuItem to the File menu (File1 for a default VCL menu). For a Delphi/C++Builder "Reopen" style MRU list (a popup menu off the top-level menu), set MenuItem to the menu item within the top-level menu that will contain the MRU list. The MRU items will be inserted into the menu based on the value of the AddPosition property.

By default TOvcMenuMRU assumes a traditional MRU list. AnchorItem and MenuItem will be automatically assigned when you place a TOvcMenuMRU component on your form, provided that the form already contains a MainMenu component, that the menu component has a top-level File menu, and that the File menu contains an Exit menu item.

See also: AddPosition, AnchorItem, Style

OnClick event

```
property OnClick : TOvcMRUClickEvent

TOvcClickEvent = procedure (
  Sender : TObject; ItemText : string;
  var Action : TOvcMRUClickAction)of object;

TOvcMRUClickAction = (caMoveToTop, caRemove, caNoAction);
```

↳ Defines an event handler that will be fired when an item in the MRU list is clicked.

Provide an event handler for the OnClick event in order to respond to clicks on MRU menu items. Both mouse clicks and menu selection via the keyboard result in the OnClick event being generated. Write code in your OnClick handler to perform some user action based on the text of the item that was clicked (to open a file, for example). ItemText contains the text string corresponding to the MRU item that was clicked. Action controls what happens to the clicked item in the MRU list. The default action is caMoveToTop.

The following table shows the possible values for Action and their effect on the MRU list:

TOvcMRUClickAction	**Result**
caMoveToTop	The clicked item will be moved to the top of the MRU list.
caRemove	The clicked item will be removed from the MRU list.
caNoAction	The clicked item will retain its current location in the list.

For file name MRU lists, most Windows applications either move the clicked item to the top of the MRU list or remove the item altogether. If you choose to remove the clicked item from the MRU list you should add it back again when the current file is closed.

The following example reopens a file in an OnClick event handler:

```
procedure TForm1.MRUMenuClick(
  Sender : TObject; const ItemText : string;
  var Action : TOvcMRUClickAction);
begin
  Memo1.Lines.LoadFromFile(ItemText);
end;
```

Options property

```
property Options : TOvcMRUOptions
```

```
TOvcMRUOption = (
  moAddAccelerators, moStripPath, moAddSeparator);
```

```
TOvcMRUOptions = set of TOvcMRUOption;
```

Default: moAddAccelerators, moAddSeparator

Determines how the MRU items are displayed on the menu.

The following is a list of the possible Options property values and a description of each:

TOvcMRUOptions	**Result**
moAddAccelerators	An alpha-numeric accelerator character is added to the beginning of each item in the MRU list. The first nine accelerators are numbered from 1 to 9. If more than nine items are in the MRU list, subsequent items are lettered from A to Z.
moStripPath	Only the file name is shown in the MRU list.
moAddSeparator	A menu separator is added beneath the MRU list.

When using accelerators, bear in mind that theaccelerator values may conflict with other accelerators on your menu. While this is rarely a problem for numeric accelerators, it may be more problematic when alphabetic accelerators are used (when there are more than nine items in an MRU list). For most applications, an MRU list with a maximum of nine items is sufficient.

See also: Add, AddSplit, Items, MaxMenuWidth

Remove method

```
procedure Remove(const Value : string);
```

Removes an item from the MRU list.

Value is the path and file name of the item to remove.

See also: Add, AddSplit, Items

Style property

```
property Style : TOvcMRUStyle
```

```
TOvcMRUStyle = (msNormal, msSplit);
```

Default: msNormal

Determines the MRU list style.

When implementing a traditional MRU list, set Style to msNormal. For a Delphi/C++Builder- type MRU list, set Style to msSplit. When Style is msNormal, use the Add method to add items to the list. When Style is set to msSplit, use the AddSplit method to add items to either the top or the bottom portion of the split list.

See also: Add, AddSplit

Visible property

```
property Visible : Boolean
```

Default: True

Determines if the MRU list is visible when the list is associated with a menu.

If the MenuItem property is not assigned, Visible will be ignored.

See also: MenuItem

Chapter 9: Data Entry Basics

In addition to the Orpheus entry field components, Orpheus 4 introduces a whole new way of validating data entry using a specialized component called a Validator. In the future, Orpheus will move away from dependence on the OvcController and in the direction of Validators and ValidatorPools. For more information on the the power of the new Validators see chapter 13.

This chapter introduces some basic concepts that apply to several of the older style Orpheus components. A thorough understanding of the picture mask and validation concepts discussed in this chapter will enable you to use the Orpheus data entry components to their fullest.

Probably the most important basic concept is that of the *picture mask*. A picture mask is a string of characters that is used to describe both the legal contents of a field and the way that the field is displayed. Picture masks are discussed in detail in “Picture Masks” on page 316.

Orpheus provides a method that allows you to change a field’s picture mask at run time, primarily to allow you to deal effectively with international differences in date, time, and currency formats. This method is discussed in “Changing a field’s picture mask” on page 328.

The field validation scheme in Orpheus is very flexible, allowing you to choose how fields are validated, whether to validate immediately or wait, what happens when a field is invalid, and whether to perform special, custom validation. This is discussed in “Field Validation” on page 331.

Orpheus also provides for uninitialized fields, required fields, read-only fields, and password fields. They are all discussed in “Special Field Types” on page 336.

Picture Masks

In Orpheus, as in most data entry packages, picture masks serve two purposes. On one hand, they are used to control the formatting of data. You can use them to ask, for example, that a date be displayed in the “mm/dd/yyyy” format, meaning that the month should be displayed first, followed by the day and then the year, with a slash separating each component of the date from the other: “12/31/1999.” In this sense, the picture mask controls the process by which it is converted from its native type (integer, real, extended, date, etc.) into a formatted string.

Picture masks also serve a second and equally important purpose, that of limiting the range of characters that can be entered at a given position in a string. In this sense, a picture mask provides the means by which a field can perform input validation on a character-by-character basis. More stringent tests can be performed later, when the user tries to leave the field, but there is no point in performing them at all if the user has entered “ab/cd/efgh” in a field that expects a date. An entry field must be able to reject characters that clearly make no sense at the time they are entered, and with the help of a picture mask it can.

Consider again the example of the picture mask for a date field, “mm/dd/yyyy”. This field actually has three component parts—a month, a day, and a year. These individual parts are referred to as *subfields*. Many seemingly simple data types actually break down into multiple parts when they are displayed on the screen: dates, times (hours, minutes, seconds), phone numbers (area code, prefix, suffix), zip or postal codes, even real numbers. The characters used to separate one subfield from another are called separators. The general form of a field is thus:

```
Subfield ([Separator] [Subfield]) ... ([Separator] [Subfield])
```

In most cases, a field consists of a single subfield, but you should keep in mind while reading this section that matters are not always so simple.

The maximum length of a field and of a picture mask is 255 characters. A field can have a length greater than the length of its picture mask. The last character in the picture mask is assumed to apply to all of the remaining character positions.

In Orpheus, a picture mask consists of one or more of three classes of mask characters: formatting characters, literal characters, and semi-literal characters.

Formatting characters

As far as data entry is concerned, formatting characters denote where in a field new text can be entered or existing text can be modified. Each formatting character is associated with its own character set and only characters in that set can be entered in the corresponding slot. For example, a picture mask of ‘999’ indicates that only numbers (‘0’-’9’) and spaces can be entered in the field. Several formatting characters–'!', 'A', and 'L', for example–also indicate that input should automatically be converted to upper or lower case. For display purposes, formatting characters denote places where a field's "raw" value can be plugged into the picture mask. For example, given the picture mask "$###.##" and a real value of 1.201, the result of merging the value with the picture mask is "$1.20". The #'s indicate the locations in the output string where the digits to the left and right of the fixed decimal point should appear.

Regular mask characters

The regular mask characters are the most commonly used formatting characters. They are used to format strings, numbers, characters, and Boolean values. Each is associated with a set of characters that determines what characters can be entered at a given position in a string. Each regular mask character is also associated with a case-changing quality: no change, force upper case, force lower case, or force mixed case. Unless the comment on the right in Table 9.1 indicates otherwise, the case-changing quality of the mask character is "no change."

Table 9.1: *Regular mask characters*

Constant	**Character**	**Comment**
pmAnyChar	'X'	Any character
pmForceUpper	'!'	Any character, force upper case
pmForceLower	'L'	Any character, force lower case
pmForceMixed	'x'	Any character, force mixed case
pmAlpha	'a'	Alphas only
pmUpperAlpha	'A'	Alphas only, force upper case
pmLowerAlpha	'l'	Alphas only, force lower case
pmPositive	'9'	Numbers and spaces only
pmWhole	'i'	Numbers, spaces, minus
pmDecimal	'#'	Numbers, spaces, minus, period
pmScientific	'E'	Numbers, spaces, minus, period, 'e'
pmTrueFalse	'B'	'T', 't', 'F', 'f'
pmYesNo	'Y'	'Y', 'y', 'N', 'n'
pmBinary	'b'	'0', '1', space
pmHexadecimal	'K'	'0'-'9', 'A'-'F', space, force upper case
pmOctal	'O'	'0'-'7', space

Several of the regular mask characters deserve special attention. The ‘x’ mask character, which forces “mixed case,” affects only the entry of new data. If “sam jones” is entered in an empty field, for example, the result is “Sam Jones.” That is, if the cursor is at the first position in the field, or the previous character is a space, the next character entered is converted to upper case. Characters in other positions are not forced to upper case, nor is the case of existing text changed when a field is displayed or when insertion or deletion of characters alters the position of other characters.

The ‘a’, ‘A’, and ‘l’ mask characters limit the field to alphabetic characters, spaces, hyphens, periods, and commas. Obviously, the set of valid alphabetic characters varies from country to country, so Orpheus uses the Windows API function IsCharAlpha() to determine which characters other than ‘A’-’Z’ should be considered alphabetics. The set of valid alphabetic characters is calculated only once, when an application using one of the Orpheus entry field components begins. It is not recalculated if the language driver is changed while the application is in memory.

The ‘9’ and ‘I’ mask characters are intended primarily for use with fields representing numeric values, either unsigned (‘9’) or signed (‘I’). For a real field that requires the user to enter a decimal character, ‘#’ should be used.

The ‘E’ mask character, which allows real numbers to be entered and displayed using scientific notation, can be used only with simple field types. It cannot be used in the mask for a picture or numeric field. Simple, picture, and numeric fields are described in “Chapter 4: Entry Fields” on page 105.

The ‘K’ mask character, which allows numbers to be entered and displayed in hexadecimal format, should generally be used only with fields of an integer type (Byte, ShortInt, Integer, Word, and LongInt). Furthermore, if a picture mask contains even one instance of a ‘K’ it is assumed that there is a single subfield in the picture mask consisting entirely of ‘K’ characters. The ‘b’ and ‘O’ mask characters are much like ‘K’, except are used when it is necessary to enter or display integer values in binary or octal formats, respectively.

Date and time mask characters

The date and time routines provided by Orpheus store dates and times in compact formats, convert them to other formats when necessary, and perform date and time arithmetic. Long integers are used to represent both dates and times, so that dates in the range 1/1/1600 to 12/ 31/3999 and times in the range 00:00:00 to 23:59:59 can be supported.

One of the chief strengths of the Orpheus date and time routines is that they provide numerous functions for converting Julian dates and times to strings and back again. For more information about Julian dates, see “Date and Time Routines” on page 84. To allow the maximum amount of flexibility, and to insure that the same routines can be used internationally, Orpheus lets you specify a picture mask to indicate how a string is formatted. For example, if a Julian date variable contains January 2, 1900 and you want to convert it to a string (using the DateToDateString routine), there are several picture masks that you could use:

Table 9.2: *Date strings*

Picture	**Result**
`mm/dd/yy`	`"01/02/00"`
`MM/dd/yyyy`	`" 1/02/1900"`
`dd/mm/yyyy`	`"02/01/1900"`
`DD/mm/yyyy`	`" 2/01/1900"`
`dd-nnn-yyyy`	`"02-Jan-1900"`
`dd-NNN-yyyy`	`"02-JAN-1900"`
`www dd nnn yyyy`	`"Tue 02 Jan 1900"`
`WWW NNN dd yyyy`	`"TUE JAN 02 1900"`

Time strings are handled in a similar fashion, except that a special formatting character is available for working with times in am/pm (12-hour) format as well as 24-hour format. For example, if you want to convert a Time variable containing 1:02pm (13:02) into a string using TimeToTimeString, you could use one of these picture masks:

Table 9.3: *Time strings*

Time String	**Result**
`hh:mm:ss`	`"13:02:00"`
`hh:mm`	`"13:02"`
`hh:mm tt`	`"01:02 AM"`
`hh:mmt`	`"01:02P"`

String representations of times are assumed to be in 24-hour format unless a ‘t’ appears in the picture mask. A ‘t’ can appear either once or twice in the picture mask. Keep in mind when using ‘t’ that most countries don’t use the 12-hour format. If you run a program that uses ‘t’ in a picture mask and it is not supported, ‘t’ is replaced by a blank and 24-hour format is used instead.

The following special characters are meaningful in date and time picture masks. Any characters not shown in Table 9.4 are treated as literals when they appear in a picture mask.

Table 9.4: *Date and time mask charcters*

Constant	Character	Comment
pmMonth	'm'	Month - zero-padded
pmMonthU	'M'	Month - blank-padded
pmDay	'd'	Day - zero-padded
pmDayU	'D'	Day - blank-padded
pmYear	'y'	Year - zero-padded
pmMonthName	'n'	Month name
pmMonthNameU	'N'	Month name - force upper case
pmWeekDay	'w'	Weekday
pmWeekDayU	'W'	Weekday - force upper case
pmDateSlash	'/'	mm/dd/yy separator
pmHour	'h'	Hours - zero-padded
pmHourU	'H'	Hours - blank-padded
pmMinute	'm'	Minutes - zero-padded
pmMinuteU	'M'	Minutes - blank-padded
pmSecond	's'	Seconds - zero-padded
pmSecondU	'S'	Seconds - blank-padded
pmAmPm	't'	for "am" and "pm"
pmTimeColon	':'	hh:mm:ss separator

In most cases, the difference between the upper case and lower case formatting characters is that the lower case ones cause numbers to be padded with '0', while the upper case ones cause them to be padded with spaces. For example, if the date is 1/1/1980, a picture mask of "MM/ dd/yyyy" would produce " 1/01/1980." A picture mask of "mm/dd/yyyy" would produce "01/01/ 1980." 'Y' is not allowed as a formatting character, since year values are never padded with spaces.

The characters 'n' and 'N' can be used in place of 'm' and 'M' when the name of the month is desired instead of its number. For example, a picture mask of "dd-nnn-yyyy" yields "01-Jan-1980." Upper case 'N' indicates that the name should be converted to upper case: "dd-NNN-yyyy" yields "01-JAN-1980." The name of the month is truncated based on the number of n's in the mask.

The characters ‘w’ and ‘W’ are used to include the day of the week in a date string. For example, “www dd nnn yyyy” yields “Mon 01 Jan 1980.” Upper case ‘W’ indicates that the name should be converted to upper case: for example, “WWW dd NNN yyyy” yields “MON 01 JAN 1980.” The day of the week is truncated based on the number of w’s in the mask. Orpheus does not allow text to be entered in a subfield containing ‘w’ or ‘W’. The day of the week is supplied automatically when a valid date is entered.

The Epoch property

With the end of the 20th century in our rear-view mirror, there is still some confusion about how programmers and users should handle two-digit year entry. The answer to the quandry is to define the epoch. Since a two-digit entry can only define 100 years, the Epoch property is used to specify the beginning year of the 100 year period. If you specify an Epoch of 1996, the two-digit entry then allows for the years 1996 through 2095.

The Epoch property is used by first breaking it into a century and a year. For example, if Epoch = 1996, the Epoch Century is 1900 and the Epoch Year is 96. The two-digit year entered by the user is then compared to the Epoch Year to determine the correct century as shown in Table 9.5.

Table 9.5: *The Epoch property*

Two-digit year Is	**Result**
`< Epoch Year`	`Epoch Century + 1 + two-digit year`
`>= Epoch Year`	`Epoch Century + two-digit year`

The result is that any two-digit year that is less than the Epoch Year is assumed to be in the next century. For example, if Epoch is 1996, the two-digit entry “99” is converted to “1999” and the two-digit entry “00” is converted to “2000.” All two digit entries in the range of “00” to “95” are converted to “2000 + the two-digit year” while those in the range of “96” to “99” are converted to “1900 + the two-digit year.”

The Pack Parameter

All routines that convert date or time values to strings have a Boolean parameter called Pack. The purpose of Pack is best explained by an example. Suppose you have a date variable that contains 12/1/1994, and that you pass it to DateToDateString with a mask of “MM/DD/yyyy.” If Pack is False, the returned string contains a space in the fourth position: ‘12/ 1/1994’. If Pack is True, this space is removed: “12/1/1994.” Note, however, that literal spaces, those that appear in the picture mask, are never removed. Given a mask of “DD nnn yyyy”, for example, the resulting string is “1 Dec 1992” if Pack is True, or “ 1 Dec 1992” if Pack is False.

User-defined mask characters

Although the standard mask characters defined by Orpheus are adequate for most tasks, there may be times when you need to define your own mask characters. There are eight user-defined mask characters available for this purpose as shown in Table 9.6.

Table 9.6: *User-defined mask characters*

Constant	**Character**	**Comment**
pmUser1	‘1’	User-defined character set 1
pmUser8	‘8’	User-defined character set 8

For example, suppose that you have a Character field in which you want to limit the possible values to ‘A’, ‘B’, ‘C’, or ‘X’. If the list of valid characters didn’t include ‘X’, you could handle this situation by setting a range. The best way to define a non-contiguous set of characters is to use a user-defined mask character (‘1’-’8’). You need to:

1. Assign an appropriate value to the PictureMask property (this example uses ‘1’) in the property editor for the entry field .
2. Create an instance of the TOvcUserData class (see page 1155), and change the meaning of ‘1’.
3. Assign your new instance of the TOvcUserData class to the entry field’s UserData property.

Here is some sample code to show you how to do it:

```
uses
  OvcUser, OvcData;

var
  MyUserData : TOvcUserData;
...
{creates an instance of the TOvcUserData class and initializes
all of the internal data members to default values. By default,
this sets up all user character sets to allow all possible
characters.}

MyUserData := TOvcUserData.Create;

{replaces the default character set for the '1' user mask with
the new set of allowable characters}

MyUserData.UserCharSet[pmUser1] := ['A'..'C', 'X'];

{replaces the default of mcNoChange for the '1' user mask with
mcUpperCase which will force all input to upper case}
```

```
MyUserData.ForceCase[pmUser1] := mcUpperCase;
...
{object assigned to the field's UserData property}
EntryField1.UserData := MyUserData;
```

The contents of the user data object are used whenever the entry field sees a mask character of pmUser1..pmUser8 (‘1’-‘8’).

The TOvcUserData class serves three purposes. First, it defines the set of valid characters for the mask character. In the example, the pmUser1 character (‘1’) is associated with the set [‘A’..‘C’, ‘X’]. Second, the TOvcUserData object defines the case-changing characteristics of the mask character. In the example, the default of mcNoChange is changed to mcUpperCase to give the pmUser1 mask character the upper case characteristic. The third purpose served by a TOvcUserData object is to define the meaning of the eight substitution characters, Subst1-Subst8.

Substitution characters

Suppose that you want to define a picture mask that contains a literal decimal place (.) in it, such as “!!!!!!!!.!!!”. This mask may look OK, but it isn’t, because ‘.’ has a predefined meaning. The dot in the mask represents a decimal point, and when a field’s mask contains a decimal point it’s assumed to represent a floating point number, leading the entry field to perform some special processing that is not appropriate if the field doesn’t contain a number. The way to get around this problem is to use a substitution character.

There are eight user-defined substitution characters available as shown in Table 9.7.

Table 9.7: *User-defined substitution characters*

	Substitution Character	**Description**
Subst1	#241	For user-defined substitution
Subst8	#248	For user-defined substitution

Instead of using a dot in the picture mask, use the Subst1 mask character (ASCII value 241) and then define Subst1 to represent the decimal place. Then, when the entry field is formatting a string and sees Subst1 in a picture mask, it substitutes the dot for Subst1.

The procedure for creating a user-defined substitution character is basically the same as the procedure for creating a user-defined mask character. The only difference is that you must assign the substituted character to the SubstChar property instead of the assigning a set of characters to the UserCharSet property. The example shown above works equally well here if you replace the UserCharSet and ForceCase assignment statements with the following:

```
MyUserData.SubstChars[Subst1] := '.';
```

The only tricky part about using substitution characters is that Windows makes it difficult to enter ASCII characters with values in the range 241 (Subst1) to 248 (Subst8), something you need to be able to do when designing your form. Here's how to enter one of these characters: turn on NumLock, press and hold <Alt> while entering '0nnn' on the numeric keypad, where 'nnn' is 241-248.

Literal characters

In general, any character in a picture mask that has not been assigned a function as a formatting character is treated as a literal, meaning that its location in the field cannot be altered during editing, and that it is not altered when a value is converted to a string. For example, given a picture mask of "[999]" and any integer value between 0 and 999, the result of merging the value with the picture mask is a string that starts with '[' and ends with ']' because these two characters are treated as literals.

In most cases, the literal characters in a picture mask are stripped out in the process of converting a string back to its native data type, and restored when converting it back to a string. In the case of data entry fields of type string, these literal characters become a permanent part of the string being edited, to ensure that they will look the same when you display them as they do when they are being edited. There is an option to disable this behavior.

Semi-literal characters

Semi-literal characters are like literal characters in the sense that they usually correspond to locations in a field that cannot be altered while editing. The two currency characters ('c' and 'C'), for example, always mark slots in a field that cannot be changed. What appears on the screen and in the edited string, however, is usually a '$'. In a couple of cases, however, a semi- literal character requires more than a simple substitution. The floating dollar sign ('$') sets aside a slot in a field that is used only if the rest of the field is full. If the field is not full, the dollar sign is moved just to the left of the leftmost digit. Somewhat similarly, the comma character indicates a slot is to be filled with a comma (or its equivalent) only if there is a digit to put to the left of it. Otherwise, its slot is filled with a dollar sign, minus sign, or with a digit entered by the user. The characters used to fill positions corresponding to semi-literal mask characters are determined at run time based on configuration data found in WIN.INI.

In most cases, the semi-literal mask characters represent locations in the field where a one-to- one substitution is made when the field is being formatted. For example, if the settings in WIN.INI indicate that a dot should be used as a separator in strings representing real numbers, then all instances of commas in the picture mask are translated to a dot when the field's value is merged with the picture mask. Note that the dollar sign and comma mask characters can be used in picture entry fields only if the field contains a fixed decimal point or the field allows only whole numbers, hence, "$###,###.##" and "$999,999" are both valid masks, while “$###,###” is not, since it allows a decimal point to be entered by the user but does not assign it a fixed position. There are no such restrictions for using these mask characters in masks for numeric entry fields. Table 9.8 shows a list of semi-literal mask characters.

Table 9.8: *Semi-literal mask characters*

	Semi-literal Mask Character	**Description**
`pmFloatDollar`	`‘$’`	`Floating dollar sign`
`pmCurrencyLt`	`‘c’`	`Currency to left of amount`
`pmCurrencyRt`	`‘C’`	`Currency to right of amount`
`pmDecimalPt`	`‘.’`	`Insert decimal point character`
`pmComma`	`‘,’`	`Character used to separate numbers`
`pmNegParens`	`‘p’`	`() should be used for negative #'s`
`pmNegHere`	`‘g’`	`Place holder for minus sign`

The last two semi-literals are used to format negative numbers. The ‘g’ mask character indicates that the minus sign, if there is one, should be displayed in the indicated position. It is typically used at the end of the mask. The ‘p’ mask character must be placed at the end of the picture mask and indicates that negative numbers should be displayed in parentheses. For example, suppose that you have a mask of “iiiip” and “-12” is entered. The field is displayed as “(12).” These two mask characters can be used only in numeric entry fields. They cannot be used in simple or picture fields.

Samples of picture masks

To give you a better idea of what the mask characters mean, Table 9.9 shows examples of some typical picture masks and the resulting strings.

Table 9.9: *Examples of picture masks*

Picture	**Result**	**Remarks**
`“XXXXXXXXXX”`	`"Hello world"`	Any character accepted
`“!!!!!!!!!!”`	`"HELLO WORLD"`	Any character, force upper case
`“LLLLLLLLLL”`	`"hello world"`	Any character, force lower case
`“xxxxxxxxxx”`	`"Hello World"`	Any character, force mixed case (on entry only)
`“aaaaaaaaaa”`	`"Hello world"`	Alphas only
`“AAAAAAAAAA”`	`"HELLO WORLD"`	Alphas only, force upper case
`“llllllllll”`	`"hello world"`	Alphas only, force lower case
`“[9999]”`	`"[ 123]"`	Digits and ' ' only, brackets are literals
`“[iiii]”`	`"[-123]"`	'-' is allowed
`“EEEEEEEEEE”`	`"1.234567E+17"`	Scientific notation
`'KKKK'`	`"07FE"`	Hexadecimal numbers
`“bbbbbbbb”`	`"01110011"`	Binary numbers
`‘B’`	`'T'`	Boolean (True-False)
`‘Y’`	`'N'`	Yes/No
`“c999.99”`	`"$ 12.50"`	Dollar sign is fixed
`“$999.99”`	`" $12.50"`	Dollar sign floats
`“$###,###.##”`	`" $1,234.56"`	‘,’ is discretionary, ‘$’ floats
`“$###,###.##”`	`" -$12.34"`	Normal handling of minus sign
`“$###,###.##g”`	`" $12.34-"`	Minus sign in fixed position
`“$###,###.##p”`	`" ($12.34)"`	Negative numbers shown in parentheses
`“mm/dd/yyyy”`	`"01/06/1926"`	Month and date are padded with zeros
`“MM/dd/yyyy”`	`" 1/06/1926"`	Month is padded with spaces
`“mm/dd/yy”`	`"01/06/26"`	Century omitted
`'dd/mm/yyyy'`	`'06/01/1926'`	British date format

Table 9.9: *Examples of picture masks (continued)*

Picture	**Result**	**Remarks**
“dd/mm/yyyy”	"06/01/1926"	British date format
“dd-nnn-yyyy”	"06-Jan-1926"	Month name instead of number
“hh:mm”	"21:30"	Time in 24-hour format
“hh:mmt”	"11:30p"	Time in am/pm format, no 'm'
“HH:mm tt”	" 1:30 pm"	Am/pm format, hour padded with spaces

In these examples there is no mixing of formatting characters within subfields. For example, there are no masks like “!XXXX” (convert first character to upper case, leave others alone). Although picture masks like these are legal, they are not guaranteed to produce the desired results, because there are too many situations in which the rules imposed by the picture mask become ambiguous. For example, what if the picture mask were (nonsensically) “9#LA”? It would be easy enough to enforce the implied restrictions if the field were empty initially and new text was being entered. But what should be done if the field is full (e.g., “1.uA”) and the first character in the string is deleted, thereby pulling the end of the string toward the front (“.uA ”)? Should the deletion be allowed? What if the user wanted only to delete the character and immediately insert another, valid character in its place? The approach taken in Orpheus is to enforce the rules implied by the picture mask only when the character is being entered. It is more prudent in such cases to allow the entry and validate the field by some other means later on.

Many of the examples represent fields that are composed of multiple subfields. In each instance, the separator is either a literal or a semi-literal character, never a formatting character (such as ‘9’ or ‘!’). In terms of picture masks, a subfield is generally defined as a series of formatting characters bounded by either a subfield separator (such as ‘/’ or ‘-’), a literal character (such as a space character), a semi-literal character with a fixed position (such as ‘c’), or one end of the field. The only exceptions to this rule are the rare cases where nothing separates two subfields, such as “hh:mmt”, where the ‘t’ immediately follows the final ‘m’ in the minutes subfield. Although these cases can be handled if necessary, they should generally be avoided for the same reason that mixing of formatting characters within a subfield should be avoided.

Changing a field’s picture mask

Several important guidelines should be kept in mind when changing a field’s picture mask. First, the new picture mask should be assigned before the field is actually displayed. This can usually be done in the form’s OnCreate method. The main purpose of this ability is to allow a program to change the masks used for date, time, and currency fields based on Windows’ current international settings. It might be possible to change the picture mask in other situations, at other points in the program, but it could cause you problems.

Second, you must be very careful when changing the mask for fields of type string. The new mask must be the same length as the old mask, or it will cause problems when data is transferred to and from the field. If necessary, pad the end of the mask with blanks to the maximum length. This is not a concern with other field types, because the amount of data transferred is a function of the size of the data type, not the maximum length of the field.

Third, minimal error checking is done on the mask assigned to the PictureMask property. If the mask is bad, you can generate an exception or some other undesirable result. For example, if the field is a simple field, passing an empty string as the mask creates a situation in which the cursor has nowhere to go in the field; the results are unpredictable. If you are in doubt as to whether or not a mask is legal, try creating a field of the same type and mask at design time. If the mask is blatantly invalid, it will be rejected by the property editor, and if it has subtle problems they will probably be revealed when you test the form.

Finally, you must be careful when changing the picture mask for a field containing a monetary value. Suppose that, when designing a form, you add a real picture field with a mask of “##########”, the default mask for real fields, and leave decimal places set to 0. You then use InternationalCurrency and the PictureMask property to change the mask at run time, and the result is “$###,###.##”. If the value to be displayed initially in the field is 1.23, you will see “$1.00” on the screen.

Why does this happen? When the field first converts the data to a display string, it is transformed into ‘   1’, since the initial mask indicated no decimal point and the DecimalPlaces property was left at its default value of 0. When the picture mask is changed, Orpheus converts the value currently in the edit string to its native data type (a real) and stores it in a temporary variable. It changes the mask for the field, then reloads the data from the temporary variable. In this case, the temporary variable was set to ‘1’, so after the mask change the field contains “ $1.00”. This problem can be avoided if the original mask or DecimalPlaces property specifies two digits of precision to the right of the decimal point.

The pad character

Figure 9.1 shows the Pad Character dialog box.

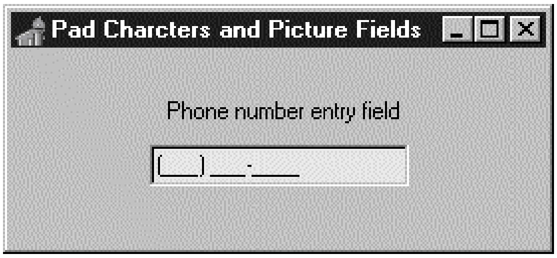


Figure 9.1: Pad Characters and Picture Fields dialog box

The edit string for both picture and numeric fields is always padded to its maximum length with spaces. This is one of the things that happens when a field's raw data is converted to a string and then merged with the field's picture mask. Although the character used to pad the edit string cannot be changed, you can change the character that is used to pad the display string. This character is called the pad character and it is a space by default. One common use of this feature is to pad a field with underscore characters, to indicate how much data can be entered in the field and where. This effect can be achieved by assigning the '_' character to the PadChar property.

Another use of this feature is to left-pad numeric entry fields of type Word with zeros.

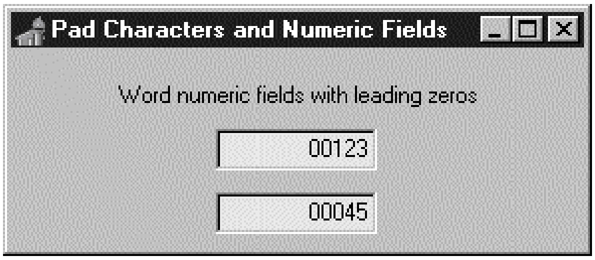

Pad Characters and Numeric Fields
Word numeric fields with leading zeros
00123
00045

Figure 9.2: Pad Characters and Numeric Fields dialog box

Control characters

Orpheus entry field components allow control characters to be entered in any field—as long as the mask or mask character do not prevent it—by entering the key sequence assigned to the ccControlChar command followed by the control character. (When using the WordStar command table, the key sequence is <CtrlP> followed by the control character.) See "TOvcCommandProcessor Class" on page 50 for additional information concerning command tables and the ccControlChar command in particular.

Control characters are represented as upper-case alphabetics, and are displayed in a special color, normally red. You can change the color by assigning a new color value to the ControlCharColor property.

Field Validation

When the input focus leaves any of the data entry field components, Orpheus performs a test to validate the contents of the field. The nature of the test varies depending on the type of data associated with the field, but in most cases it includes a range check and some form of validity check (e.g., for a date field, ‘02/30/1993’ is not valid because there is no such date). You can also hook into the validation system to perform your own tests by writing and assigning an event handler to the OnUserValidation event.

What happens when a field is invalid? That’s up to you. By default, Orpheus displays an error message in a dialog box and returns the focus to the invalid field. This is the way validation errors have traditionally been handled. If you have assigned a method to the OnError event, the entry field passes the error number and corresponding message text to the method you assigned and then asks its Controller to return the focus. Alternatively, you can select the “soft validation” option. This allows the input focus to leave the field, but a flag is set to mark the field as invalid. Your program can then check for this condition. Additionally, invalid fields can be displayed using alternate colors by setting the value of the ErrorColor property. This is the way most other validated data entry packages for Windows work. Orpheus lets you choose any of these three methods.

“Soft” vs. “hard” validation

The default is to perform strict (or “hard”) validation on the input in each field. If an invalid response is entered in a field or a required field is left empty, the input focus is forced to return to the field and the Controller, or your error handler, is called to report the error.

In most cases, this is desirable both for the user and the programmer. The user learns about errors immediately. And the programmer doesn’t have to deal with invalid responses when the data entered by the user is needed before the form is accepted (e.g., when implementing calculated fields). In other cases, however, it might be desirable to postpone the enforcement process until the user accepts the form, and to instead use some visual effect to indicate that a field’s data is invalid. This is the “soft validation” option, and you can select it either for a particular field or all the fields in the form. See “Address Book” on page 1278 for an explanation of why soft validation is desirable in such cases.

To set the soft validation option for an individual field, simply set the SoftValidation property to True. To set this option for multiple fields, select all of them before setting the SoftValidation property.

The soft validation option does not stop Orpheus from performing its validation tests. It simply changes what happens when a test fails. Rather than forcing the focus to return to the field and calling the method assigned to the OnError event, Orpheus sets a flag to indicate that the field's value is invalid. You can test the status of this flag by calling the field's IsValid method.

Custom validation

Although the built-in validation facilities are adequate in most cases, there might be times when you need to perform some special validation on a field. A good example of such a case is the data entry form in the Address Book demonstration program (see page 1278), where the user must enter a two-character abbreviation for a state. To prevent an invalid state abbreviation from being entered, a custom validation routine is used to compare the entry to a list of valid abbreviations, and to generate an error if the entry is invalid.

To implement your own validation routine, create a method and assign its name to the OnUserValidation event. The method assigned to the OnUserValidation event is called after the default field validation is complete. See the OnUserValidation event in "TOvcBaseEntryField Class" on page 363for additional information.

When your user validation method is called, the value of the ErrorCode parameter is either 0 (indicating that the default validation routine found no errors) or one of the error codes discussed in the OnError event in "TOvcBaseEntryField Class" on page 363. In general, if ErrorCode is non-zero, there is no need to perform additional validation since the error code already indicates that the field contains invalid data. You can simply exit the method. If ErrorCode is 0, you can perform any additional validation that the field requires. If you are not relying on the validation "helper" routines (ValidateSubfields, ValidateNotPartial, and ValidateNoBlanks), you can use either the EditString property or the GetStrippedEditString function to get the string to be validated. See "Validation Helper Routines" on page 333 for more information.

If you need to perform all of the validation yourself, you can ignore the error value (if any) contained in the ErrorCode parameter and revalidate the field yourself, based on your unique criteria.

Caution: The value of the entry field cannot be changed during processing of your OnUserValidation event handler. Attempts to change the field value are ignored.

Validation Helper Routines

The OvcValid unit implements three routines that perform additional validation for picture and numeric entry fields. ValidateNoBlanks checks for a completely full field. ValidateNotPartial checks for a completely full or completely empty field. ValidateSubFields checks the subfields in a string. These routines apply to Orpheus entry fields only and should normally be used in your OnUserValidation event handler to handle special validation needs. See OnUserValidation in “TOvcBaseEntryField Class” on page 363for more information.

Functions

ValidateNoBlanks	ValidateNotPartial	ValidateSubFields

Reference Section

ValidateNoBlanks — function

```
function ValidateNoBlanks(EF : TOvcPictureBase) : Word;
```

↳ Validates that a field is completely full.

This validation helper routine verifies that no usable subfields in a string contain spaces. If an error is found, it returns an oeBlanksInField error code, otherwise it returns 0.

ValidateNoBlanks is intended to be called from an OnUserValidation event handler. The returned error code, if any, should be assigned to the OnUserValidation event handler's ErrorCode parameter.

This validation routine imposes more stringent requirements than designating a field as required, since required fields may contain partial entries.

ValidateNoBlanks cannot be used with simple field types.

ValidateNotPartial — function

```
function ValidateNotPartial(EF : TOvcPictureBase) : Word;
```

↳ Validates that a field is either completely full or completely empty.

This validation helper routine verifies that no usable subfields in a string contain spaces unless all do. If an error is found, ValidateNotPartial returns an oeBlanksInField error code. If the field is not empty, then the validation test is identical to that of ValidateNoBlanks.

ValidateNotPartial is intended to be called from an OnUserValidation event handler. The returned error code, if any, should be assigned to the OnUserValidation event handler's ErrorCode parameter.

ValidateNotPartial cannot be used with simple field types.

ValidateSubFields — function

```
function ValidateSubfields(
  EF : TOvcPictureBase; const SubfieldMask : string) : Word;
```

↳ Validates that subfields in a string meet the requirements of SubfieldMask.

This validation helper routine is intended to aid in the validation of fields containing multiple subfields, some of which are required and some of which are not. If the field meets the requirements of SubfieldMask, 0 is returned, otherwise an oePartialEntry or oeBlanksInField error code.

SubfieldMask must be the same length as the picture mask for the field being validated. The following characters are meaningful in SubfieldMask:

Character		**Description**
r	ReqdChar	Subfield is required, no blank spaces allowed.
u	UnlessChar	Subfield is required, unless entire field is empty.
p	PartialChar	Subfield is optional, but cannot be partially full.

These characters cannot be mixed within a subfield.

Suppose that a phone number field has a mask of "(999) 999-9999." The area code subfield is optional, although it should not contain a partial entry, and the other two subfields are required. Use a SubfieldMask of "(ppp) rrr-rrrr." By contrast, if you want the last two subfields to be either completely full or completely empty, use a SubfieldMask of "(ppp) uuu-uuuu."

ValidateSubFields cannot be used with simple field types.

Special Field Types

Uninitialized fields

Many users like to see a completely blank data entry form when they are entering new data. That is, they want all the fields to appear completely blank. If your entry forms consist entirely of simple string fields, accommodating these users is not a problem. But if you're using picture fields with literal characters in their masks, or if the entry form contains integer or real fields, there's a potential problem, because these fields will never be truly empty. The solution to the problem is to allow you to mark a field as uninitialized.

You can do this in two ways. To mark individual fields, set the Uninitialized property. Alternatively, you can call the form controller's MarkAsUninitialized method.

When a field's uninitialized flag is set, it is displayed completely blank except when it has the input focus. While being edited, it looks just as it would if the flag were not set. The uninitialized flag is cleared as soon as the field is modified.

To complicate matters a bit further, some users who want the "blank screen" look also want the field to be displayed normally as soon as the caret is moved through the field once, whether the field was modified or not. Although Orpheus cannot give this effect automatically, it's fairly easy to achieve. You only need to mark the fields as uninitialized and write an OnExit event handler that clears the uninitialized flag for the current entry field. For an example of how to do this, see the Address Book demonstration program on page 1278.

Required fields

In many situations, one or more fields within a data entry form absolutely must be filled in by the user for the data to be of any use. Such fields are called required fields. Orpheus makes it easy to flag a field as being required. Simply set the InputRequired property. Orpheus performs a validation test to make sure that the field is filled in when the field is losing the focus and then again if you call Controller.ValidateEntryFields(). If the field is not filled in, an error is generated.

Normally, only fields of type string, date, or time should be marked as required, since these are the only fields that can be truly left empty by the user. An integer field, for example, cannot. The validation routine for an integer field rejects any input that is not a valid whole number in the range -32768 to 32767. In some cases, however, you may want to force the user to edit the value in a field that is not one of these three types, if only to ensure that the default value is not accepted without thinking about it.

In these cases, go ahead and mark the field as required. When the field is created, Orpheus automatically marks it as uninitialized. If the uninitialized flag is set for a required field, the field fails at the validation test.

Read-only fields

There are a couple of reasons that you might want to prevent the user from modifying a field but still allow the caret to be moved into it (instead of disabling the field by setting the Enabled property to False). For example, if the field scrolls horizontally, the user can't see it completely without moving the caret into it and scrolling through it. A second reason is that you might want the user to be able to select text within the field and copy it to the clipboard. You can accomplish either of these by setting the ReadOnly property to True.

Password fields

Orpheus provides the capability for your program to prompt for a password to prevent unauthorized access to data. This can be done using either a simple or picture entry field. To create a password field, set the PasswordMode property to True. When the field is edited, entered characters are displayed as asterisks (to use a different character, assign the character to the PasswordChar property).

For more information on passwords, see "TOvcSimpleField Component" on page 398 and "TOvcPictureField Component" on page 411.

9

Chapter 10: International Support Issues

In adapting a program written with Orpheus for use in a country other than the United States, there are really only a few areas of concern. First and most obvious are the language differences. Orpheus supplies an answer for part of this problem by defining codes for all the messages generated. This allows you to write error handlers that can generate error messages in whatever language you support. Alternatively, all the strings displayed by Orpheus are defined as ResourceStrings in O32SR.inc. The default is of course, English but other language versions of O32SR.inc are available. You can easily create your own language version of Orpheus by replacing O32RS.inc with a translated, language specific version of your own. One of the issues that arises from the use of language specific versions of Orpheus is with Boolean and Yes/No fields, since 'T'/'F' and 'Y'/ 'N' aren't appropriate in all countries. Orpheus deals with this problem by reading the characters associated with True, False, Yes, and No from the StringResource file.

In addition to the language differences, different countries have different conventions on how dates, times, and monetary values are represented. Most European countries, for example, place the day before the month ("31/12/1999"), whereas in the United States the order is reversed ("12/31/1999"). And the character used to separate the various parts of the date can vary too. In Switzerland, a period is used rather than a slash or a dash: "31.12.1999."

Picture masks make it simple to deal with differences in order. If you want the day to be displayed before the month, change the order of the sub-fields in the picture mask. Differences in separators are handled automatically by means of the semi-literal mask characters shown in Table 10.1.

Table 10.1: *Semi-literal mask characters*

Char	**Meaning**	**WIN.INI Entry**
'.'	Decimal point	sDecimal
','	Thousands separator	sThousands
'$'	Floating currency	sCurrency, iCurrency
'c'	Fixed currency - left	"
'C'	Fixed currency - right	"
':'	Time separator	sTime
't'	Am/pm indicators	s1159, s2359
'/'	Date separator	sDate

When an application or DLL that uses an Orpheus entry field is first loaded, initialization code checks the international settings maintained by Windows to determine what character(s) should be substituted for each of the semi-literal mask characters shown in Table 10.1.

This much of the problem is handled for you automatically, and if your entry forms don't display date, time, or monetary values, you don't have to do anything special to make your entry forms adapt themselves at run-time to the current configuration settings.

Monetary values

The key to dealing with monetary differences lies in the InternationalCurrency method defined in "TOvcIntlSup Class" on page 344. This method constructs an appropriate picture mask for a field that displays a monetary value. You tell it what mask character to use ('9', 'i', or '#'), how many digits to display to the left of the decimal point, whether the currency symbol should float, and whether commas are desired. It returns a picture mask that takes into account all the relevant configuration settings. The only remaining step is to use the resulting string to change the picture mask for the appropriate fields in your entry form. See "Changing a field's picture mask" on page 328 for information about what is required to change a picture mask at run time.

Dates and times

When an application or DLL that uses an Orpheus date or time routine is first loaded, initialization code gets the current international settings from Windows. Orpheus uses the following information:

- The character used as a separator in dates (replaces '/')
- The character used as a separator in times (replaces ':')
- The string used to represent am/pm in times, if any (replaces 't' or "tt")
- The standard format for short dates
- The standard format for long dates
- The standard format for times

The first three settings are used throughout Orpheus when converting date and time variables to strings. The last three are used only by InternationalDate, InternationalLongDate, and InternationalTime (see "Long dates and short dates" on page 342). The purpose of these routines is to allow you to create picture masks that are appropriate, given the current international settings.

If you don’t use the international routines faithfully, and your applications are used in different countries, you run the risk of displaying dates and times in a format that is only partially right—and potentially quite confusing. For example, in the U.S. it is correct to use a mask of “mm/dd/yyyy”, which yields a string such as “12/01/1992.” But if your program is run in Germany, where the standard date format is “dd.mm.yyyy”, the string appears as “12.01.1992”, which means January 12th, not December 1st. You can avoid this kind of problem by calling InternationalDate to get your picture mask. It returns a mask of “dd/mm/yyyy” in Germany, and DateToDateString or the picture field takes care of replacing the slashes with periods.

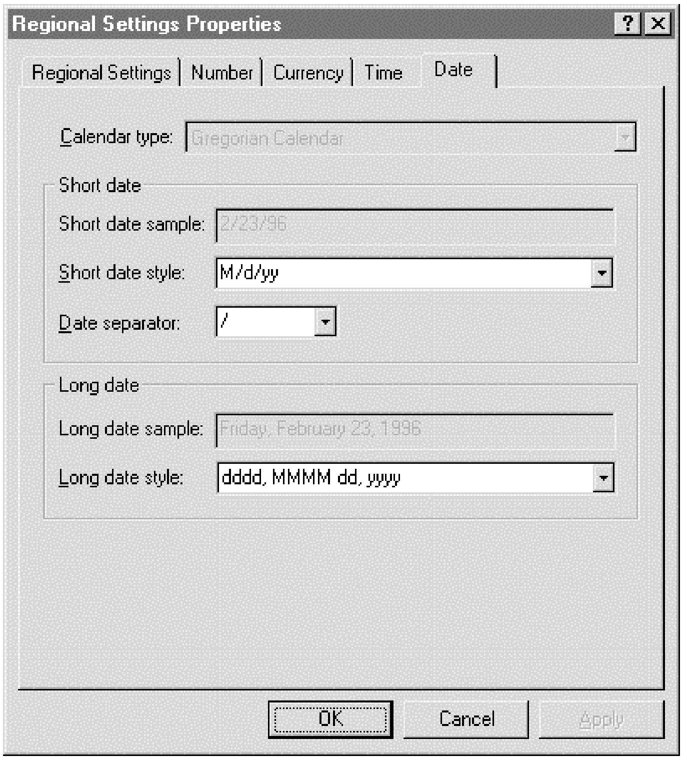
Although Windows 3.1 provides most of the information needed to display dates properly in various countries, it fails in two crucial respects: it doesn’t provide the names of the months or the names of the days of the week in each country. Delphi 1.0 addresses this deficiency by storing these names in a resource that can easily be changed depending on the target country.

The Win32 API addresses the issue, and provides strings for month and day names based on the current regional setting. Delphi and C++Builder obtain these strings from Windows.

Orpheus obtains the month and day names from VCL routines, so the original source of the strings will depend on which version of Delphi or C++Builder you are using.

Long dates and short dates

Windows defines two standard formats for dates, the long date format and the short date format. If you aren't familiar with these terms, you might want to run the Windows Control Panel program and select the International icon. This brings up a dialog box that lets you change or customize the various international settings.



Calendar type: Gregorian Calendar

Short date

Short date sample: 2/23/96

Short date style: M/d/yy

Date separator: /

Long date

Long date sample: Friday, February 23, 1996

Long date style: dddd, MMMM dd, yyyy

Figure 10.1: Regional Setting Properties dialog box

In the dialog box shown in Figure 10.1, the current date is displayed in two formats. The first is the short date format and the second is the long date format.

Orpheus makes it easy to write applications that observe the preferences and conventions reflected in these settings. When you want to display dates using the short date format, call InternationalDate to obtain a picture mask. To use the long date format, call InternationalLongDate instead.

When using the long date format, the mask you get from InternationalLongDate sometimes sets aside more space for a month or day-of-week name than is actually needed. In the U.S., for example, 9 spaces are reserved for the month name, to account for September. If the month is May, however, that's 6 spaces too many. To remove these extra spaces when converting a date to a string with DateToDateString, simply pass True as the Pack parameter.

TOvcIntlSup Class

The TOvcIntlSup class serves as a link between the international settings maintained by Windows and components whose behavior depends on the Windows international settings. This class and its companion unit, STDATE, provide a full suite of date and time handling routines. STDATE is described in “Date and Time Routines” on page 84.

When an instance of the TOvcIntlSup class is created, it retrieves the international settings from Windows. These settings are stored in the instance of TOvcIntlSup and are used during formatting operations and to create picture masks that are used by several Orpheus components.A global instance of the TOvcIntlSup class (named OvcIntlSup) is created automatically and used as the default OvcIntlSupport property value for several Orpheus components.

Caution: Directly modifying OvcIntlSup will affect all components that use it. Therefore, be careful when changing OvcIntlSup’s default property values.

Hierarchy

TObject (VCL)

  TOvcIntlSup (OvcIntl)

Properties

AutoUpdate	CurrencyRtStr	SlashChar
CommaChar	DecimalChar	TrueChar
Country	FalseChar	YesChar
CurrencyDigits	ListChar	
CurrencyLtStr	NoChar	

Methods

CurrentDateString	DMYtoDateString	ResetInternationalInfo
CurrentTimeString	InternationalCurrency	TimeStringToHMS
DateStringIsBlank	InternationalDate	TimeStringToTime
DateStringToDate	InternationalLongDate	TimeToAmPmString
DateStringToDMY	InternationalTime	TimeToTimeString
DateToDateString	MonthStringToMonth	
DayOfWeekToString	MonthToString	

Events

OnWinIniChange

Reference Section

AutoUpdate — run-time property

```
property AutoUpdate : Boolean
```

Default: False

Determines if changes in the Windows international settings are detected.

If AutoUpdate is True, changes made in the Windows international settings while an application is running are detected and used to reset the values of the TOvcIntlSup properties automatically.

When AutoUpdate is True, a window handle is created so that an internal window procedure can respond to WM_WININICHANGE messages. Setting AutoUpdate to False destroys the internal window handle and disables WM_WININICHANGE handling.

When an instance of TOvcIntlSup is created, all its properties are initialized to match the current international settings. Therefore, it is not necessary to set this property to True unless you want changes to the international settings to be reflected in your application at run time.

CommaChar — run-time property

```
property CommaChar : AnsiChar
```

The character used to separate digits in groups of a thousand.

CommaChar is initialized to the “Digit grouping symbol” value defined by Windows.

See also: DecimalChar

Country — run-time, read-only property

```
property Country : string
```

Returns the country string contained in the Windows international settings.

CurrencyDigits — run-time property

```
property CurrencyDigits : Byte
```

The number of decimal places to use when formatting currency values.

CurrencyDigits is initialized to the “No. of digits after decimal” value defined by Windows.

CurrencyLtStr run-time property

```
property CurrencyLtStr : TCurrencySt
```

```
TCurrencySt = array[0..5] of AnsiChar;
```

Holds the characters placed to the left of a currency value.

Normally, this is a single character, but it can contain up to five characters.

See also: CurrencyRtStr

CurrencyRtStr run-time property

```
property CurrencyRtStr : TCurrencySt
```

```
TCurrencySt = array[0..5] of AnsiChar;
```

Holds the characters placed to the right of a currency value.

Normally, this property is an empty string, but it can contain up to five characters.

See also: CurrencyLtStr

CurrentDateString method

```
function CurrentDateString(
  const Picture : string; Pack : Boolean): string;
```

Returns today's date as a string.

This method calls StDate.CurrentDate to get the current date, then converts it (using DateToDateString) to a string of the form specified by Picture. If Pack is True, CurrentDateString compresses non-literal spaces in the returned string.

The following example gets today's date as a string of the form "MM/DD/yyyy" with extraneous spaces removed:

```
  S := OvcIntlSup.CurrentDateString('MM/DD/yyyy', True);
```

See also: CurrentTimeString, DateToDateString

CurrentTimeString method

```
function CurrentTimeString(
  const Picture : string; Pack : Boolean): string;
```

Returns the current time as a string.

This method calls StDate.CurrentTime to get the current time, then converts it (using TimeToTimeString) to a string of the form specified by Picture. If Pack is True, CurrentTimeString compresses non-literal spaces in the returned string.

The following example gets the current time as a string of the form “hh:mm:ss tt”:

```
  var
    S : string;
  ...
  S := OvcIntlSup.CurrentTimeString('hh:mm:ss tt', False);
```

See also: CurrentDateString, TimeToTimeString

DateStringIsBlank method

```
function DateStringIsBlank(
  const Picture, S : string) : Boolean;
```

Returns True if the date is blank.

Given a string representation of a date (S), DateStringIsBlank returns True if the month, day, and year in S are all blank.

The following example returns True:

```
  var
    B : Boolean;
  ...
  B := OvcIntlSup.DateStringIsBlank('mm/dd/yyyy', '  /  /    ');
```

DateStringToDate method

```
function DateStringToDate(
  const Picture, S : string; Epoch : Integer) : TStDate;
```

Converts a string to a Julian date.

Given a string representation of a date (S), this method converts it to a Julian date, returned as the function result. If S is not a string of the same form as Picture, the results are unpredictable. If S is the correct form, but the date represented by S is invalid, BadDate is returned.

If Year is less than 100 (a two-digit year), Epoch must be a four-digit year defining the starting year of the 100 year period. See “The Epoch property” on page 321.

The following example returns the Julian date for “01/01/1999”:

```
  var
    D : TStDate;
  ...
  D := OvcIntlSup.DateStringToDate('mm/dd/yyyy', '01/01/1999', 0);
```

See also: DateToDateString, DateStringToDMY

DateStringToDMY method

```
function DateStringToDMY(
  const Picture, S : string;
  var Day, Month, Year : Integer; Epoch : Integer) : Boolean;
```

Extracts the day, month, and year from a date string.

Given a string representation of a date (S), this method extracts the Day, Month, and Year, and returns True if the date is valid. If S is not a string of the same form as Picture, the results are unpredictable.

If Year is less than 100 (a two-digit year), Epoch must be a four-digit year defining the starting year of the 100-year period. See “The Epoch property” on page 321 for more information.

The following example extracts the day, month, and year for “11/13/1987” returning True, with Day = 13, Month = 11, and Year = 1987:

```
  var
    Day : Integer;
    Month : Integer;
    Year : Integer;
  ...
  OvcIntlSup.DateStringToDMY('mm/dd/yyyy', '11/13/1987',
                             Day, Month, Year, 1980);
```

See also: DateToDateString, DMYToDateString

DateToDateString method

```
function DateToDateString(
  const Picture : string;  Julian : TStDate;
  Pack : Boolean) : string;
```

↳ Converts a Julian date to a string.

Given a date (Julian), this method converts it to a string of the form specified by Picture and returns it as the function result. If Julian equals BadDate, the month, day, and year are represented as blank spaces. If Pack is True, DateToDateString compresses non-literal spaces in the returned string.

The following example returns a string of the form “MM/DD/yyyy” representing the Julian date contained in MyDateVar with extraneous spaces removed:

```
  var
    S : string;
  ...
  S := OvcIntlSup.DateToDateString('MM/DD/yyyy', MyDateVar, True);
```

See also: DateStringToDate, DMYToDateString

DayOfWeekToString method

```
function DayOfWeekToString(WeekDay : TDayType) : string;
```

```
TDayType = (
  Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
```

10

↳ Returns a string for the specified day of the week.

This method returns the name of the specified day of the week as the function result.

Day names are obtained using the LongDayNames string array.

The following example returns the string, “Tuesday”:

```
  var
    S : string;
  ...
  S := OvcIntlSup.DayOfWeekToString(Tuesday);
```

DecimalChar run-time property

```
property DecimalChar : AnsiChar
```

The character displayed to the left of any decimal places.

DecimalChar is initialized to the “Decimal symbol” value defined by Windows.

See also: CommaChar

DMYtoDateString method

```
function DMYtoDateString(
  const Picture : string; Day, Month, Year : Integer;
  Pack : Boolean; Epoch : Integer) : string;
```

Merges the month, day, and year into a string.

Given a Day, Month, and Year, this method returns a string representation of the date in the format indicated by Picture. The date string is returned as the function result. If Pack is True, DMYtoDateString compresses non-literal spaces in the returned string.

If Year is less than 100 (a two-digit year), Epoch must be a four-digit year defining the starting year of the 100 year period. See “The Epoch property” on page 321 for more information.

The following example sets S1 and S2 to “07/11/1964”:

```
  var
    S1 : string;
    S2 : string;
  ...
  S1 :=
   OvcIntlSup.DMYtoDateString('mm/dd/yyyy', 11, 7, 1964, False, 0);
  S2 := OvcIntlSup.DMYtoDateString('mm/dd/yyyy', 11, 7,
    64, False, 1950);
```

See also: DateStringToDMY

FalseChar run-time property

```
property FalseChar : AnsiChar
```

Default: ‘F’

The character used to display a False condition for True or False fields.

See also: TrueChar

```
function InternationalCurrency(
  FormChar : AnsiChar; MaxDigits : Byte;
  Float, AddCommas, IsNumeric : Boolean) : string;
```

Returns a picture mask for a currency string.

This method constructs a picture mask for a field containing a monetary value. The mask it returns is typically assigned to the PictureMask property of a TOvcPictureField or TOvcNumericField component.

The generated mask is returned as the function result. FormChar is the formatting character to use (‘9’, ‘i’, or ‘#’). MaxDigits is the number of digits that can appear to the left of the decimal point. (WIN.INI indicates how many, if any, should appear to the right.) Float indicates whether ‘$’ or ‘c’ should be used if the currency symbol goes on the left. If Float is False, ‘c’ is used instead of ‘$’. AddCommas indicates whether discretionary commas should appear in the picture mask. IsNumeric indicates whether or not the mask will be used with a numeric field. If it’s True, the returned mask can contain a ‘p’ or ‘g’ character; if it’s False, it won’t.

The returned mask can be as large as 8+MaxDigits characters long, not counting any space set aside for commas (MaxDigits-1 / 3 characters). This is because a currency symbol can be as long as 5 characters, and there is usually a decimal point with two digits on its right as well. (In some cases, a ‘p’ or ‘g’ character can appear in the mask as well, but in these cases the currency symbol is never 5 characters long.) Keep this in mind when laying out your forms. If your program is going to change the picture mask based on the result of InternationalCurrency, leave room for the largest picture mask that you might possibly receive.

The following example returns “$###,###.##” for the U.S. (e.g., “$-12,345.67”), “###,###.##CCC” for Iceland (e.g., “-12.345,67 kr”), “$###,###.##p” for Mexico (e.g., “($12,345.67)”):

```
var
  S : string;
...
S := OvcIntlSup.InternationalCurrency('#', 6, True, True, True);
```

The following example returns “c####.##” for the U.S. (e.g., “$ -12.34”), “cccc####.##” for Switzerland (e.g., “Fr. -12.34”), “####.##ccccc” for Portugal (e.g., “1234,56 Esc.”):

```
var
  S : string;
...
S := OvcIntlSup.InternationalCurrency('#', 4, False, False,
False);
```

See also: CurrencyLtStr, CurrencyRtStr, CurrencyDigits

InternationalDate method

```
function InternationalDate(ForceCentury : Boolean) : string;
```

↳ Returns a picture mask for a short date string.

This method uses the international short date information to construct a picture mask for a date string. The mask it returns can be assigned to the PictureMask property of a TOvcPictureField component.

The ForceCentury parameter indicates whether the picture mask should leave room for a century (yyyy) as well as a year (yy), regardless of the preferences indicated in WIN.INI. The constructed picture mask is returned as the function result.

The following example returns “MM/DD/yy” for the U.S. (e.g., “ 4/ 9/92”), “DD/MM/yyyy” for Finland (e.g., “09.04.1992”), “dd/mm/yy” for Denmark (e.g., “09-04-92”):

```
var
  S : string;
...
S := OvcIntlSup.InternationalDate(False);
```

See also: InternationalCurrency, InternationalTime, InternationalLongDate

```
function InternationalLongDate(
  ShortNames : Boolean; ExcludeDOW : Boolean) : string;
```

Returns a picture mask for a long date string.

This method uses the international long date information to construct a picture mask for a date string. This long date format is rarely used in data entry forms.

ShortNames indicates whether the day of week and month names, if any, should appear in shortened form (Mon instead of Monday, Jan instead of January, etc.). The ExcludeDOW parameter indicates whether the picture mask should omit the day of the week, regardless of the preferences indicated by the Windows international settings. The constructed picture mask is returned as the function result.

The following example returns “wwwwwwwww, nnnnnnnnn dd, yyyy” for the U.S. (e.g., “Thursday , March 26, 1992”):

```
  var
    S : string;
  ...
  S := OvcIntlSup.InternationalLongDate(False, False);
```

The following example returns “www, nnn dd, yyyy” for the U.S. (e.g., “Thu, Mar 26, 1992”):

```
  var
    S : string;
  ...
  S := OvcIntlSup.InternationalLongDate(True, False);
```

The following example returns “nnn dd, yyyy” for the U.S. (e.g., “Mar 26, 1992”):

```
  var
    S : string;
  ...
  S := OvcIntlSup.InternationalLongDate(True, True);
```

See also: InternationalCurrency, InternationalDate, InternationalTime

InternationalTime method

```
function InternationalTime(ShowSeconds : Boolean) : string;
```

↳ Returns a picture mask for a time string.

This method uses the international time information to construct a picture mask for a time string or an entry field containing a time. The mask it returns as the function result can be assigned to the PictureMask property of a TOvcPictureField component.

The ShowSeconds parameter indicates whether the mask should include seconds as well as minutes and hours.

The following example returns “HH:mm:ss tt” for the U.S. (e.g., “ 1:40:21 PM”), “Hh:mm:ss” for Italy (e.g., “13.40.21”), “hh:mm:ss” for Switzerland (e.g., “13,40,21”):

```
  var
    S : string;
  ...
  S := OvcIntlSup.InternationalTime(True);
```

See also: InternationalCurrency, InternationalDate, InternationalLongDate

ListChar run-time, read-only property

```
property ListChar : AnsiChar
```

↳ Returns the character used to separate lists of items.

ListChar is initialized to the “List separator” value defined by Windows.

MonthStringToMonth method

```
function MonthStringToMonth(
  const S : string; Width : Byte) : Byte;
```

↳ Converts a month string to a month number (1-12).

This method returns the month number that corresponds to the month name specified in S. Width indicates the number of characters to use when comparing S with the internal array of month names. The comparison is not case sensitive.

The following example returns 6:

```
  var
    M : Byte;
  ...
  M := OvcIntlSup.MonthStringToMonth('Jun', 3);
```

See also: MonthToString

MonthToString method

```
function MonthToString(Month : Integer) : string;
```

Returns the month as a string.

This method returns the name of the specified month as the function result. By default, the English name for the month is returned.

The following example returns “March”:

```
var
  S : string;
...
S := OvcIntlSup.MonthToString(3);
```

See also: MonthStringToMonth

NoChar run-time property

```
property NoChar : AnsiChar
```

Default: ‘N’

The character that indicates a False condition for Yes or No fields.

See also: YesChar

OnWinIniChange run-time event

```
property OnWinIniChange : TNotifyEvent
```

Defines an event handler that is called when Windows broadcasts the WM_WININICHANGE message.

The method assigned to the OnWinIniChange event is called when Windows informs the TOvcIntlSup object that one or more Windows settings changed. This occurs only if the AutoUpdate property is set to True. If the AutoUpdate property is False, the TOvcIntlSup object does not create a window handle and therefore can not receive any window messages.

This event provides you an opportunity to inspect and possibly change any properties that might have been updated because of the changes. This event also provides you an opportunity to save the existing values of affected entry fields, call ResetInternationalInfo, then restore the values which will then be displayed using the new international settings.

Caution: If an event handler is assigned to this event and AutoUpdate is True, it is assumed that ResetInternationalInfo will be called within your event handler to update the international settings.

TNotifyEvent is defined in the Classes unit.

ResetInternationalInfo method

```
procedure ResetInternationalInfo;
```

Resets international property values.

This method updates all property values to match the current settings in WIN.INI.

SlashChar run-time property

```
property SlashChar : AnsiChar
```

The character used to separate string date values.

TimeStringToHMS method

```
function TimeStringToHMS(
  const Picture, S : string;
  var Hour, Minute, Second : Integer) : Boolean;
```

Extracts hours, minutes, seconds from a time string.

Given a string representation of a time (S), this method converts it to Hour, Minute, and Second. If S is not a string of the same form as Picture, the results are unpredictable. If S is the correct form, but the date represented by S is invalid, the function returns False.

The following example returns True, with Hour = 1, Minute = 2, and Second = 3:

```
  OvcIntlSup.TimeStringToHMS(
    'hh:mm:ss', '01:02:03', Hour, Minute, Second);
```

See also: TimeStringToTime, TimeToAmPmString, TimeToTimeString

TimeStringToTime method

```
function TimeStringToTime(
  const Picture, S : string) : TStTime;
```

Converts a time string to a time variable.

Given a string representation of a time (S), this method converts it to a time variable, returned as the function result. If S is not a string of the same form as Picture, the results are unpredictable. If S is the correct form, but the date represented by S is invalid, BadTime is returned.

The following example returns the time value representing 1:02:03 am:

```
var
  T : TStTime;
...
T := OvcIntlSup.TimeStringToTime('hh:mm:ss', '01:02:03');
```

See also: TimeStringToHMS, TimeToAmPmString, TimeToTimeString

TimeToAmPmString method

```
function  TimeToAmPmString(
  const Picture : string; T : TStTime; Pack : Boolean) : string;
```

Converts a time variable to a time string in am/pm format.

This method is identical to TimeToTimeString except that it forces the time designated by T to be represented in am/pm format, even if the Picture mask does not require it. If T equals BadTime, the hour, minutes, and seconds are represented as spaces in the returned string. If Pack is True, TimeToAmPmString compresses non-literal spaces in the returned string.

The following example sets S1 = '12:00:00', S2 = '00:00:00', and S3 = " : : ":

```
var
  S1 : string;
  S2 : string;
  S3 : string;
...
S1 := OvcIntlSup.TimeToAmPmString('hh:mm:ss', 0, False);
S2 := OvcIntlSup.TimeToTimeString('hh:mm:ss', 0, False);
S3 := OvcIntlSup.TimeToAmPmString('hh:mm:ss', BadTime, False);
```

See also: TimeToTimeString

TimeToTimeString method

```
function TimeToTimeString(
  const Picture : string; T : TStTime; Pack : Boolean) : string;
```

Converts a time variable to a time string.

Given a time of day (T), this routine converts it to a string of the form specified by Picture and returns it as the function result. If T equals BadTime, the hour, minutes, and seconds are represented as spaces in the returned string. If Pack is True, TimeToTimeString compresses non-literal spaces in the returned string.

The following example returns a string representation of the time value contained in MyTimeVar:

```
var
  S : string;
...
S := OvcIntlSup.TimeToTimeString('hh:mm:ss', MyTimeVar, False);
```

See also: TimeStringToHMS, TimeStringToTime, TimeToAmPmString

TrueChar — run-time property

```
property TrueChar : AnsiChar
```

Default: ‘T’

↳ The character that indicates a True condition for True/False fields.

See also: FalseChar

YesChar — run-time property

```
property YesChar : AnsiChar
```

Default: ‘Y’

↳ The character that indicates a True condition for Yes/No fields.

See also: NoChar

10

Chapter 11: Validated Data Entry Components

New in Orpheus 4 is the TO32FlexEdit, O32*Validators, and the O32ValidatorPool. They represent the future of Orpheus data-entry as we transition away from dependency on the OvcController. For more information on the Validators see page 453, ValidatorPool see page 468, and FlexEdit see page 437.

Orpheus entry field components provide true field-by-field validation, range checking, and the ability to postpone validation. With easy-to-use picture masks, entry fields can edit and validate data in virtually any imaginable format.

Orpheus entry fields provide fully automatic support for international data entry and display. This includes currency values, date and time formats, displaying fractional values, and much more. This feature can be disabled or customized for each field.

In addition to the default validation that tests entered values to make sure they are within the limits of the field's data type, upper and lower range limits can be specified for nearly all field types. Additionally, special methods and events are provided to allow you to perform custom validation and custom error handling.

This chapter describes the two classes that provide the base functions for entry field components:

- TOvcBaseEntryField
- TOvcBasePictureField

It also describes the three high level components that are derived from the base classes. These are the components to use for all your data entry needs:

- TOvcSimpleField
- TOvcPictureField
- TOvcNumericField

TOvcSimpleField implements the Orpheus simple entry field, which is similar to the Windows standard edit control and the standard VCL TEdit component. It is limited to a single line and its picture mask is one character. The mask character indicates the set of characters that are allowed.

The picture entry field, which is implemented in TOvcPictureField, gives you more flexibility in format. It allows different mask characters for each character position in the field. This also allows you to specify literal characters in the entry field.

The numeric entry field, which is implemented in TOvcNumericField, is very similar to a picture entry field. The difference is that a numeric entry field gives you a calculator-style editor where numbers are entered in the field from right to left.

A data-aware version of each entry field type is also provided. The TOvcDbSimpleField, TOvcDbPictureField, and TOvcDbNumericField can be connected to a data source to allow editing of database fields.

TOvcBaseEntryField Class

The TOvcBaseEntryField class is an abstract class derived from TOvcCustomControlEx. It provides the core functionality for the higher-level entry field components. The functions provided by this class implement editing commands that are used in all of the derived entry field components.

Entry field commands

The following commands are available in any Orpheus entry field component. The commands and key assignments described here are available when you use the "Default" or "WordStar" command tables (several command tables can be used at the same time). See "TOvcCommandProcessor Class" on page 50 for information on how to customize the command tables.

Many of the commands mentioned here work differently when used in a numeric entry field. The reason for the difference in behavior is that there is no caret movement in a numeric field. The caret always remains in a fixed position at the right of the field contents. The differences are noted inTable 11.1

Table 11.1: *Entry field commands*

Command	**Default**	**WordStar**	**Description**
ccBack	<BkSp>	<CtrlH>	Delete the character to the left of the caret.
ccCompleteDate	Not assigned.	Not assigned.	Insert the current day, month, or year (depending on which is selected) in a date subfield.
ccCompleteTime	Not assigned.	Not assigned.	Insert the current hour, minute, or second (depend ing on which is selected) in a time subfield.
ccCopy	<CtrlIns>, <CtrlC>	Not assigned.	Copy the selected text to the clipboard.

11

Table 11.1: *Entry field commands (continued)*

Command	**Default**	**WordStar**	**Description**
ccCtrlChar	Not assigned.	<CtrlP>	Insert a control character. For example, to insert a <CtrlG>, enter <CtrlP><CtrlG>. This command does nothing if the field's mask does not allow control characters.
ccCut	<ShiftDel>, <CtrlX>	Not assigned.	Delete the selected text after copying it to the clipboard.
ccDec	Not assigned.	Not assigned.	Decrement the value of the field or subfield.
ccDel	<Del>	<CtrlG>	Delete the character at the caret.
ccDelBol	Not assigned.	Not assigned.	Delete from the beginning of the subfield to the caret. This command does nothing in a numeric entry field.
ccDelEol	Not assigned.	<ShiftCtrlY>, <CtrlQ><Y>	Delete from the caret to the end of the current sub field. This command does nothing in a numeric entry field.
ccDelLine	Not assigned.	<CtrlY>	Clear the entire field.
ccDelWord	Not assigned.	<CtrlT>	Delete the word to the right of the caret. This com mand does nothing in a numeric entry field.
ccDown	<Down>	<CtrlX>	Move the caret down to the next component.

11

Table 11.1: *Entry field commands (continued)*

Command	Default	WordStar	Description
ccEnd	<End>	<CtrlQ><D>	Move the caret to the end of the field. This command does nothing in a numeric entry field.
ccExtendEnd	<ShiftEnd>	Not assigned.	Extend the selection to the end of the field.
ccExtendHome	<ShiftHome>	Not assigned.	Extend the selection to the start of the field.
ccExtendLeft	<ShiftLeft>	Not assigned.	Extend the selection to the left by one character.
ccExtendRight	<ShiftRight>	Not assigned.	Extend the selection to the right by one character.
ccExtWordLeft	<CtrlShiftLeft>	Not assigned.	Extend the selection to the left by one word.
ccExtWordRight	<CtrlShiftRight>	Not assigned.	Extend the selection to the right by one word.
ccHome	<Home>	<CtrlQ><S>	Move the caret to the beginning of the field. This command does nothing in a numeric entry field.
ccInc	Not assigned.	Not assigned.	Increment the value of the field or subfield.

Table 11.1: *Entry field commands (continued)*

Command	**Default**	**WordStar**	**Description**
ccIns	<Ins>	<CtrlV>	Toggle insert mode. The caret reflects the current mode. By default, a solid line indicates insert mode; a block indicates overwrite mode. Although this command is recognized by numeric entry fields, it doesn't do anything except change the caret.
ccLeft	<Left>	<CtrlS>	Move the caret left one character. If the caret is already at the beginning of the field (or the field is a numeric entry field) and AutoAdvanceLeftRight is True, the caret is moved to the previous compo nent, just as it is if <ShiftTab> is pressed.
ccNone	<.>	Never assigned.	In a picture entry field that contains a decimal point, this command moves the caret to the position just beyond the decimal point (if it isn't beyond the decimal point already).
ccPaste	<ShiftIns>, <CtrlV>	Not assigned.	Paste the text from the clipboard into the field.

11

Table 11.1: *Entry field commands (continued)*

Command	Default	WordStar	Description
ccRestore	<AltBkSp>, <CtrlZ>	<CtrlQ><L>	Restore the original contents of the field (i.e., the value the field contained when it last received the focus).
ccRight	<Right>	<CtrlD>	Move the caret right one character. If the caret is already at the end of the field (or the field is a numeric entry field) and the AutoAdvanceLeftRight property is True, the caret is moved to the next component, just as it is if <Tab> is pressed.
ccUp	<Up>	<CtrlE>	Move the caret to the previous component.
ccWordLeft	<CtrlLeft>	<CtrlA>	Move the caret left one word. This command does nothing in a numeric entry field.
ccWordRight	<CtrlRight>	<CtrlF>	Move the caret right one word. This command does nothing in a numeric entry field.

Marking, cutting, copying, and pasting text

The commands for marking, cutting, copying, and pasting text are handled differently by Orpheus than they are by the standard VCL edit components. Although these differences are not major, you should be aware of them. However, it should not be necessary for your users to be aware of these differences. Most of the cases described here are not likely to be encountered in ordinary day-to-day use of your applications. They will generally arise only during testing.

Simple entry fields

The Orpheus simple entry field handles these commands very much like the standard VCL edit components. Text can be selected in three ways:

- Click and drag the mouse cursor to mark the desired text.
- Double-click the mouse on a word to mark it.
- Hold down the shift key while executing a cursor command.

Once text is selected, it can be cut to the clipboard by pressing the keys assigned to the ccCut command or copied to the clipboard with the ccCopy command-key sequence. Text already in the clipboard can be pasted into the field with the ccPaste command-key sequence.

The only difference between the simple entry field and the standard VCL edit component is that the simple entry field must make pasted text conform to the restrictions imposed by the field's mask character. If the mask character is '!' (allows any character but forces upper case), then any text pasted into the field is converted to upper case. If the mask character is '9' (allows spaces and digits only), any alphabetic characters in the clipboard are rejected. For example, if the clipboard contains "A9B3", only "93" is pasted into the field.

Picture entry fields

Picture entry fields present new complications because the field's mask can contain literal characters. A complication arises when marking text because when you extend the highlight in an edit component, the caret should move to the next character beyond the highlight. However, if the field's mask contains literal characters, there are places in the field where the caret cannot go but the highlight can. Orpheus deals with this situation by allowing the highlight to be extended to cover any character in the field, and moving the caret to the appropriate place if at all possible.

A complication also arises when pasting text, because the algorithm used for the simple entry field (insert character if you can, reject it if you can't, advance caret if character was inserted) doesn't work if the mask contains literals. Orpheus implements a complex algorithm that looks both at the field's mask and at the text to be pasted and tries to determine whether to accept the character, reject it, or treat it as a literal. The goal is to avoid the case in which executing the cut and paste commands in succession will change the contents of the field.

An additional complication arises when the field's mask contains a decimal point. Suppose, for example, that the mask is "999.99", the entire field is highlighted, the clipboard contains "1.23", and the user enters the ccPaste command-key sequence. The expected result is " 1.23", and that's what you get, because when Orpheus tries to paste in the dot, it just moves the caret beyond the decimal point. Now suppose that the clipboard contains "1234.56." In this case, the caret is already beyond the decimal point when Orpheus tries to paste in the dot. If it used the same rule that it used in the first case, the result would be "123.56." It doesn't. If the caret is already beyond the decimal point, a dot is ignored, so the result in this case is "123.45."

Numeric entry fields

The numeric entry field faces similar complications when marking text. Since the caret cannot be moved, the contents of the entire field are highlighted when selecting with the mouse or when using the cursor keys rather than one character at a time.

Another complication arises when pasting text into a numeric field in which parentheses are used to represent negative numbers. In this case Orpheus treats '(' as equivalent to '-' when pasting text.

Hierarchy

TCustomControl (VCL)

    ❶ TOvcCustomControl (OvcBase) 16

        ❷ TOvcCustomControlEx (OvcBase) 19

            TOvcBaseEntryField (OvcEF)

Properties

- ❶ About
- AsBoolean
- AsCents
- AsDateTime
- AsExtended
- AsFloat
- AsInteger
- AsOvcDate
- AsOvcTime
- AsStDate
- AsString
- AsStTime
- AsVariant
- ❶ AttachedLabel
- AutoSize
- BorderStyle
- CaretIns
- CaretOvr
- ControlCharColor
- Controller
- CurrentPos
- DataSize
- DecimalPlaces
- DisplayString
- EditString
- EFColors
- Epoch
- EverModified
- IntlSupport
- ❶ LabelInfo
- LastError
- MaxLength
- Modified
- PadChar
- PasswordChar
- RangeHi
- RangeLo
- SelectedText
- SelectionLength
- SelectionStart
- TextMargin
- Uninitialized
- UserData
- ZeroDisplay
- ZeroDisplayValue

Methods

- ClearContents
- ClearSelection
- CopyToClipboard
- CutToClipboard
- DecreaseValue
- Deselect
- GetStrippedEditString
- GetValue
- IncreaseValue
- IsValid
- MergeWithPicture
- MoveCaret
- MoveCaretToEnd
- MoveCaretToStart
- PasteFromClipboard
- ProcessCommand
- ResetCaret
- Restore
- SelectAll
- SetInitialValue
- SetRangeHi
- SetRangeLo
- SetSelection
- SetValue
- ValidateContents
- ValidateSelf

Events

- ❶ AfterEnter
- ❶ AfterExit
- OnChange
- OnGetEpoch
- OnError
- ❶ OnMouseWheel
- OnUserCommand
- OnUserValidation

Reference Section

AsBoolean — run-time property

```
property AsBoolean : Boolean
```

↳ Determines the value of a Boolean entry field component.

If you use the AsBoolean property with a field that does not have a Boolean data type, an EInvalidDataType exception is raised.

The following example shows testing and setting the Boolean value of an entry field:

```
  if Field1.AsBoolean then
    {do something based on the field value being true}

  Field1.AsBoolean := False; {set Field's value to False}
```

See also: GetValue, SetValue

AsCents — run-time property

```
property AsCents : LongInt
```

↳ Determines the value of an entry field editing a floating-point data type.

The AsCents property provides a simple conversion to and from a whole number value (representing pennies) for editing by an entry field with a data type of Real, Double, Single, or Extended. The entry field does this by dividing the assigned value by 100 when the field value is set using AsCents and by multiplying the field value by 100 when the field contents are obtained using AsCents.

See also: GetValue, SetValue

AsDateTime — run-time property

```
property AsDateTime : TDateTime
```

↳ Determines the value of a Date or Time picture field component.

If you use the AsDateTime property with a field that does not have a Date or Time data type, an EInvalidDataType exception is raised.

The following example assigns the current date to Field1 and the current time to Field2:

```
  Field1.AsDateTime := SysUtils.Date;
  Field2.AsDateTime := SysUtils.Time;
```

See also: GetValue, SetValue

AsExtended run-time property

```
property AsExtended : Extended
```

↳ Determines the value of a field with an extended data type.

AsExtended allows you to set and retrieve the contents of an Extended or Comp field. If you use AsExtended with a field that has any other data type, an EInvalidDataType exception is raised.

See also: GetValue, SetValue

AsFloat run-time property

```
property AsFloat : Double
```

↳ Determines the value of a field with a floating point data type.

AsFloat allows you to set and retrieve the contents of a Real, Double, or Single field. If you use AsFloat with a field that has any other data type, an EInvalidDataType exception is raised.

The following example uses AsFloat to assign a value to a field:

```
  Field1.AsFloat := 5.34;
```

See also: GetValue, SetValue

AsInteger run-time property

```
property AsInteger : LongInt
```

↳ Determines the value of a field with a whole number data type.

AsInteger allows you to set and retrieve the contents of an Integer, LongInt, ShortInt, Byte, or Word field. If you use AsInteger with a field that does not have a whole number data type, an EInvalidDataType exception is raised.

The following example uses AsInteger to assign a value to a field:

```
  Field1.AsInteger := 30;
```

See also: GetValue, SetValue

AsOvcDate run-time property

```
property AsOvcDate : TStDate
```

↳ Determines the value of a date picture field component.

If you use the AsOvcDate property with a field that does not have a TStDate data type, an EInvalidDataType exception is raised. This property is provided for compatibility with previous versions of Orpheus.

The following example assigns an Orpheus date value to Field1:

```
  Field1.AsOvcDate := StDate.CurrentDate;
```

See also: StDate.CurrentDate, GetValue, SetValue

AsOvcTime run-time property

```
property AsOvcTime : TStTime
```

↳ Determines the value of a time picture field component.

If you use the AsOvcTime property with a field that does not have a TStTime data type, an EInvalidDataType exception is raised. This property is provided for compatibility with previous versions of Orpheus.

The following example assigns an Orpheus time value to Field1:

```
  Field1.AsOvcTime := StDate.CurrentTime;
```

See also: StDate.CurrentTime, GetValue, SetValue

AsStDate run-time property

```
property AsStDate : TStDate
```

↳ Determines the value of a date picture field component.

If you use the AsStDate property with a field that does not have a TStDate data type, an EInvalidDataType exception is raised.

The following example assigns an Orpheus date value to Field1:

```
  Field1.AsStDate := StDate.CurrentDate;
```

See also: GetValue, SetValue, StDate.CurrentDate

AsString run-time property

```
property AsString : string
```

Determines the value of any Orpheus entry field component.

The returned string represents the value displayed by the entry field component. If the TrimBlanks property is True, leading and trailing blanks are removed before the value is returned. Use EditString to obtain the display string without modifications. Or, use GetStrippedEditString to obtain the edit string with all mask characters removed.

The string value assigned to the AsString property must be convertible into the appropriate field data type. For example, assigning "ABC" to a field with an Integer data type is invalid. If the value assigned to this property is invalid, unpredictable display values will result.

The following example assumes that Field1 is an Integer data type:

```
  Field1.AsString := '345';
```

See also: EditString, GetStrippedEditString, GetValue, SetValue

AsStTime run-time property

```
property AsStTime : TStTime
```

Determines the value of a time picture field component.

If you use the AsStTime property with a field that does not have a TStTime data type, an EInvalidDataType exception is raised.

The following example assigns an Orpheus time value to Field1:

```
  Field1.AsStTime := StDate.CurrentTime;
```

See also: StDate.CurrentTime, GetValue, SetValue

AsVariant run-time property

```
property AsVariant : string
```

Determines the value of any Orpheus entry field component.

The returned value is automatically converted to match the type of the variable that it is being assigned (see the Variant data type for additional information). The value assigned to the AsVariant property must be convertible into the appropriate field data type. For example, assigning "ABC" to a field with an Integer data type is invalid. If the value assigned to AsVariant is invalid, the display values are unpredictable.

See also: GetValue, SetValue

AutoSize property

```
property AutoSize : Boolean
```

Default: True

↳ Determines whether the entry field component should resize its height when the font changes.

If AutoSize is True, the height of the component is adjusted so that the font is completely visible.

BorderStyle property

```
property BorderStyle : TBorderStyle
```

Default: bsSingle

↳ Determines the style used when drawing the border.

The possible values are bsSingle (a single line border is drawn around the component) or bsNone (no border line).

The TBorderStyle type is defined in the VCL's Controls unit.

CaretIns property

```
property CaretIns : TOvcCaretType
```

Default: Vertical line caret

↳ Determines the characteristics of the editing caret.

This property allows you to customize the attributes of the editing caret used while the edit field is in insert mode. For additional information see “TOvcCaret Class” on page 1181.

See also: CaretOvr

CaretOvr property

```
property CaretOvr : TOvcCaretType
```

Default: Block caret

↳ Determines the characteristics of the editing caret.

This property allows you to customize the attributes of the editing caret used while the edit field is in overwrite mode. For additional information see “TOvcCaret Class” on page 1181.

See also: CaretIns

ClearContents method

```
procedure ClearContents;
```

↳ Deletes the field’s entire display string.

Literal characters, such as date and time separators, are not deleted.

See also: ClearSelection

ClearSelection method

```
procedure ClearSelection;
```

↳ Deletes the selected portion of the display string.

Literal characters, such as date and time separators, are not deleted. If no text is selected, ClearSelection does nothing.

See also: ClearContents

ControlCharColor property

```
property ControlCharColor : TColor
```

Default: clRed

↳ The color used to display control characters in an entry field.

Controller property

```
property Controller : TOvcController
```

↳ Allows access to the properties of the TOvcController non-visual component attached to the entry field.

This property provides access to a common command processor that interprets all keyboard input, a central error handler that is used if a field-specific error handler is not assigned, and additional properties that allow the entry field to ignore field errors if the focus is moving to a Help, Cancel, or Restore button.

All entry fields must have a controller assigned and in most cases this is done automatically. If the Controller is not assigned, an ENoControllerAssigned exception is raised at run time. See “TOvcController Component” on page 28 for additional information.

CopyToClipboard method

```
procedure CopyToClipboard;
```

Copies the selected portion of the display string to the Windows clipboard.

CopyToClipboard erases the existing contents of the clipboard. If no text is selected, calling CopyToClipboard has no effect and the existing clipboard contents are left intact. See "Marking, cutting, copying, and pasting text" on page 368 for more information.

See also: CutToClipboard, PasteFromClipboard

CurrentPos run-time property

```
property CurrentPos : Integer
```

Determines the position of the caret within the entry field component at run time.

Valid values are 0 (the start of the field) through the length of the current display string. If a value less than 0 is assigned to CurrentPos, the caret is placed at the start of the entry field. If a value greater than the length of the display string is assigned to CurrentPos, the caret is positioned just past the last character.

If the requested position is at a literal or semi-literal, the caret is placed at the next position that is not a literal or semi-literal. If the field does not have the focus, assignments to CurrentPos are ignored and -1 is returned.

The following examples show how to position the caret at the beginning and at the end of a field:

```
  {set the position of the caret to the beginning of the field}
  Field1.CurrentPos := 0;
  {set the position of the caret to the end of the display string}
  Field1.CurrentPos := MaxInt;
```

CutToClipboard method

```
procedure CutToClipboard;
```

Copies the selection to the Windows clipboard and then deletes the selected text.

CutToClipboard erases the existing contents of the clipboard. If no text is selected, calling CutToClipboard has no effect and the existing clipboard contents are left intact. See "Marking, cutting, copying, and pasting text" on page 368 for more information.

See also: CopyToClipboard, PasteFromClipboard

DataSize run-time property

```
property DataSize : Word
```

↳ Returns the number of bytes required to store the specified data type.

The data type is determined by the DataType property. For example, if the field's data type is Byte, DataSize returns 1 and if the data type is LongInt, DataSize returns 4.

See also: DataType

DecimalPlaces property

```
property DecimalPlaces : Byte
```

Default: 0

↳ The number of digits that are displayed to the right of the decimal point.

This property can be used in the standard and data-aware picture entry fields to set a decimal position that will display a fractional portion of the value only if one exists. Otherwise setting the decimal position in the picture mask of a picture entry field causes the decimal point to always be displayed.

DecimalPlaces is useful only for simple and picture fields configured to edit floating-point data types and is not used by others.

DecreaseValue method

```
procedure DecreaseValue(Wrap : Boolean; Delta : Double);
```

↳ Decrements the field value by the amount specified in Delta.

If Wrap is True and the reduced field value would be below the field's allowable lower limit, the new value is set to the field's upper limit. DecreaseValue has no effect for string field types.

The following example shows how DecreaseValue wraps a field value:

```
  {Field1 is a byte field and its current value is 0}
  {using default range settings}
  Field1.DecreaseValue(True, 1);
  {new display string shows "255"}
  ...
  {Field2 is a byte field and its current value is 50}
  {lower range is set to 40 and upper range is 60}
  Field2.DecreaseValue(True, 20);
  {new display string shows "60"}
```

See also: IncreaseValue

Deselect — method

```
procedure Deselect;
```

↳ Removes the highlight for selected text.

See also: ClearContents, ClearSelection

DisplayString — run-time, read-only property

```
property DisplayString : string
```

↳ Returns the string displayed in the entry field.

This read-only property provides a copy of the string displayed by the entry field. The display string differs from the edit string in special cases such as when the PasswordMode property is True.

See also: EditString, TOvcPictureField.PasswordMode, TOvcSimpleField.PasswordMode

EditString — run-time, read-only property

```
property EditString : string
```

↳ Returns a copy of the entry field's edit string.

The returned string contains all mask characters and any leading and trailing spaces.

The following example gets a copy of the display string in Field1:

```
  S := Field1.EditString;
```

See also: AsString, GetStrippedEditString

EFColors — property

```
property EFColors : TOvcEfColors
```

↳ Determines several color values for the entry field.

See “TOvcEfColors Class” on page 434 for additional information.

Epoch property

```
property Epoch : Integer
```

Default : 0

↳ Specifies the starting year of a 100 year period.

The default of zero means that the value specified in Controller.Epoch is used. See “The Epoch property” on page 321 for more information.

EverModified run-time property

```
property EverModified : Boolean
```

↳ Determines the modified status of the entry field component.

The EverModified state of an entry field component is cleared when a value is assigned to the field. This property differs from the Modified property in that once EverModified becomes True, it remains True until the field value is changed by some action in your program.

When EverModified is set to False, the Modified property is also set to False.

The following example checks to see if Field1 has been modified since the form was displayed:

```
  if Field1.EverModified then begin
    {field value has changed, save the contents to a file}
    ...
  end;
```

See also: Modified

GetStrippedEditString virtual method

```
function GetStrippedEditString : string; virtual;
```

↳ Returns the edit string, stripped of all literal and semi-literal characters.

For simple fields, this simply returns the edit string. Descendent entry field classes override this method to strip literal and semi-literal characters from the display string.

See also: GetValue

GetValue method

```
function GetValue(var Data) : Word;
```

Retrieves the value of a field directly into a variable of the appropriate data type.

Data is an untyped variable parameter that must match the data type of the entry field. You must ensure that they match because GetValue does not.

GetValue forces a validation of the field contents. The function result is an error code if the field fails the validation, or zero if there is no error.

The following example assumes that Field1 is a simple real field:

```
var
  R : Real;
  I : Integer;
begin
...
  if Field1.GetValue(R) = 0 then
    {return value is valid--do something with it}
...
  {*** the following is incorrect
       and will cause a memory overwrite ***}
  if Field1.GetValue(I) = 0 then ...
  {*** I is not the same data type as the field ***}
```

See also: TOvcNumericField.DataType, TOvcPictureField.DataType,
TOvcSimpleField.DataType, SetValue

IncreaseValue method

```
procedure IncreaseValue(Wrap : Boolean; Delta : Double);
```

Increments the field value by the amount specified in Delta.

If Wrap is True and the increased field value would be above the field’s allowable upper limit, the new value is set to the field’s lower limit.

IncreaseValue has no effect for string field types.

The following example shows how IncreaseValue wraps a field value:

```
{Field1 is a byte field and its current value is 255}
{using default range settings}
Field1.IncreaseValue(True, 1);
{new display string shows "0"}
...
{Field2 is a byte field and its current value is 50}
{lower range is set to 40 and upper range is 60}
Field2.IncreaseValue(True, 20);
{new display string shows "40"}
```

See also: DecreaseValue

IntlSupport run-time property

```
property IntlSupport : TOvcIntlSup
```

Default: OvcIntlSup

↳ Provides access to the international support object that is attached to the entry field component.

A global international support object is created during initialization and all entry field components use it by default. You can create an additional TOvcIntlSup object, tailor the international settings (see "Chapter 10: International Support Issues" on page 339 for details), and attach it to the entry field component by assigning it to this property.

The following example creates a new international support object and attaches it to Field1. When the field needs to display the comma character, it uses a dot instead of the character specified by the operating system.

```
var
  MyIntlSupport : TOvcIntlSup;
...
  MyIntlSupport := TOvcIntlSup.Create;
  {turn off auto update flag so changes to WIN.INI}
  {will not trigger resetting of our changes}
  MyIntlSupport.AutoUpdate := False;
  {assign a new comma character}
  MyIntlSupport.CommaChar := '.';
...
  {attach the new international support object to our field}
  Field1.IntlSup := MyIntlSupport;
```

IsValid method

```
function IsValid : Boolean;
```

↳ Determines whether a field passed its last validation test.

IsValid returns True if the field passed its last validation test. Otherwise, it returns False. This method is meaningful only if the field is using the SoftValidation option. IsValid is useful when the form is about to be closed or when preparing to commit the data in a form to storage.

See also: SoftValidation

LastError run-time property

```
property LastError : Word
```

↳ The result of the last field validation.

LastError is initialized to zero when the field receives focus, and set appropriately when the entry field is validated.

See OnError for a description of the possible error values.

See also: OnError, ValidateSelf

MaxLength property

```
property MaxLength : Word
```

Default: dependent on the field's data type

↳ The maximum number of characters that can be entered in the field.

MaxLength determines the logical width (in characters) of the entry field. This value does not change the physical width of the entry field. It limits the number of characters that are accepted.

The MaxLength property is not used by numeric entry fields.

MergeWithPicture virtual method

```
procedure MergeWithPicture(const S : string); virtual;
```

↳ Merges a string with a field's picture mask.

MergeWithPicture combines the string specified by S with the field's picture mask and updates the edit string.

The following example uses a picture mask of “(999) 999-999.” The result is “(800) 333-4160” as the entry field’s edit string.

```
  Field1.MergeWithPicture('8003334160');
```

See also: TOvcNumericField.PictureMask, TOvcPictureField.PictureMask, and TOvcSimpleField.PictureMask

Modified run-time property

```
property Modified : Boolean
```

Determines the modified state of the entry field component.

The modified status of an entry field component is initially False when the field receives the focus. It is changed to True when the field is changed. This property is most useful in a post-edit routine (see the OnPostEdit event of the TOvcController Component” on page 35) when deciding whether to take an action that is necessary only if the field changed.

When Modified is set to True, EverModified is set to True. When Modified is set to False, EverModified is not changed.

See also: EverModified

MoveCaret method

```
procedure MoveCaret(Delta : Integer);
```

Positions the editing caret in the edit field relative to its current position.

If Delta is positive, the editing caret is moved to the right by the number of character positions specified by Delta. If Delta is negative, the editing caret is moved to the left.

If Delta specifies a position before the beginning or after the end of the edit field, the caret is placed at the beginning or the end of the field respectively. If the new position of the caret is on a non-editable position (such as a literal), the caret is placed before (if Delta is negative) or after (if Delta is positive) the literal character.

See also: MoveCaretToEnd, MoveCaretToStart, ResetCaret

MoveCaretToEnd method

```
procedure MoveCaretToEnd;
```

Moves the caret to last editable position within the field.

See also: MoveCaret, MoveCaretToStart, ResetCaret

MoveCaretToStart — method

```
procedure MoveCaretToStart;
```

↳ Moves the caret to the first editable position within the field.

See also: MoveCaret, MoveCaretToEnd, ResetCaret

OnChange — event

```
property OnChange : TNotifyEvent
```

↳ Defines an event handler that is called when an entry field is edited.

The method assigned to the OnChange event is called when the edit field is modified (for example, each time a character is entered). The method is not called when a value is assigned to the field.

TNotifyEvent is defined in the VCL Classes unit.

OnError — event

```
property OnError : TValidationErrorEvent;

TValidationErrorEvent = procedure(Sender : TObject;
  ErrorCode : Word ErrorMessage : string) of object;
```

↳ Defines an event handler that is called when an entry field error occurs.

The method assigned to the OnError event is called when an error occurs. If no method is assigned to this event, the event handler assigned to the field's controller OnError event is called.

The possible values for ErrorCode are:

ErrorCode	**Value**
oeRangeError	The entered value is not within the accepted range.
oeInvalidNumber	An invalid number was entered in a numeric field.
oeRequiredField	A required field was left blank.
oeInvalidDate	An invalid date was entered in a date field.
oeInvalidTime	An invalid time was entered in a time field.

ErrorCode	Value
oeBlanksInField	A blank was left in a field or subfield that does not allow blanks. This error is reported only by the validation helper routines ValidateNoBlanks and ValidateSubfields in OVCVALID.
oePartialEntry	A partial entry was given in a field or subfield that must either be entirely empty or entirely full. This error is reported only by the validation helper routines ValidateNotPartial and ValidateSubfields in OVCVALID.
oeCustomError	The first error code reserved for user applications. All error values less than oeCustomError are reserved for use by Orpheus. oeCustomError is defined in the OvcData unit as 32768.

ErrorMessage is a short description of the error.

See “TOvcController Component” on page 28 for additional information about error handling for multiple entry fields.

OnUserCommand event

```
property OnUserCommand : TUserCommandEvent
```

```
TUserCommandEvent = procedure(
  Sender : TObject; Command : Word) of object;
```

Defines an event handler that is called when a user-defined command is entered.

The method assigned to the OnUserCommand event is called when you enter the key sequence corresponding to one of the user-defined commands (ccUser1, ccUser2, etc.).

For example, suppose you add the <CtrlD> key sequence to one of the active command tables and assign it to the ccUser1 command. When you press <CtrlD>, the method assigned to the OnUserCommand event is called and ccUser1 is passed in Command.

See “TOvcCommandProcessor Class” on page 50 for additional information about user-defined commands and command tables.

```
property OnUserValidation : TUserValidationEvent

TUserValidationEvent = procedure(
  Sender : TObject; var ErrorCode : Word) of object;
```

Defines an event handler that is called when field validation is required.

The method assigned to the OnUserValidation event is called after the default validation is performed on the field. Validation is performed when the field loses the focus and when the contents of the field are retrieved using the AsXxx properties or the GetValue method.

ErrorCode contains the result of the validation. If it is zero, then the default validation didn't find any errors and you can perform any additional validation that is required. The error constants are defined in the OnError event.

The validation helper routines can be used to perform special validation. See "Validation Helper Routines" on page 333 for more information. If the OnUserValidation event handler determines that the field is invalid, assign a value to ErrorCode before exiting the method. The error value should be greater than or equal to oeCustomError (32768). Note that attempts to change the field's value while user validation is in progress are ignored.

Caution: The event handler assigned to the OnUserValidation event must not perform any actions that could change the focus. This could cause Windows to lose track of where the focus is and cause other problems as well. Do not display a form or dialog reporting the error—that should only be done in a method assigned to the field's OnError event handler or the TOvcController OnError event handler. The OnUserValidation event should only do what is necessary to confirm that the field is valid.

See "TOvcController Component" on page 28 for additional information about error handling for multiple entry fields.

The following example displays an error message if the user enters 0:

```
uses
  ...
  OvcData;
const
  MyError1 = oeCustomError + 1;
...

procedure TForm1.APictureFieldUserValidation(
  Sender : TObject; var ErrorCode : Word);
begin
  if (Sender as TOvcPictureField).AsFloat = 0 then
    ErrorCode := MyError1;
end;

procedure TForm1.APictureFieldError(
  Sender : TObject; ErrorCode : Word; ErrorMsg : string);
begin
  case ErrorCode of
    MyError1 : ErrorMsg := 'Value cannot be 0';
    ...
  end;
  ShowMessage(ErrorMsg);
end;
```

See also: OnError

PadChar property

```
property PadChar : AnsiChar
```

Default: ASCII 32 (space)

↳ The character that is used to fill the blank portion of the display string.

The following example sets the pad character to an underscore. Field1 is a picture string field with a mask of “(999) 999-999”:

```
Field1.PadChar := '_';
```

The displayed field will be “(___) ___-____”.

PasswordChar property

```
property PasswordChar : AnsiChar
```

Default: ‘*’

↳ The character that is displayed instead of the entered text if the field is in password mode.

If the PasswordMode property is True, the character assigned to PasswordChar is displayed instead of the characters entered into the field.

See also: PasswordMode

PasteFromClipboard method

```
procedure PasteFromClipboard;
```

↳ Copies the contents of the Windows clipboard into the edit field.

See “Marking, cutting, copying, and pasting text” on page 368 for more information.

See also: CopyToClipboard, CutToClipboard

ProcessCommand method

```
procedure ProcessCommand(Cmd, CharCode : Word);
```

↳ Sends commands and characters to the entry field.

This method allows sending commands and characters to the entry field, just as if they were entered from the keyboard. Typical uses for this could include a simple macro playback facility.

See “TOvcCommandProcessor Class” on page 50 for more information.

RangeHi property

```
property RangeHi : string
```

Default: depends on the field’s data type

↳ The upper limit for the range of allowable field values.

This property is intended primarily for use at design time. However, it can be used at run time by assigning a string version of the desired range to the property. Range settings are ignored for string type fields.

You can directly enter a range limit in the RangeHi edit field in the Object Inspector or you can invoke the Range Property Editor, which allows entry and validation of the range limit into an Orpheus entry field of the appropriate data type.

The following example sets an upper limit of 200 for Field1:

```
{Field1 is a simple field with a data type of Byte}
Field1.RangeHi := '200';
```

See also: RangeLo, SetRangeHi, SetRangeLo

RangeLo property

```
property RangeLo : string
```

Default: depends on the field’s data type

↳ The lower limit for the range of allowable field values.

This property is intended primarily for use at design time. However, it can be used at run time by assigning a string version of the desired range to the property. Range settings are ignored for string type fields.

You can directly enter a range limit in the RangeLo edit field in the Object Inspector or you can invoke the Range Property Editor, which allows entry and validation of the range limit into an Orpheus entry field of the appropriate data type.

The following example sets a lower limit of 10 for Field1:

```
{Field1 is a simple field with a data type of Byte}
Field1.RangeLo := '10';
```

See also: RangeHi, SetRangeHi, SetRangeLo

ResetCaret method

```
procedure ResetCaret;
```

↳ Moves the caret to the beginning or end of the field.

Calling ResetCaret will place the caret at the first editable position when the value of the CaretToEnd is False and at the last editable position when CaretToEnd is True.

See also: CaretToEnd

Restore virtual method

```
procedure Restore; virtual;
```

↳ Returns the contents of the edit field to the value it contained when it last received the focus.

If you use the default command-key mapping table, this method is called when you enter <AltBackSpace> or <CtrlZ>.

SelectAll method

```
procedure SelectAll;
```

↳ Selects the entire contents of the edit field.

See also: ClearSelection, SelectedText, SelectionLength, SelectionStart, SetSelection

SelectedText run-time, read-only property

```
property SelectedText : string
```

↳ Returns a string containing the selected portion of the field's display string.

The field does not need to have the focus to use this property.

See also: ClearSelection, SelectAll, SelectionLength, SelectionStart, SetSelection

SelectionLength run-time property

```
property SelectionLength : Integer
```

↳ The length of the selected portion of the field's display string.

The valid range for SelectionLength is from 0 to the length of the field's display string. Leading and trailing blanks are highlighted if they are within the specified area. If the requested length exceeds the maximum length of the field, the selection is extended to the end of the field. The following example increases the selection by 2 characters:

```
  Field1.SelectionLength := Field1.SelectionLength + 2;
```

See also: ClearSelection, SelectAll, SelectedText, SelectionStart, SetSelection

SelectionStart run-time property

```
property SelectionStart : Integer
```

↳ The starting position of the selected portion of the field's display string.

The valid range for SelectionStart is from 0 to the length of the field's display string. If the starting position is greater than or equal to the ending position, no text is highlighted.

The following example sets the starting position for the highlighted portion of the field's display string to 2:

```
  Field1.SelectionStart := 2;
```

See also: ClearSelection, SelectAll, SelectedText, SelectionLength, SetSelection

SetInitialValue method

```
procedure SetInitialValue;
```

Initializes the field contents.

The initial value for string fields is an empty string. For numeric entry fields, the initial value is 0 or the lowest range value if the low range value is anything other than the default low range value.

For example, if you call SetInitialValue for an entry field editing a real data type with RangeLo set to -10, the field contents are set to -10.

See also: DataType, RangeLo

SetRangeHi method

```
procedure SetRangeHi(const Value : OvcData.TRangeType);
```

Sets the upper limit for the field.

The following example sets the upper limit for Field1 to 200. If a value larger than 200 is entered, a range exception is raised:

```
var
  R : TRangeType;
...
  R.rtWord := 200;
  {Field1 is a "Word" field}
  Field1.SetRangeHi(R);
```

See also: RangeHi, RangeLo, SetRangeLo

SetRangeLo method

```
procedure SetRangeLo(const Value : OvcData.TRangeType);
```

Sets the lower limit for the field.

The following example sets the lower limit for Field1 to 50. If a value smaller than 50 is entered, a range exception is raised:

```
var
  R : TRangeType;
...
  R.rtWord := 50;
  {Field1 is a "Word" field}
  Field1.SetRangeLo(R);
```

See also: RangeHi, RangeLo, SetRangeHi

SetSelection method

```
procedure SetSelection(Start, Stop : Word);
```

↳ Changes the selected portion of a field's display string.

SetSelection highlights the range starting at the character position specified by Start and ending with the character position specified by Stop (inclusive). The valid range for Start and Stop is from 0 to the length of the field's display string. If the starting position is greater than or equal to the ending position, no text is highlighted.

The following example sets the starting position for selection at the second character position and the ending position at the fifth character position:

```
  Field1.SetSelection(2, 5);
```

See also: ClearSelection, SelectAll, SelectedText, SelectionLength, SelectionStart

SetValue method

```
procedure SetValue(const Data);
```

↳ Assigns a value to a field directly from a variable of the appropriate data type.

Data is an untyped parameter that must have the same data type as the entry field. The value passed in Data is assumed to be valid.

Due to the way the VCL deals with untyped parameters, you should not pass a function as the parameter. The compiler will pass the address of the function and not the address of the result. Instead, assign the result of the function to a variable, then pass the variable to SetValue.

For example, assume the field is a date field and you want to set it to the current date. The following would produce unpredictable results:

```
  SetValue(CurrentDate);
```

Use the following example instead:

```
  var
    ADate : TStDate;
  ...
  ADate := CurrentDate;
  SetValue(ADate);
```

The following example assumes that Field1 is a simple real field:

```
var
  R : Real;
  I : Integer;
begin
...
  R := 33;
  {set the value of Field1}
  Field1.SetValue(R);
...
  {*** the following is incorrect
       and will cause unpredictable results ***}
  Field1.SetValue(I);
  {*** I is not the same data type as the field ***}
```

See also: TOvcNumericField.DataType, TOvcPictureField.DataType,
TOvcSimpleField.DataType, GetValue

TextMargin property

```
property TextMargin : Integer
```

Default: 2

↳ Determines the field's display indent.

The TextMargin property controls the left indent in pixels for the simple and picture entry fields and array editors. For the numeric entry field and array editor, this property is the right margin since the painting is done from right to left. The minimum value for TextMargin is 2.

Uninitialized property

```
property Uninitialized : Boolean
```

↳ Determines if the field contents are displayed.

If Uninitialized is True, the field is blanked out completely except when it has the focus. The state of this property is set to False when the field is modified.

UserData — run-time property

```
property UserData : TOvcUserData
```

Default: OvcUserData

Determines which TOvcUserData object the field uses for user-defined mask and substitution characters.

See “User-defined mask characters” on page 322 and “Substitution Characters” on page 323for additional information.

ValidateContents — method

```
function ValidateContents(ReportError: Boolean) : Word;
```

Validates the field contents.

If a method is assigned to the OnUserValidation event, that method is automatically called when the focus leaves the field. ValidateContents is provided in case you need to validate the field contents manually. If the field contents are invalid, ValidateContents returns the error code. ReportError determines if error reporting is performed. ValidateContents does not force the focus to the field if it detects an error.

See also: IsValid, OnUserValidation, ValidateSelf

ValidateSelf — method

```
function ValidateSelf : Boolean;
```

Validates the field contents while the field still has the focus.

If a method is assigned to the OnUserValidation event, that method is automatically called when the focus leaves the field. ValidateSelf is provided in case you need to validate the field contents manually before the focus leaves the field. If the field contents are invalid, ValidateSelf returns False.

See also: IsValid, OnUserValidation, ValidateContents

ZeroDisplay property

```
property ZeroDisplay : TZeroDisplay
```

```
TZeroDisplay = (zdShow, zdHide, zdHideUntilModified);
```

Default: zdShow

↳ Determines whether values are displayed when the field does not have the focus.

ZeroDisplay is used only when the entry field contains a numeric value. If ZeroDisplay equals zdHide, an empty display string is displayed when the field value is equal to the value assigned to ZeroDisplayValue. If ZeroDisplay equals zdHideUntilModified, the behavior is the same as zdHide, except that the field is displayed normally after it has been edited. If ZeroDisplay equals zdShow (the default), the field is always displayed.

Regardless of the setting of this property, the field value is always displayed when the field has the focus.

See also: ZeroDisplayValue

ZeroDisplayValue property

```
property ZeroDisplayValue : Double
```

Default: 0

↳ The value that determines if the field is displayed as blank.

ZeroDisplayValue is used only when the entry field contains a numeric value. It is used by the ZeroDisplay property to determine if the entry field value should be displayed when it does not have the focus.

See also: ZeroDisplay

TOvcCustomSimpleField Class

The TOvcCustomSimpleField class is the immediate ancestor of the TOvcSimpleField component. It implements all of the methods and properties used by the TOvcSimpleField component and is identical to the TOvcSimpleField component except that no properties are published.

TOvcCustomSimpleField is provided to facilitate creation of descendent simple field components. For property and method descriptions, see “TOvcSimpleField Component” on page 398.

Hierarchy

TCustomControl (VCL)

  TOvcCustomControl (OvcBase) 16

    TOvcCustomControlEx (OvcBase) 19

      TOvcBaseEntryField (OvcEF) 363

        TOvcCustomSimpleField (OvcSF)

TOvcSimpleField Component

The Orpheus simple entry field component is similar to the Windows standard edit control and the standard VCL TEdit component. The default editing and caret movement commands are basically the same. The differences are the additional features that the Orpheus simple entry field provides. It supports input validation on both a character-by-character and field- by-field basis. Additionally, the simple entry field has the unique ability to respond to alternate keyboard commands, such as WordStar-style commands or even user-defined commands. See “TOvcCommandProcessor Class” on page 50 for information on how to create and install user-defined commands and command tables.

The simple entry field is limited to a single line and a single character picture mask. The mask character indicates the set of characters that are allowed. For example, a simple Word field might have a mask character of ‘9’, which allows digits and blanks only, and a maximum length of 5. A simple string field might have a mask character of ‘X’, which allows any character.

Example

This example shows how you can use the TOvcSimpleField to allow entry and validation of decimal and whole number values.

Create a new project, add components, and set the property values as indicated in Table 11.2.

Table 11.2: *TOvcSimpleField property values*

Component	Property	Value
OvcSimpleField	DataType	sftReal
	DecimalPlaces	2
	RangeHi	200
	RangeLo	-200
OvcSimpleField	DataType	sftByte

When you add the first TOvcSimpleField to the form, a default TOvcController non-visual component is automatically added (if there isn’t already one on the form) and the field’s Controller property is changed to reflect the name of the TOvcController.

Run the project and experiment with the generated program. Try entering both valid and invalid values into either of the fields. When a field detects an error, it forces the focus to return to the field so the error can be corrected.

Hierarchy

TCustomControl (VCL)

- ❶ TOvcCustomControl (OvcBase). 16
  - ❷ TOvcCustomControlEx (OvcBase) 19
    - ❸ TOvcBaseEntryField (OvcEF) 363
      - TOvcCustomSimplefield (OvcSF). 397
        - TOvcSimpleField (OvcSF)

Properties

❶ About
❸ AsBoolean
❸ AsCents
❸ AsDateTime
❸ AsExtended
❸ AsFloat
❸ AsInteger
❸ AsOvcDate
❸ AsOvcTime
❸ AsStDate
❸ AsStTime
❶ AttachedLabel
❸ AutoSize
❸ BorderStyle
❸ CaretIns
❸ CaretOvr
❸ ControlCharColor
❷ Controller
❸ CurrentPos
❸ DataSize
DataType
❸ DecimalPlaces
❸ DisplayString
❸ EditString
❸ EFColors
❸ Epoch
❸ EverModified
❸ Font
❸ IntlSupport
❶ LabelInfo
❸ LastError
❸ MaxLength
❸ Modified
❸ Options
❸ PadChar
❸ PasswordChar
PictureMask
❸ RangeHi
❸ RangeLo
❸ SelectedText
❸ SelectionLength
❸ SelectionStart
❸ TextMargin
❸ Uninitialized
❸ UserData
❸ ZeroDisplay
❸ ZeroDisplayValue

Methods

- ❸ ClearContents
- ❸ ClearSelection
- ❸ CopyToClipboard
- ❸ CutToClipboard
- ❸ DecreaseValue
- ❸ Deselect
- ❸ FieldIsEmpty
- ❸ GetStrippedEditString
- ❸ GetValue
- ❸ IncreaseValue
- ❸ IsValid
- ❸ MergeWithPicture
- ❸ MoveCaret
- ❸ MoveCaretToEnd
- ❸ MoveCaretToStart
- ❸ PasteFromClipboard
- ❸ ProcessCommand
- ❸ ResetCaret
- ❸ Restore
- ❸ SelectAll
- ❸ SetInitialValue
- ❸ SetRangeHi
- ❸ SetRangeLo
- ❸ SetSelection
- ❸ SetValue
- ❸ ValidateContents
- ❸ ValidateSelf

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❸ OnChange
- ❸ OnGetEpoch
- ❸ OnError
- ❶ OnMouseWheel
- ❸ OnUserCommand
- ❸ OnUserValidation

Reference Section

DataType property

```
property DataType : TSimpleDataType
```

Default: sftString

Determines what type of information the simple entry field can edit.

The possible values and their corresponding data types are:

	Delphi	**C++Builder**
sftString	string	AnsiString
sftChar	AnsiChar	unsigned char
sftBoolean	Boolean	bool
sftYesNo	Boolean	bool
sftLongInt	LongInt	long or int
sftWord	Word	Word (unsigned short int)
sftInteger	SmallInt	short int
sftByte	Byte	Byte (unsigned char)
sftShortInt	ShortInt	ShortInt (signed char)
sftReal	Real	double
sftExtended	Extended	Extended (long double)
sftDouble	Double	double
sftSingle	Single	Single (float)
sftComp	Comp	Comp (double)

Caution: When DataType is changed, the field's properties are reset to the defaults that apply to the new data type.

```
property PictureMask : AnsiChar
```

Default: pmAnyChar ('X')

Defines which characters the field accepts and the character format.

This property defines the input and display mask used for the entry field. The possible values are:

Value	**Description**
`pmAnyChar     = 'X';`	`{Allows any character}`
`pmForceUpper  = '!';`	`{Allows any character, forces upper case}`
`pmForceLower  = 'L';`	`{Allows any character, forces lower case}`
`pmForceMixed  = 'x';`	`{Allows any character, forces mixed case}`
`pmAlpha       = 'a';`	`{Allows alphas only}`
`pmUpperAlpha  = 'A';`	`{Allows alphas only, forces upper case}`
`pmLowerAlpha  = 'l';`	`{Allows alphas only, forces lower case}`
`pmPositive    = '9';`	`{Allows numbers and spaces only}`
`pmWhole       = 'i';`	`{Allows numbers, spaces, minus}`
`pmDecimal     = '#';`	`{Allows numbers, spaces, minus, period}`
`pmScientific  = 'E';`	`{Allows numbers, spaces, minus, period, 'e'}`
`pmHexadecimal = 'K';`	`{Allows 0-9 and A-F, forces upper case}`
`pmBinary      = 'b';`	`{Allows 0-1, space}`
`pmOctal       = 'O';`	`{Allows 0-7, space}`
`pmTrueFalse   = 'B';`	`{Allows T, t, F, f}`
`pmYesNo       = 'Y';`	`{Allows Y, y, N, n}`

The value assigned to PictureMask is tested only against the list shown above. It is still possible to assign a character that is inappropriate because of the field's data type. For example, if the field has a data type of sftReal, setting the PictureMask property to 'A' is allowed but is invalid because it allows only alphabetic characters.

You can directly enter a mask character in the PictureMask edit field in the Object Inspector or you can invoke the picture mask property editor to view and select from a list of the picture mask characters. Double-click on the PictureMask property value or select the dialog button to display the "Simple Mask" property editor.

See "Picture Masks" on page 316 for more information about picture masks and their usage.

TOvcDbSimpleField Component

TOvcDbSimpleField is a direct descendant of TOvcCustomSimpleField and inherits all of its properties and methods. These properties and methods are documented in "TOvcSimpleField Component" on page 398.

The primary differences in the behavior of the TOvcDbSimpleField and TOvcSimpleField are that TOvcDbSimpleField is capable of connecting to a data source and editing the contents of several types of database fields. It can edit fields of the following types:

```
ftString
ftSmallInt
ftInteger
ftWord
ftBoolean
ftFloat
```

TOvcDbSimpleField provides four additional properties. The DataField, DataSource, Field, and FieldType properties are exactly the same as the like-named properties in VCL's standard data-aware components. The FieldType property is set automatically when the TOvcDbSimpleField is connected to an active data source and assigned a field name.

To use a TOvcDbSimpleField component, just assign a TDataSource to the DataSource property and assign an appropriate field name to the DataField property. The entry field automatically gets the database value and updates the data source with the changed data when the field loses the focus.

Caution: Do not attempt to change the value of a data-aware entry field programmatically using methods or properties other than those provided by the Field property. The results can be unpredictable. For example, you should not use the following:

```
MyEntryField.AsString := 'Some text';
```

The display is changed, but only until the field next receives the focus or some other change in the data source causes the data-aware entry field to refresh itself. Instead, you should use:

```
MyEntryField.Field.AsString := 'Some text';
```

This changes the underlying database field value. The data-aware entry field detects this change and updates its display.

Hierarchy

TCustomControl (VCL)

❶ sTOvcCustomControl (OvcBase) 16

❷ TOvcCustomControlEx (OvcBase) 19

❸ TOvcBaseEntryfield (OvcEF) 363

TOvcCustomSimpleField (OvcSF) 397

TOvcDbSimpleField (OvcDbSF)

Properties

❶ About
❸ AsBoolean
❸ AsCents
❸ AsDateTime
❸ AsExtended
❸ AsFloat
❸ AsInteger
❸ AsOvcDate
❸ AsOvcTime
❸ AsStDate
❸ AsStTime
❶ AttachedLabel
❸ AutoSize
❸ BorderStyle
❸ CaretIns
❸ CaretOvr
❸ ControlCharColor
❷ Controller
❸ CurrentPos
DataField
DataSize
DataSource
❸ DecimalPlaces
❸ DisplayString
❸ EditString
❸ EFColors
❸ Epoch
❸ EverModified
Field
FieldType
❸ Font
❸ IntlSupport
❶ LabelInfo
❸ LastError
❸ MaxLength
❸ Modified
❸ Options
❸ PadChar
❸ PasswordChar
PictureMask
❸ RangeHi
❸ RangeLo
❸ SelectedText
❸ SelectionLength
❸ SelectionStart
❸ TextMargin
❸ Uninitialized
❸ UserData
UseTFieldMask
ZeroAsNull
❸ ZeroDisplay
❸ ZeroDisplayValue

Methods

- ❸ ClearContents
- ❸ ClearSelection
- ❸ CopyToClipboard
- ❸ CutToClipboard
- ❸ DecreaseValue
- ❸ Deselect
- ❸ FieldIsEmpty
- ❸ GetStrippedEditString
- ❸ GetValue
- ❸ IncreaseValue
- ❸ IsValid
- ❸ MergeWithPicture
- ❸ MoveCaret
- ❸ MoveCaretToEnd
- ❸ MoveCaretToStart
- ❸ PasteFromClipboard
- ❸ ProcessCommand
- ❸ ResetCaret
- ❸ Restore
- ❸ SelectAll
- ❸ SetInitialValue
- ❸ SetRangeHi
- ❸ SetRangeLo
- ❸ SetSelection
- ❸ SetValue
- ❸ ValidateContents
- ❸ ValidateSelf

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❸ OnChange
- ❸ OnGetEpoch
- ❸ OnError
- ❶ OnMouseWheel
- ❸ OnUserCommand
- ❸ OnUserValidation

Reference Section

DataField — property

```
property DataField : string
```

Identifies the field (in the data source component) from which the entry field component displays data.

DataSource — property

```
property DataSource : TDataSource
```

Specifies the data source component where the entry field obtains the data to display.

Field — run-time, read-only property

```
property Field: TField
```

Returns the TField object to which the entry field component is linked.

Use the Field object to change the value of the data in the field programmatically.

FieldType — property

```
property FieldType : TFieldType
```

```
TFieldType = (
  ftUnknown, ftString, ftSmallint, ftInteger, ftWord, ftBoolean,
  ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime, ftBytes,
  ftVarBytes, ftBlob, ftMemo,ftGraphic);
```

Default: ftUnknown

Determines the type of the database field.

When a data-aware entry field is attached to an active data source and the DataField property is set to the name of the field, this property is automatically set to match the data type of the database field. If it is not desirable to activate the data source, this field must be set to the correct field type so that the Orpheus entry field can configure its internal data type and initialize the picture mask.

The data-aware entry fields do not support all of the database field types shown in the definition of TFieldType (see the online help for the VCL DB unit). If you select one that is not supported, an error message is displayed.

See also: DataField, DataSource

UseTFieldMask property

```
property UseTFieldMask : Boolean
```

Default: False

↳ Determines which mask is used to draw the field.

If UseTFieldMask is False (the default), the entry field uses the mask defined by the PictureMask property to format and display its value. If UseTFieldMask is True, the entry field uses the mask provided by the underlying TField object to format and display its value. When the field has the focus, the PictureMask value is always used to format the display, regardless of the value of UseTFieldMask.

ZeroAsNull property

```
property ZeroAsNull : Boolean
```

Default: False

↳ Determines if zero values are stored as null.

If ZeroAsNull is set to True, fields editing a numeric data type that contain zero are stored as null database values. If False, the numeric zero is stored in the attached database field.

Setting this property to True is not necessary if you just want to initialize the database field to null. You can call the Field.Clear method to do this on an as-needed basis.

See also: Field

TOvcBasePictureField Class

The TOvcBasePictureField class is a direct descendant of the TOvcBaseEntryField class and provides the foundation for both the picture and numeric entry field components.

TOvcBasePictureField implements string picture masks and supporting methods, adds additional validation logic, and deals with the issues involved with caret positioning in a field that contains literal characters. No new methods or properties are documented in this class.

Hierarchy

TCustomControl (VCL)

    TOvcCustomControl (OvcBase) 16

        TOvcCustomControlEx (OvcBase) 19

            TOvcBaseEntryField (OvcEF) 363

                TOvcBasePictureField (OvcPB)

TOvcCustomPictureField Class

The TOvcCustomPictureField class is the immediate ancestor of the TOvcPictureField component. It implements all of the methods and properties used by the TOvcPictureField component and is identical to the TOvcPictureField component except that no properties are published.

TOvcCustomPictureField is provided to facilitate creation of descendent picture field components. For property and method descriptions, see "TOvcPictureField Component" on page 411.

Hierarchy

TCustomControl (VCL)

  TOvcCustomControl (OvcBase) 16

    TOvcCustomControlEx (OvcBase) 19

      TOvcBaseEntryField (OvcEF) 363

        TOvcBasePictureField (OvcPB). 409

          TOvcCustomPictureField (OvcPF)

TOvcPictureField Component

Picture entry fields differ from simple entry fields in that picture entry fields allow different mask characters for each character position in the field. For example, a social security number field could have a picture mask of ‘999-99-9999’. The hyphens are treated as literal characters, and cannot be changed during data entry. In the positions that contain ‘9’, only digits or blanks can be entered. Picture entry fields are always padded to their maximum length with blanks.

Although the picture entry field differs from the simple entry field in the degree to which input can be validated on a character-by-character basis, in essence they both provide the same facilities for validation. They use the same process for validating data on a field-by-field basis.

Picture entry fields allow the same data types supported by simple entry fields, plus support for the date and time data types.

Example

This example shows how you can use TOvcPictureFields to allow entry and validation of dates and times.

Create a new project, add components, and set the property values as indicated in Table 11.3.

Component	**Property**	**Value**
`OvcPictureField`	`DataType`	`pftDate`
`OvcPictureField`	`DataType`	`pftTime`
	`PictureMask`	`Hh:mm`

When you add the first TOvcPictureField to the form, a default TOvcController non-visual component is automatically added (if there isn’t already one on the form) and the field’s Controller property is changed to reflect the name of the TOvcController.

Default picture masks are based on the current international settings in the WIN.INI file. For applications used in several countries, set the PictureMask property at run time using one of the InternationalXxx methods in “TOvcIntlSup Class” on page 344.

Run the project and experiment with the generated program. Try entering both valid and invalid values into either of the fields. When a field detects an error, it forces the focus to return to the field so the error can be corrected.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase). 16

❷ TOvcCustomControlEx (OvcBase) 19

❸ TOvcBaseEntryField (OvcEF) 363

TOvcBasePictureField (OvcPB). 409

TOvcCustomPictureField (OvcPF) . . . .

TOvcPictureField (OvcPF)

Properties

❶	About	❸	ControlCharColor	❸	Modified
❸	AsBoolean	❷	Controller	❸	Options
❸	AsCents	❸	CurrentPos	❸	PadChar
❸	AsDateTime		DataType	❸	PasswordChar
❸	AsExtended	❸	DecimalPlaces		PictureMask
❸	AsFloat	❸	DisplayString	❸	RangeHi
❸	AsInteger	❸	EditString	❸	RangeLo
❸	AsOvcDate	❸	EFColors	❸	SelectedText
❸	AsOvcTime	❸	Epoch	❸	SelectionLength
❸	AsStDate	❸	EverModified	❸	SelectionStart
❸	AsStTime	❸	Font	❸	TextMargin
❶	AttachedLabel		InitDateTime	❸	Uninitialized
❸	AutoSize	❸	IntlSupport	❸	UserData
❸	BorderStyle	❶	LabelInfo	❸	ZeroDisplay
❸	CaretIns	❸	LastError	❸	ZeroDisplayValue
❸	CaretOvr	❸	MaxLength		

Methods

❸ ClearContents	❸ IncreaseValue	❸ Restore
❸ ClearSelection	❸ IsValid	❸ SelectAll
❸ CopyToClipboard	❸ MergeWithPicture	❸ SetInitialValue
❸ CutToClipboard	❸ MoveCaret	❸ SetRangeHi
❸ DecreaseValue	❸ MoveCaretToEnd	❸ SetRangeLo
❸ Deselect	❸ MoveCaretToStart	❸ SetSelection
❸ FieldIsEmpty	❸ PasteFromClipboard	❸ SetValue
❸ GetStrippedEditString	❸ ProcessCommand	❸ ValidateContents
❸ GetValue	❸ ResetCaret	❸ ValidateSelf

Events

❶ AfterEnter	❸ OnGetEpoch	❸ OnUserCommand
❶ AfterExit	❸ OnError	❸ OnUserValidation
❸ OnChange	❶ OnMouseWheel	

Reference Section

DataType property

```
property DataType : TPictureDataType
```

Default: pftString

Determines what type of information the picture entry field can edit.

The possible values and their corresponding data types are:

	Delphi	**C++Builder**
pftString	string	AnsiString
pftChar	AnsiChar	unsigned char
pftBoolean	Boolean	bool
pftYesNo	Boolean	bool
pftLongInt	LongInt	long or int
pftWord	Word	Word (unsigned short int)
pftInteger	SmallInt	short int
pftByte	Byte	Byte (unsigned char)
pftShortInt	ShortInt	ShortInt (signed char)
pftReal	Real	double
pftExtended	Extended	Extended (long double)
pftDouble	Double	double
pftSingle	Single	Single (float)
pftComp	Comp	Comp (double)
pftDate	TStDate	TStDate
pftTime	TStTime	TStDate

Caution: When DataType is changed, the field’s properties are reset to the defaults that apply to the new data type.

InitDateTime property

```
property InitDateTime : Boolean
```

Default: False

Determines if a date or time picture entry field is initialized to the current date or time.

If InitDateTime is set to True, picture entry fields that have a DataType of pftDate or pftTime are automatically initialized to the current system date or time when the component's window handle is created.

PictureMask property

```
property PictureMask : string
```

Default: "XXXXXXXXXXXXXXX"

Defines which characters the field accepts and the character format.

This property defines the input and display mask used for the picture entry field. The possible values are:

Value	**Description**
`pmAnyChar     = 'X';`	`{allows any character}`
`pmForceUpper  = '!';`	`{allows any character, forces upper case}`
`pmForceLower  = 'L';`	`{allows any character, forces lower case}`
`pmForceMixed  = 'x';`	`{allows any character, forces mixed case}`
`pmAlpha       = 'a';`	`{allows alphas only}`
`pmUpperAlpha  = 'A'`	`{allows alphas only, forces upper case}`
`pmLowerAlpha  = 'l';`	`{allows alphas only, forces lower case}`
`pmPositive    = '9';`	`{allows numbers and spaces only}`
`pmWhole       = 'i';`	`{allows numbers, spaces, minus}`
`pmDecimal     = '#';`	`{allows numbers, spaces, minus, period}`
`pmScientific  = 'E';`	`{allows numbers, spaces, minus, period, 'e'}`
`pmHexadecimal = 'K';`	`{allows 0-9 and A-F, forces upper case}`
`pmBinary      = 'b';`	`{allows 0-1, space}`
`pmOctal       = 'O';`	`{allows 0-7, space}`
`pmTrueFalse   = 'B';`	`{allows T, t, F, f}`
`pmYesNo       = 'Y';`	`{allows Y, y, N, n}`

The value assigned to PictureMask is not tested for validity. It is possible to assign a mask that is invalid. You can directly enter a mask string in the PictureMask edit field in the Object Inspector or you can invoke the picture mask property editor to view and select from several example picture masks. Double-click the PictureMask property value in the Object Inspector or select the dialog button to display the “Picture Mask” property editor.

See “Picture Masks” on page 316 for more information about picture masks and their usage.

TOvcDbPictureField Component

TOvcDbPictureField is a direct descendant of TOvcCustomPictureField and inherits all of its properties and methods. These properties and methods are documented in “TOvcPictureField Component” on page 411.

The primary differences in the behavior of the TOvcDbPictureField and TOvcPictureField are that TOvcDbPictureField is capable of connecting to a data source and editing the contents of several types of database fields. It can edit fields of the following types:

```
ftString
ftSmallInt
ftInteger
ftWord
ftBoolean
ftFloat
ftCurrency
ftDate
ftTime
ftDateTime
```

TOvcDbPictureField provides five additional properties. The DataField, DataSource, DateOrTime, Field, FieldType properties are exactly the same as the like-named properties in VCL’s standard data-aware components. A FieldType property is set automatically when the TOvcDbPictureField is connected to an active data source and assigned a field name.

To use a TOvcDbPictureField component, just assign a TDataSource to the DataSource property and assign an appropriate field name to the DataField property. The entry field automatically gets the database value and updates the data source with the changed data when the field loses the focus.

Caution: Do not attempt to change the value of a data-aware entry field programmatically using methods or properties other than those provided by the Field property. The results can be unpredictable. For example, you should not use the following:

```
MyEntryField.AsString := 'Some text';
```

The display is changed, but only until the field next receives the focus or some other change in the data source causes the data-aware entry field to refresh itself. Instead, you should use:

```
MyEntryField.Field.AsString := 'Some text';
```

This changes the underlying database field value. The data-aware entry field detects this change and updates its display.

Hierarchy

TCustomControl (VCL)

- ❶ TOvcCustomControl (OvcBase) 16
  - ❷ TOvcCustomControlEx (OvcBase) 19
    - ❸ TOvcBaseEntryField (OvcEF) 363
      - TOvcBasePictureField (OvcPB) 409
        - TOvcCustomPictureField (OvcPF) 410
          - TOvcDbPictureField (OvcDbPF)

Properties

❶ About
❸ AsBoolean
❸ AsCents
❸ AsDateTime
❸ AsExtended
❸ AsFloat
❸ AsInteger
❸ AsOvcDate
❸ AsOvcTime
❸ AsStDate
❸ AsStTime
❶ AttachedLabel
❸ AutoSize
❸ BorderStyle
❸ CaretIns
❸ CaretOvr
❸ ControlCharColor
❷ Controller
❸ CurrentPos
DataField
❸ DataSize
DataSource
DateOrTime
❸ DecimalPlaces
❸ DisplayString
❸ EditString
❸ EFColors
❸ Epoch
❸ EverModified
Field
FieldType
❸ Font
❸ IntlSupport
❶ LabelInfo
❸ LastError
❸ MaxLength
❸ Modified
❸ Options
❸ PadChar
❸ PasswordChar
PictureMask
❸ RangeHi
❸ RangeLo
❸ SelectedText
❸ SelectionLength
❸ SelectionStart
❸ TextMargin
❸ Uninitialized
❸ UserData
UseTFieldMask
ZeroAsNull
❸ ZeroDisplay
❸ ZeroDisplayValue

Methods

- ❸ ClearContents
- ❸ ClearSelection
- ❸ CopyToClipboard
- ❸ CutToClipboard
- ❸ DecreaseValue
- ❸ Deselect
- ❸ FieldIsEmpty
- ❸ GetStrippedEditString
- ❸ GetValue
- ❸ IncreaseValue
- ❸ IsValid
- ❸ MergeWithPicture
- ❸ MoveCaret
- ❸ MoveCaretToEnd
- ❸ MoveCaretToStart
- ❸ PasteFromClipboard
- ❸ ProcessCommand
- ❸ ResetCaret
- ❸ Restore
- ❸ SelectAll
- ❸ SetInitialValue
- ❸ SetRangeHi
- ❸ SetRangeLo
- ❸ SetSelection
- ❸ SetValue
- ❸ ValidateContents
- ❸ ValidateSelf

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❸ OnChange
- ❸ OnGetEpoch
- ❸ OnError
- ❶ OnMouseWheel
- ❸ OnUserCommand
- ❸ OnUserValidation

Reference Section

DataField property

```
property DataField : string
```

Identifies the field (in the data source component) from which the entry field component displays data.

DataSource property

```
property DataSource : TDataSource
```

Specifies the data source component where the entry field obtains the data to display.

DateOrTime property

```
property DateOrTime : TDateOrTime
```

```
TDateOrTime = (
  ftUseDate, ftUseTime, ftUseBothEditDate, ftUseBothEditTime);
```

Default: ftUseDate

Determines whether ftDateTime database fields are edited as dates or times.

Orpheus does not provide an entry field capable of editing both dates and times. This property is used when the data-aware field is connected to an ftDateTime field to determine which of the two values to edit.

The behavior of the field for various values of DateOrTime is as follows:

Value	**Result**
ftUseDate	The entry field displays and edits the date portion of the ftDateTime value.
ftUseTime	The entry field displays and edits the time portion of the ftDateTime value.
ftUseBothEditDate	The entry field displays the date and time of the ftDateTime value when the field is unfocused, and allows you to edit the date portion when the field is focused.
ftUseBothEditTime	The entry field displays the date and time of the ftDateTime value when the field is unfocused, and allows you to edit the time portion when the field is focused.

In all cases, the portion of the ftDateTime field that is not edited is preserved.

Field — run-time, read-only property

```
property Field: TField
```

Returns the TField object to which the entry field component is linked.

Use the Field object when you want to change the value of the data in the field programmatically.

FieldType — property

```
property FieldType : TFieldType

TFieldType = (
  ftUnknown, ftString, ftSmallint, ftInteger, ftWord, ftBoolean,
  ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime, ftBytes,
  ftVarBytes, ftBlob, ftMemo, ftGraphic);
```

Default: ftUnknown

Determines the type of the database field.

When a data-aware entry field is attached to an active data source and the DataField property is set to the name of the field, this property is automatically set to match the data type of the database field. If it is not desirable to activate the data source, this field must be set to the correct field type so that the Orpheus entry field can configure its internal data type and initialize the picture mask.

The data-aware entry fields do not support all of the possible database field types shown in the definition of TFieldType (see the online help for the VCL DB unit). If you select one that is not supported, an error message is displayed. For example, if the ftGraphic field type is selected, an error is displayed because none of the data-aware entry fields support that field type.

See also: DataField, DataSource.

UseTFieldMask property

```
property UseTFieldMask : Boolean
```

Default: False

↳ Determines which mask is used to draw the field.

If UseTFieldMask is False (the default), the entry field uses the mask defined by the PictureMask property to format and display its value. If UseTFieldMask is True, the entry field uses the mask provided by the underlying TField object to format and display its value. When the field has the focus, the PictureMask value is always used to format the display, regardless of the value of UseTFieldMask.

ZeroAsNull property

```
property ZeroAsNull : Boolean
```

Default: False

↳ Determines if zero values are stored as null.

11

If ZeroAsNull is set to True, fields editing a numeric data type that contain zero are stored as null database values. If False, the numeric zero is stored in the attached database field.

Setting this property to True is not necessary if you just want to initialize the database field to null. You can call the Field.Clear method to do this on an as-needed basis.

See also: Field

TOvcCustomNumericField Class

The TOvcCustomNumericField class is the immediate ancestor of the TOvcNumericField component. It implements all of the methods and properties used by the TOvcNumericField component and is identical to the TOvcNumericField component except that no properties are published.

TOvcCustomNumericField is provided to facilitate creation of custom numeric field components. For property and method descriptions, see "TOvcNumericField Component" on page 424.

Hierarchy

TCustomControl (VCL)

  TOvcCustomControl (OvcBase) 16

    TOvcCustomControlEx (OvcBase) 19

      TOvcBaseEntryField (OvcEF) 363

        TOvcBasePictureField (OvcPB). 409

          TOvcCustomerNumericField (OvcNF)

TOvcNumericField Component

Numeric fields are similar to picture fields, but they use a calculator-style editor that enters data into the field from right-to-left. For example, in a numeric entry field with a mask of "$###,###.##", assume that the field is initialized to 0, and you enter "1234.5." Here is what you see as each character is entered:

```
initially               '      $0.00'
enter '1'               '         $1'
enter '2'               '        $12'
enter '3'               '       $123'
enter '4'               '     $1,234'
enter '.'               '    $1,234.'
enter '5'               '   $1,234.5'
leave field             '  $1,234.50'
```

As characters are entered, they are inserted at the end of the picture mask (the caret is displayed at the end of the picture mask and it cannot be moved). You can enter more or fewer digits to the right of the decimal point than the mask allows, but not to the left of the decimal point. When the caret is moved out of the field, it is automatically redisplayed in the format specified by the picture mask. When appropriate, the entered value is rounded to conform to the picture mask's requirements. For example, if you enter "1234.567", the field displays " $1,234.57" when the caret leaves the field.

Example

11

This example shows how you can use TOvcNumericFields to allow entry and validation of numeric values that use calculator-style input.

Create a new project, add components, and set the property values as indicated in Table 11.3.

Table 11.3:

Component	Property	Value
OvcNumericField	DataType	nftDouble
OvcNumericField	DataType	nftShortInt

When you add the first TOvcNumericField to the form, a default TOvcController non-visual component is automatically added (if there isn't already one on the form) and the field's Controller property is changed to reflect the name of the TOvcController.

Run the project and experiment with the generated program. Try entering both valid and invalid values into either of the fields. When a field detects an error, it forces the focus to return to the field so the error can be corrected.

Hierarchy

TCustomControl (VCL)

  ❶ TOvcCustomControl (OvcBase). 16

    ❷ TOvcCustomControlEx (OvcBase) 19

      ❸ TOvcBaseEntryField (OvcEF) 363

        TOvcPictureBase (OvcPB)

          TOvcCustomNumericField (OvcNF)

            TOvcNumericField (OvcNF)

Properties

❶ About
❸ AsBoolean
❸ AsCents
❸ AsDateTime
❸ AsExtended
❸ AsFloat
❸ AsInteger
❸ AsOvcDate
❸ AsOvcTime
❸ AsStDate
❸ AsStTime
❶ AttachedLabel
❸ AutoSize
❸ BorderStyle
❸ CaretIns
❸ CaretOvr
❸ ControlCharColor
❷ Controller
❸ CurrentPos
❸ DataSize
DataType
❸ DecimalPlaces
❸ DisplayString
❸ EditString
❸ EFColors
❸ Epoch
❸ EverModified
❸ Font
❸ IntlSupport
❶ LabelInfo
❸ LastError
❸ MaxLength
❸ Modified
❸ Options
❸ PadChar
❸ PasswordChar
PictureMask
❸ RangeHi
❸ RangeLo
❸ SelectedText
❸ SelectionLength
❸ SelectionStart
❸ TextMargin
❸ Uninitialized
❸ UserData
❸ ZeroDisplay
❸ ZeroDisplayValue

Methods

- ❸ ClearContents
- ❸ ClearSelection
- ❸ CopyToClipboard
- ❸ CutToClipboard
- ❸ DecreaseValue
- ❸ Deselect
- ❸ FieldIsEmpty
- ❸ GetStrippedEditString
- ❸ GetValue
- ❸ IncreaseValue
- ❸ IsValid
- ❸ MergeWithPicture
- ❸ MoveCaret
- ❸ MoveCaretToEnd
- ❸ MoveCaretToStart
- ❸ PasteFromClipboard
- ❸ ProcessCommand
- ❸ ResetCaret
- ❸ Restore
- ❸ SelectAll
- ❸ SetInitialValue
- ❸ SetRangeHi
- ❸ SetRangeLo
- ❸ SetSelection
- ❸ SetValue
- ❸ ValidateContents
- ❸ ValidateSelf

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❸ OnChange
- ❸ OnGetEpoch
- ❸ OnError
- ❶ OnMouseWheel
- ❸ OnUserCommand
- ❸ OnUserValidation

DataType property

```
property DataType : TNumericDataType
```

Default: nftLongInt

Determines which type of information the numeric entry field can edit.

The possible values and their corresponding data types are:

	Delphi	**C++Builder**
`nftLongInt`	`LongInt`	`Long or int`
`nftWord`	`Word`	`Word (unsigned short int)`
`nftInteger`	`SmallInt`	`Short int`
`nftByte`	`Byte`	`Byte (unsigned char)`
`nftShortInt`	`ShortInt`	`ShortInt (signed char)`
`nftReal`	`Real`	`Double`
`nftExtended`	`Extended`	`Extended (long double)`
`nftDouble`	`Double`	`Double`
`nftSingle`	`Single`	`Single (float)`
`nftComp`	`Comp`	`Comp (double)`

Caution: When DataType is changed, the field's properties are reset to the defaults that apply to the new data type.

```
property PictureMask : string
```

Default: “##########”

Determines which characters the field accepts and the character format.

This property defines the input and display mask used for the entry field. The possible values are:

Value	**Meaning**
`pmPositive    = ‘9’;`	`{allows numbers and spaces only}`
`pmWhole       = ‘i’;`	`{allows numbers, spaces, minus}`
`pmDecimal     = ‘#’;`	`{allows numbers, spaces, minus, period}`
`pmHexadecimal = ‘K’;`	`{allows 0-9 and A-F, forces upper case}`
`pmOctal       = ‘O’;`	`{allows 0-7, space}`
`pmBinary      = ‘b’;`	`{allows 0-1, space}`

The value assigned to PictureMask is not tested for validity. It is possible to assign a mask that is invalid.

You can directly enter a mask string in the PictureMask edit field in the Object Inspector or you can invoke the picture mask property editor to view and select from several example picture masks. Double-click the PictureMask property value in the Object Inspector or select the dialog button to display the “Numeric Mask” property editor.

See “Picture Masks” on page 316 for more information about picture masks and their usage.

TOvcDbNumericField Component

TOvcDbNumericField is a direct descendant of TOvcCustomNumericField and inherits all of its properties and methods. These properties and methods are documented in "TOvcNumericField Component" on page 424.

There are very few differences in the behavior of the TOvcDbNumericField and TOvcNumericField except that TOvcDbNumericField is capable of connecting to a data source and editing the contents of several types of database fields. It can edit fields of the following types:

```
ftCurrency
ftSmallInt
ftInteger
ftWord
ftFloat
```

TOvcDbNumericField provides four additional properties. The DataField, DataSource, Field, and FieldType properties are exactly the same as the like-named properties in VCL's standard data-aware components. The FieldType property is set automatically when the TOvcDbNumericField is connected to an active data source and assigned a field name.

To use a TOvcDbNumericField component, just assign a TDataSource to the DataSource property and assign an appropriate field name to the DataField property. The entry field automatically gets the database value and updates the data source with the changed data when the field loses the focus.

Caution: Do not attempt to change the value of a data-aware entry field programmatically using methods or properties other than those provided by the Field property. The results can be unpredictable. For example, you should not use the following:

```
MyEntryField.AsString := 'Some text';
```

The display is changed, but only until the field next receives the focus or some other change in the data source causes the data-aware entry field to refresh itself. Instead, you should use:

```
MyEntryField.Field.AsString := 'Some text';
```

This changes the underlying database field value. The data-aware entry field detects this change and updates its display.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase). 16

❷ TOvcCustomControlEx (OvcBase) 19

❸ TOvcBaseEntryField (OvcEF) 363

TOvcBasePictureField (OvcPB). 409

TOvcCustomNumericField (OvcNF) 423

TOvcDbNumericField (OvcDbNF)

Properties

❶	About	❷	Controller	❸	Modified
❸	AsBoolean	❸	CurrentPos	❸	Options
❸	AsCents		DataField	❸	PadChar
❸	AsDateTime		DataSource	❸	PasswordChar
❸	AsExtended	❸	DecimalPlaces	❸	RangeHi
❸	AsFloat	❸	DisplayString	❸	RangeLo
❸	AsInteger	❸	EditString	❸	SelectedText
❸	AsOvcDate	❸	EFColors	❸	SelectionLength
❸	AsOvcTime	❸	Epoch	❸	SelectionStart
❸	AsStDate	❸	EverModified	❸	TextMargin
❸	AsStTime		Field	❸	Uninitialized
❶	AttachedLabel		FieldType	❸	UserData
❸	AutoSize	❸	Font		UseTFieldMask
❸	BorderStyle	❸	IntlSupport		ZeroAsNull
❸	CaretIns	❶	LabelInfo	❸	ZeroDisplay
❸	CaretOvr	❸	LastError	❸	ZeroDisplayValue
❸	ControlCharColor	❸	MaxLength		

Methods

- ❸ ClearContents
- ❸ ClearSelection
- ❸ CopyToClipboard
- ❸ CutToClipboard
- ❸ DecreaseValue
- ❸ Deselect
- ❸ FieldIsEmpty
- ❸ GetStrippedEditString
- ❸ GetValue
- ❸ IncreaseValue
- ❸ IsValid
- ❸ MergeWithPicture
- ❸ MoveCaret
- ❸ MoveCaretToEnd
- ❸ MoveCaretToStart
- ❸ PasteFromClipboard
- ❸ ProcessCommand
- ❸ ResetCaret
- ❸ Restore
- ❸ SelectAll
- ❸ SetInitialValue
- ❸ SetRangeHi
- ❸ SetRangeLo
- ❸ SetSelection
- ❸ SetValue
- ❸ ValidateContents
- ❸ ValidateSelf

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❸ OnChange
- ❸ OnGetEpoch
- ❸ OnError
- ❶ OnMouseWheel
- ❸ OnUserCommand
- ❸ OnUserValidation

Reference Section

DataField property

```
property DataField : string
```

Identifies the field (in the data source component) from which the entry field component displays data.

DataSource property

```
property DataSource : TDataSource
```

Specifies the data source component where the entry field obtains the data to display.

Field run-time, read-only property

```
property Field: TField;
```

Returns the TField object to which the entry field component is linked.

Use the Field object to change the value of the data in the field programmatically.

FieldType property

```
property FieldType : TFieldType
```

```
TFieldType = (
  ftUnknown, ftString, ftSmallint, ftInteger, ftWord, ftBoolean,
  ftFloat, ftCurrency, ftBCD, ftDate, ftTime, ftDateTime, ftBytes,
  ftVarBytes, ftBlob, ftMemo, ftGraphic);
```

Default: ftUnknown

Determines the type of the database field.

When a data-aware entry field is attached to an active data source and the DataField property is set to the name of the field, this property is automatically set to match the data type of the database field. If it is not desirable to activate the data source, this field must be set to the correct field type so that the Orpheus entry field can configure its internal data type and initialize the picture mask. The data-aware entry fields do not support all of the possible database field types shown in the definition of TFieldType (see the online help for the VCL's DB unit). If you select one that is not supported, an error message is displayed. For example, if the ftGraphic field type is selected, an error is displayed because none of the data-aware entry fields support that field type.

See also: DataField, DataSource

UseTFieldMask property

```
property UseTFieldMask : Boolean
```

Default: False

↳ Determines which mask is used to draw the field.

If UseTFieldMask is False (the default), the entry field uses the mask defined by the PictureMask property to format and display its value. If UseTFieldMask is True, the entry field uses the mask provided by the underlying TField object to format and display its value. When the field has the focus, the PictureMask value is always used to format the display, regardless of the value of UseTFieldMask.

ZeroAsNull property

```
property ZeroAsNull : Boolean
```

Default: False

↳ Determines if zero values are stored as null.

If ZeroAsNull is set to True, fields editing a numeric data type that contain zero are stored as null database values. If False, the numeric zero is stored in the attached database field.

Setting this property to True is not necessary if you just want to initialize the database field to null. You can call the Field.Clear method to do this on an as-needed basis.

See also: Field

TOvcEfColors Class

TOvcEFColors is a class that is used to encapsulate the color settings of the Orpheus Entry Fields.

Hierarchy

TPersistent (VCL)

    TOvcEfColors (OvcEF)

Properties

Disabled	Error	Highlight

Reference Section

Disabled property

```
property Disabled : TOvcColors
```

Default: (clGrayText, clWindow)

↳ Controls the text and background colors used for entry fields that have their Enabled property set to False.

See page 1186 for additional information about the TOvcColors class.

Error property

```
property Error : TOvcColors
```

Default: (clBlack, clRed)

↳ Controls the text and background colors used for entry fields that have failed validation.

See page 1186 for additional information about the TOvcColors class.

Highlight property

```
property Highlight : TOvcColors
```

Default: (clHighlightText, clHighlight)

↳ Controls the text and background colors used for the highlighted portion of an entry field's value.

See page 1186 for additional information about the TOvcColors class.

Chapter 12: FlexEdit

The FlexEdit is yet another edit control except it has some remarkable features that make it very flexible. It descends from TEdit through TO32CustomEdit, which provides the attached label and the Orpheus About property.

TO32CustomFlexEdit Class

The TO32CustomFlexEdit class is the immediate ancestor of the TO32FlexEdit component. It implements all of the methods and properties of the TO32FlexEdit component and is identical to the TO32FlexEdit component except that none of its properties are published.

Hierarchy

TCustomEdit (VCL)

 TO32CustomEdit (O32EditF)

  TO32CustomFlexEdit (O32FlxEd)

TO32FlexEdit Component

The TO32FlexEdit component encapsulates a very flexible edit control. It has the ability to contain more than one line of data (the default is one line, no word wrap and no tabs) and can be configured to display a different number of lines while hovering the mouse over, focused and un-focused. It also contains an attached label component and a button for displaying pop-up windows, calendars, calculators, menus or anything else that you can imagine.

The FlexEdit component also implements a borders property, which allows you to enable or disable each of the four border sides. The borders can also be displayed in six different styles, Channel, Flat, Ridge, Raised, Lowered, or None. If a Flat border style is selected, then the border's color can be set using the FlatColor property. Otherwise, the border will use system colors.

It also has the ability to validate its contents via a TO32Validator descendant. See more about the TO32Validator family of components on page 453.

Furthermore, several accessor methods allow you to access the information displayed in the control in different ways—as an integer, a floating-point value, a string, a date or a time.

Hierarchy

TCustomEdit (VCL)

    TO32CustomEdit (O32EditF)

        TO32CustomFlexEdit (O32FlxEd)

            TO32FlexEdit (O32FlxEd)

Properties

AsBoolean	Borders	PopupAnchor
AsYesNo	ButtonGlyph	ShowButton
AsDateTime	DataType	Strings
AsFloat	EditLines	Validation
AsExtended	EfColors	WantReturns
AsInteger	LabelInfo	WantTabs
AsVariant	MaxLength	WordWrap

Methods

BeginUpdate	EndUpdate
ButtonClick	Restore

Events

AfterValidation	OnButtonClick	OnValidationError
BeforeValidation	OnUserValidation	

Reference Section

AfterValidation event

```
property AfterValidation : TNotifyEvent
```

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

Defines a notification event handler that is called when validation on the FlexEdit's contents is completed.

AsBoolean property

```
property AsBoolean : Boolean
```

Provides access to the contents of the FlexEdit as a Boolean.

If DataType is set to feLogical then AsBoolean gets or sets the contents of the FlexEdit as a logical (True / False) value. Otherwise calls to AsBoolean are ignored.

See also: AsYesNo

AsDateTime property

```
property AsDateTime : TDateTime
```

Provides access to the contents of the FlexEdit as a Date/Time.

If DataType is set to feDateTime then AsDateTime provides access to the contents of the FlexEdit as a TDateTime value.

AsFloat property

```
property AsFloat : Double
```

Provides access to the contents of the FlexEdit as a Float.

If DataType is set to feFloat then AsFloat provides access to the FlexEdit's contents as a floating-point value.

AsExtended property

```
property AsExtended : Extended
```

Provides access to the contents of the FlexEdit as an Extended value.

If DataType is set to feExtended then AsExtended provides access to the FlexEdit's contents as an extended value.

AsInteger property

```
property AsInteger : Integer
```

Provides access to the contents of the FlexEdit as an Integer.

↳ If DataType is set to feInteger then AsInteger provides access to the FlexEdit's contents as an Integer value.

AsVariant property

```
property AsVariant : Variant
```

↳ Provides access to the contents of the FlexEdit as a Variant.

When assigning a value to the FlexEdit using AsVariant, if the contents cannot be typecast to the type specified by DataType then it will be ignored.

AsYesNo property

```
property AsYesNo : Boolean
```

↳ Provides access to the contents of the FlexEdit as a Boolean (Yes, No) value.

If DataType is set to feLogical then AsYesNo gets or sets the contents of the FlexEdit as a logical (True / False) value. Otherwise calls to AsYesNo are ignored.

AsYesNo differes from AsBoolean in that the contents of the FlexEdit are displayed as "True" or "False" for AsBoolean and the contents are displayed as "Yes" or "No" when using AsYesNo.

See also: AsBoolean

BeforeValidation event

```
property BeforeValidation : TNotifyEvent
```

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

Defines a notification event handler that is called immediately before validation on the FlexEdit's contents is started.

BeginUpdate method

```
procedure BeginUpdate;
```

Temporarily disables repainting of the control.

Call BeginUpdate to disable repainting of the component's items if you are programmatically adding many items to a folder. Call EndUpdate after adding items to force the component to be repainted. Calls to BeginUpdate are cumulative so each call to BeginUpdate must have a corresponding EndUpdate call. The following example disables painting of the component, adds items to the first folder in the folder list, and then re-enables painting:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I  : Integer;
begin
  O32FlexEdit1.BeginUpdate;
  for I := 0 to O32FlexEdit.EditLines.MaxLines - 1 do
    O32FlexEdit1.Strings.Add('Item ' + IntToStr(I + 1));
  O32FlexEdit1.EndUpdate;
end;
```

See also: EndUpdate

Borders property

```
property Borders : TO32Borders
```

Default: Borders enabled, all borders visible and bstyRidge BorderStyle

An implementation of the TO32Borders class.

The borders property allows you to enable or disable each of the four border sides. The borders can also be displayed in six different styles, Channel, Flat, Ridge, Raised, Lowered, or None. If a Flat border style is selected, then the border's color can be set using the FlatColor property. Otherwise, the border will use current system colors.

See also: TO32Borders

ButtonClick dynamic method

```
procedure ButtonClick; dynamic;
```

↳ Provides programatic access to the FlexEdit's button.

Calling ButtonClick will cause the application to behave as if the user had clicked the button.

See also: OnButtonClick

ButtonGlyph property

```
property ButtonGlyph : TBitmap
```

Default: nil

↳ The image that appears on the FlexEdit's button.

DataType property

```
property DataType : TO32FlexEditDataType
```

```
TO32FlexEditDataType = (
  feString, feFloat, feInteger, feDateTime,
  feExtended, feStDate, feStTime, feLogical);
```

Default: feString

↳ Sets the type of data that the FlexEdit allows.

Entries which do not match the specified DataType are ignored. Calls to the different AsXxx properties which are incompatible with the current DataType are ignored.

EditLines property

```
property EditLines : TO32EditLines
```

Default: 1

↳ Implements an instance of the TO32EditLines class, which determines how the edit control will display multi-line content.

See also: TO32EditLines

EFColors property

```
property EFColors : TOvcEfColors
```

↳ Determines several color values for the associated edit control.

See "TOvcEfColors Class" on page 434 for additional information.

EndUpdate method

```
procedure EndUpdate;
```

Repaints the component after modifying the contents with painting disabled.

Call EndUpdate to repaint the component after modifying items when repainting has been disabled (by calling BeginUpdate). Each call to BeginUpdate must have a corresponding call to EndUpdate. See BeginUpdate for an example.

See also: BeginUpdate

LabelInfo property

```
property LabelInfo : TOvcLabelInfo
```

Provides a design-time interface to the Orpheus attached label class.

It allows you to decide if the attached label is visible and to fine tune the label position.

See also: TOvcLabelInfo

MaxLength property

```
property MaxLength : Integer
```

Default: 0

Determines the maximum allowable number of characters that the FlexEdit can display.

A value of 0 provides for an unlimited number of characters.

OnButtonClick event

```
property OnButtonClick : TO32feButtonClickEvent
```

```
TO32feButtonClickEvent = procedure(
  Sender: TO32CustomFlexEdit; PopupPoint: TPoint) of object;
```

Defines an event that is generated when the O32FlexEdit's button is clicked by the user.

PopupPoint will contain the X,Y coordinates to which the popup window should anchor its top, left corner. PopupPoint is determined by the PopupAnchor property.

OnButtonClick can also be fired programatically by calling the BtnClick procedure.

See also: ButtonClick, PopupAnchor

OnUserValidation event

```
property OnUserValidation : TFEUserValidationEvent
```

```
TFEUserValidationEvent = procedure(
  Sender : TObject; var ValidEntry : Boolean) of object;
```

↳ Defines an event that is generated when validation of the field's contents takes place, if ValidationType is "vtUser".

If the ValidationType is not vtUser then this event is not generated. The user should implement field content validation in the event handler and assign the results to ValidEntry.

OnValidationError event

```
property OnValidationError :  TFEValidationErrorEvent
```

```
TFEValidationErrorEvent = procedure(
  Sender : TObject; ErrorCode : Word; ErrorMsg : string) of object;
```

↳ Defines an event that is generated when validation of the field's contents results in an error.

An Details of the error will be contained in ErrorCode and ErrorMsg.

PopupAnchor property

```
property PopupAnchor : TO32PopupAnchor
```

```
TO32PopupAnchor = (paLeft, paRight);
```

Default: paLeft

↳ Determines where the popup window's top, left corner will be anchored to the FlexEdit.

The coordinates passed into the OnButtonClick event will correspond the point defined by PopupAnchor.

See also: OnButtonClick

Restore method

```
procedure Restore;
```

↳ provided for unforeseen situations where the developer needs to restore the previous contents to the FlexEdit.

When the FlexEdit receives Focus, its contents are stored in a temporary location. Calling Restore, forces the contents to be restored from that location.

ShowButton — property

```
property ShowButton : Boolean
```

Default: False

↳ Enables and disables use of the FlexEdit's button.

Strings — property

```
property Strings: TStrings
```

↳ Returns the contents of the FlexEdit or assigns new contents to the FlexEdit as TStrings.

If DataType is not feString then Strings returns nil and attempts to assign to Strings are ignored.

Validation — property

```
property Validation: TFlexEditValidatorOptions
```

↳ Groups the Validation properties together in the Object Inspector.

The TFlexEditValidatorOptions class encapsulates the properties used by the TO32FlexEdit to determine how validation of the control's contents should be handled.

See also: TFlexEditValidatorOptions, TValidatorOptions

WantReturns — property

```
property WantReturns : Boolean
```

Default: False

↳ Defines whether or not the control will accept returns into its contents.

The FlexEdit control will ignore all multi-line settings (MouseOverLines, MaxLines, FocusedLines, etc.) if WantReturns and WordWrap are both False.

See also: WantTabs, WordWrap

WantTabs property

```
property WantTabs : Boolean
```

Default: False

↳ Defines whether or not the control will accept tabs.

If WantTabs is False, pressing the tab key will result in the focus of the next control. Otherwise focus will not change and the tab character will be entered into the FlexEdit's contents.

See also: WantReturns, WordWrap

WordWrap property

```
property WordWrap : Boolean
```

Default: False

↳ Determines if the control will wrap text.

WordWrap is ignored if MaxLines is set to 1. The FlexEdit control will ignore all multi-line settings (MouseOverLines, MaxLines, FocusedLines, etc.) if WantReturns and WordWrap are both False.

See also: MaxLines, WantReturns, WantTabs

TO32EditLines Class

The TO32EditLines class encapsulates the properties used by the TO32FlexEdit to determine how multi-line content is displayed at different times. Its function is to group the line display properties into one area of the object inspector.

Hierarchy

TPersistent (VCL)

TO32EditLines (O32FlxEd)

Properties

DefaultLines	MaxLines
FocusedLines	MouseOverLines

Reference Section

DefaultLines property

```
property DefaultLines: Integer
```

Default: 1

↳ Defines how many lines are displayed when the associated control is not focused and not in mouse-over state.

FocusedLines property

```
property FocusedLines: Integer
```

Default: 1

↳ Defines how many lines are displayed when the associated control is focused.

MaxLines property

```
property MaxLines: Integer
```

Default: 1

↳ Defines how many lines of content are allowable in the associated control.

MouseOverLines property

```
property MouseOverLines: Integer
```

Default: 1

↳ Defines how many lines are displayed when the mouse hovers over the associated control when the control is not focused.

TFlexEditValidatorOptions Class

The TFlexEditValidatorOptions class descends from the TValidatorOptions class and implements an InputRequired property.

The TValidatorOptions class encapsulates the properties used by the TO32FlexEdit to determine how validation of the control's contents should be handled. Its function is to group the validation properties into one area of the object inspector.

For more information see "TValidatorOptions Class" on page 475.

Hierarchy

TPersistent (VCL)

  TValidatorOptions (O32VlOp1) 475

    TFlexEditValidatorOptions

Properties

  InputRequired

InputRequired property

```
property InputRequired: Boolean
```

Default: False.

Determines whether an empty control is allowed. The user will not be able to leave the control empty if InputRequired is True.

Chapter 13: Validators

The Orpheus Validator components provide a way to validate data-entry against a set of predefined rules. Orpheus contains validator components that can use regular expressions, Paradox masks and Orpheus masks with plans for other validators as the need arises.

The TO32BaseValidator is the ancestor of these new components. Descendants must override SetInput, SetMask, GetValid, and IsValid plus define the validation to provide full functionality.

Descendants who provide RegisterValidator and UnRegisterValidator procedures will be available as a selection in the ValidatorType property of components that use validators internally.

TO32BaseValidator Class

Hierarchy

TComponent (VCL)

  TO32Component (OvcBase) 23

    TO32BaseValidator(O32Vldtr)

Properties

About	Input	Valid
ErrorCode	Mask	

Methods

IsValid

Events

AfterValidation	BeforeValidation	OnValidationError

13

Reference Section

About — read-only property

```
property About : string
```

Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. The Orpheus About box can be displayed by double-clicking this property or selecting the dialog button to the right of the property value.

AfterValidation — event

```
property AfterValidation: TNotifyEvent
```

An event that is generated after validation takes place.

Examine the value of IsValid in descendant classes to determine whether the validation was successful.

See also: BeforeValidation, OnValidationError

BeforeValidation — event

```
property BeforeValidation : TNotifyEvent
```

An event that is generated immediately before validation takes place.

Use BeforeValidation for any pre-validation tasks.

See also: AfterValidation, OnValidationError

ErrorCode — property

```
property ErrorCode : Integer
```

Contains the code that refers to the last error condition encountered by the validation process.

If validation is successful, the value of ErrorCode is 0. Examine ErrorCode to determine what caused validation to fail.

See also: OnValidationError

Input — property

```
property Input : String
```

Default: Empty string

Input contains the string, which will be compared to the mask during validation.

This property calls SetInput internally, which is a virtual, abstract method and must be implemented in descendant classes.

IsValid — virtual abstract method

```
function IsValid : Boolean; virtual; abstract;
```

Executes the validation and returns the result.

IsValid is an abstract method that must be defined in descendant classes.

Mask — property

```
property Mask : String
```

Contains the string, which will be compared against the Input during validation.

Mask calls SetMask internally, which is a virtual, abstract method and must be implemented in descendant classes.

OnValidationError — event

```
property OnValidationError : TValidatorErrorEvent
```

```
TValidatorErrorEvent = procedure(
  Sender: TObject; const ErrorMsg: string) of object;
```

An event that is generated when the validator encounters an error during validation.

The ErrorCode property is set before OnValidationError is called.

See also: AfterValidation, BeforeValidation

Valid — property

```
property Valid : Boolean
```

Contains the result of the last validation.

Examine Valid to determine the result of the last validation. The value of Valid is good until either Mask or Input is changed at which time the value of Valid is reset to False.

TO32RegexValidator Component

The TO32RegexValidator component provides validation based on standard regular expressions. A regular expression is simply a way of describing a text pattern, which makes them ideal for validating user input.

Regular expressions are a very diverse and detailed topic, which are not covered in full in this manual. For more information on Regular Expressions see, “Mastering Regular Expressions”, by Jeffrey E.F. Friedl, O’Reilly publishing.

Hierarchy

TComponent (VCL)

    TO32Component (OvcBase) 23

        ❶ TO32BaseValidator(O32Vldtr). 454

            TO32RegexValidator(O32RxVld)

Properties

❶ About	IgnoreCase	❶ Mask
❶ ErrorCode	❶ Input	❶ Valid
ExprErrorPos	Logging	
Expression	LogFile	

Methods

❶ IsValid

Events

❶ AfterValidation	❶ BeforeValidation	❶ OnValidationError

Reference Section

ExprErrorPos property

```
property ExprErrorPos : Integer
```

↳ ExprErrorPos Contains the first character position of the expression error encountered while parsing the expression.

Examine this property to determine where the expression parser encountered an error while parsing the expression. The value of ExprErrorPos is 0 if no validation has taken place or if no expression error was detected.

Expression property

```
property Expression : String
```

↳ Contains the expression that will be compared to the value of the Input property during validation.

Mask property is still available publicly for automated use such as in the ValidatorPool. At run time you may use the Mask or Expression properties interchangeably.

See also: TO32BaseValidator.Mask

Logging property

```
property Logging : Boolean
```

Default: False

↳ Logging enables or disabled Regex parser logging.

Set to True to enable the Regex parser to create a log file of its activity. The log file contains a dump of the state machine generated when the Regex engine parses the expression. Logging is off by default and should only be turned on for debugging, or if you want to examine the state machine generated by the Regex engine for a particular expression.

See also: LogFile, TO32RegexEngine

LogFile property

```
property LogFile : String
```

Default: “parse.log”

Contains the filename of the parse log that will be created if Logging is True.

IgnoreCase property

```
property IgnoreCase : Boolean
```

Default: True

Instructs the Regex parser whether to consider case while validating.

See also: TO32RegexEngine.IgnoreCase

TO32RegexEngine Class

The TO32RegexEngine is the heart of the TO32RegexValidator component. It has been implemented as a separate class so that you can harness its power independently of the TO32RegexValidator. It is not available as a design time component except through the TO32RegexValidator component.

The TO32RegexEngine class creates an NFA state machine for each expression and optionally dumps the state machine to a log file for examination or debugging purposes.

There are two basic types of state maching. Deterministic Finite Automata (DFA) and Non-deterministic Finite Automata (NFA). The difference is that in a DFA there is only one possible path through the transitions while in an NFA, there are multiple possible paths as each step may have more than one possible transition. A NFA is necessary in applications, which require backtracking in the event of a transition failure, like a regular expression matching system.

Recommended reading: “Mastering Regular Expressions” by Jeffrey E. F. Friedl, O’Reilly publishing.

Hierarchy

TObject (VCL)

    TO32RegexEngine (O32RxNgn)

Properties

IgnoreCase	LogFile	Upcase
Logging	RegexString	

Methods

MatchString	Parse

Reference Section

IgnoreCase property

```
property IgnoreCase : Boolean
```

Default: True

↳ Determines whether the RegexEngine will consider case when matching a string against the expression.

Logging property

```
property Logging : Boolean
```

Default: False

↳ Enables the parser to create a dump file containing the details of the generated state machine.

Set Logging to True to create a log file for debugging purposes, or if you just want to learn more about how the RegexEngine's state machine works.

LogFile property

```
property LogFile : String
```

Default: "parse.log"

↳ Contains the name of the dump file that will be created if Logging is True.

If Logging is True and LogFile contains an empty string, the RegexEngine will still create a log file and call it "parse.log."

MatchString property

```
function MatchString(const S : String): Integer
```

↳ MatchString function compares a string against the RegexString.

MatchString calls Parse internally to parse the RegexString and create the state machine that will be used to compare the input string against the RegexString.

The following example from the TO32RegexValidator shows how to use the RegexEngine and its MatchString function:

```
function TO32RegExValidator.IsValid: Boolean;
var
  RegExStr: String;
begin
  DoBeforeValidation;

  {assume the worst}
  FValid := false;

  {Create a copy of the Expression and then enclose
   it in anchor operators so that the RegexEngine only
   matches the whole string...}
  RegExStr := FMask;
  {Check for Anchor operator ^}
  if (Pos('^', RegExStr) <> 1) then
    Insert('^', RegExStr, 1);
  {Check for Anchor operator $}
  if (Pos('$', RegExStr) <> Length(RegExStr)) then
    RegexStr := RegExStr + '$';

  {Pass the RegexStr into the RegexEngine}
  FRegexEngine.RegexString := RegExStr;

  if (RegExStr = '') then begin
    FErrorCode := EC_INVALID_EXPR;
    FValid := false;
  end else if (FInput = '') then begin
    FErrorCode := EC_INVALID_INPUT;
    FValid := false;
  end else begin
    if (FRegexEngine.MatchString(FInput) = 1) then begin
      FValid := true;
      FErrorCode := 0;
    end else begin
      FValid := false;
      FErrorCode := EC_NO_MATCH;
    end;
  end;

  result := FValid;

  DoAfterValidation
end;
```

```
function Parse(var aErrorPos : Integer;
  var aErrorCode: TO32RegexError) : Boolean;
```

Parses the RegexString and creates the NFA state machine that will be used in matching a string to an expression.

Parse is called internally by the MatchString function. For normal operation, you shouldn't have to call parse explicitly. If you want to simply check the validity of an expression without attempting to match a string to it, then you can set RegexString and call Parse to validate the expression. Setting the Logging property to True will cause Parse method to generate a dump file containing details on parsing process and the resulting state machine that was generated by the RegexEngine.

Following is the log result of parsing a simple regular expression.

Expression: (1[012]|[1-9]):[0-5][0-9] (am|pm)

Meaning: Time of day: Standard AM/PM time display such as 9:17 AM or 12:30 PM

```
  Parsing regex: "^(1[012]|[1-9]):[0-5][0-9] (am|pm)$"
  parsed start anchor
  parsed open paren
  parsed char: "1"
  concatenation
  parsed open square bracket (start of class)
  it is a normal class
  parsed charclass char: "0"
  parsed charclass char: "1"
  parsed charclass char: "2"
  parsed close square bracket (end of class)
  OR (alternation)
  parsed open square bracket (start of class)
  it is a normal class
  parsed charclass char: "1"
  -range to-
  parsed charclass char: "9"
  parsed close square bracket (end of class)
  parsed close paren
  concatenation
  parsed char: ":"
  concatenation
  parsed open square bracket (start of class)
  it is a normal class
```

```
parsed charclass char: "0"
-range to-
parsed charclass char: "5"
parsed close square bracket (end of class)
concatenation
parsed open square bracket (start of class)
it is a normal class
parsed charclass char: "0"
-range to-
parsed charclass char: "9"
parsed close square bracket (end of class)
concatenation
parsed char: " "
concatenation
parsed open paren
parsed char: "A"
concatenation
parsed char: "M"
OR (alternation)
parsed char: "P"
concatenation
parsed char: "M"
parsed close paren
parsed end anchor

Transition table dump for "^(1[012]|[1-9]):[0-5][0-9] (am|pm)$"
anchored at start of string
anchored at end of string
start state:   4
   0    char:1 next: 2
   1        --
   2     class next: 7
   3        --
   4  no match next: 0 5
   5     class next: 7
   6        --
   7    char:: next: 9
   8        --
   9     class next: 11
  10        --
  11     class next: 13
  12        --
  13    char:  next: 15 20
  14        --
  15    char:A next: 17
  16        --
```

```
   17    char:M next: 23
   18        --
   19        --
   20    char:P next: 22
   21        --
   22    char:M next: 23
   23*******END
```

Recommended reading: “Mastering Regular Expressions” by Jeffrey E. F. Friedl, O’Reilly publishing.

RegexString property

```
property RegexString : String
```

Default: Empty string

↳ Contains the Regular Expression that the input string will be compared against in the MatchString function.

See also: Parse, MatchString

Upcase property

```
property Upcase : TO32UpcaseChar
```

```
TO32UpcaseChar = function (aCh : char) : char;
```

↳ To convert the input string to all uppercase in the event that IgnoreCase is True.

TO32ParadoxValidator Component

The TO32ParadoxMaskValidator is a descendant of the TO32BaseValidator that validates an input string against a standard Paradox style mask. It provides the ability to validate an input against a Paradox style mask.

Hierarchy

TComponent (VCL)

  TO32Component (OvcBase). 23

    TO32BaseValidator(O32Vldtr). 454

      TO32ParadoxValidator(O32RxVld)

TO32OrMaskValidator Component

The TO32OrMaskValidator is a descendant of the TO32BaseValidator that validates an input string against an Orpheus style mask.

For more information on Orpheus style input masks see “Chapter 9: Data Entry Basics” on page 315.

Hierarchy

TComponent (VCL)

    TO32Component (OvcBase)

        TO32BaseValidator(O32Vldtr)

            TO32OrMaskValidator (O32RxVld)

TO32ValidatorPool Component

The TO32ValidatorPool component is designed to be a centralized location for all of your validation needs. It maintains a collection of validators and can create, maintain and destroy validators as it needs them.

Hierarchy

TComponent(VCL)

❶ TO32Component(OvcBase). 23

TO32ValidatorPool(O32VPool)

Properties

❶ About	Validators

Validators property

```
property Validators : TO32Validators
```

Provides access to the collection of validators.

Use Validators to add, delete and access individual validator objects maintained by the ValidatorPool.

See also: TO32Validators, TO32ValidatorItem

TO32Validators Class

The TO32Validators class maintains the ValidatorPool's collection of validators. It is exposed in the ValidatorPool via the Validators property.

Hierarchy

TCollection(VCL)

❶ TO32Collection(OvcBase) 45

TO32Validators(O32VPool)

Properties

❶ Count	Items	ValidatorPool
❶ Item	❶ Owner	
❶ ItemClass	❶ ReadOnly	

Methods

Add	DeleteByName	❶ ItemByName
Delete	GetValidatorByName	❶ ParentForm

Events

❶ OnChanged

Reference Section

Add method

```
function Add (
  ValidatorClass: TValidatorClass; Name: String): TCollectionItem;
```

↳ Adds a new validator to the list.

Add takes the name of the class to create and the desired name of the new validator. It returns a reference to the newly created validator.

See also: Delete, DeleteByName

Delete method

```
procedure Delete(Index: Integer);
```

↳ Deletes the validator specified by index.

See also: Add, DeleteByName

DeleteByName method

```
procedure DeleteByName(const Name: String);
```

↳ Deletes the validator item specified by name.

See also: Add, Delete

GetValidatorByName method

```
function GetValidatorByName(Name: String): TO32BaseValidator;
```

↳ Returns a reference to the validator item specified by name.

Items property

```
property Items[Index: Integer] : TO32ValidatorItem
```

An indexed property containing the validator items maintained by the ValidatorPool.

Use Items at run time to access a particular validator item by its index or to enumerate the validator items. The following example clears the mask property in the second validator in the list:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  O32ValidatorPool1.Validators.Items[1].Mask := '';
end;
```

ValidatorPool property

```
property ValidatorPool : TO32ValidatorPool
```

Provides access to the ValidatorPool which owns this collection of validators.

TO32ValidatorItem Class

The TO32ValidatorItem class is a wrapper around the individual validators maintained in the TO32ValidatorPool. It allows the validators to be contained in a collection and publishes the validator's access properties to the object inspector.

Hierarchy

TCollectionItem(VCL)

❶ TO32CollectionItem(OvcBase). 48

TO32ValidatorItem(O32VPool)

Properties

❶ About	❶ DisplayText	ValidationEvent
Component	❶ Name	

Reference Section

Component — property

```
property Component : TComponent
```

A reference to the component for which this validator will validate input.

The ValidatorPool's Items have the ability to hook standard components and validate their input. Use the Component property in conjunction with ValidationEvent to define how the validator will operate.

See also: ValidationEvent

ValidationEvent — property

```
property ValidationEvent : String
```

Specifies which of the hooked control's events will trigger validation.

Select an event from the list of available events in the Object Inspector. When that event is triggered in the hooked control, the associated validator will step in and perform a validation on the control's contents.

See also: Component

TValidatorOptions Class

The TValidatorOptions class encapsulates the properties used by an associated control to determine how validation of the control's contents should be handled. It also serves to group the validation properties into one area of the object inspector.

Hierarchy

TPersistent (VCL)

TValidatorOptions (O32VlOp1)

Properties

BeepOnError	ValidationEvent	ValidationType
Mask	ValidatorType	

Reference Section

BeepOnError property

```
property BeepOnError: Boolean
```

Default: False

Determines whether or not a MessageBeep will be generated on a validation error.

Mask property

```
property Mask : String
```

Contains the mask that will be used with the specified validator type to validate contents of the associated control.

If the Mask property contains an invalid picture mask then the validator will generate the appropriate error and validation will fail.

ValidationEvent property

```
property ValidationEvent: TValidationEvent
```

```
TValidationEvent = (veOnChange, veOnEnter, veOnExit);
```

Determines which event of the associated control will generate validation of the control's contents.

ValidatorType property

```
property ValidatorType : String
```

Contains the class name of the validator which will be used to validate the contents of the associated control.

Properly implemented TO32Validator descendants will register themselves with the IDE. All registered validators will appear as a selection in the object inspector under this property.

ValidatorType is ignored if the ValidationType is not vtValidator.

```
property ValidationType : TValidationType
```

```
TValidationType = (vtNone, vtUser, vtValidator);
```

Defines how validation will be performed.

The possible values for ValidationType are as follows:

Value	**Description**
vtNone	No validation will take place.
vtUser	The OnUserValidation event will be generated when the ValidationEvent occurs.
vtValidator	A TO32ValidatorDescendant of the specified type will be created and used to validate the contents of the associated control.

13

Chapter 14: Calendar Components

Orpheus provides a flexible monthly calendar (as shown in Figure 14.1)that is well suited for integration into data entry systems to allow the user to select or view a date. The data-aware version of the TOvcCalendar connects to a data source and allows visual manipulation of a Date or DateTime database field.

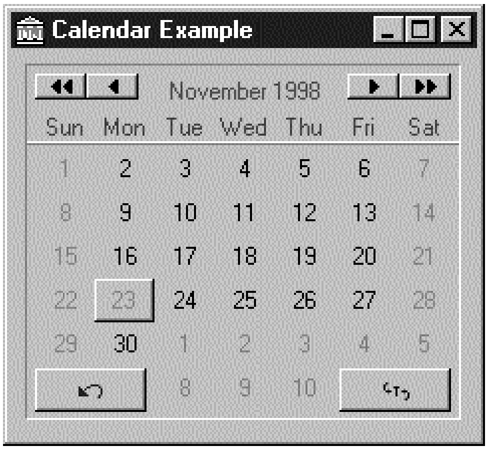


Figure 14.1: Calendar Example

The buttons at the top of the calendar allow navigating to the previous/next month (single arrow) and previous/next year (double arrows). A right-click over the name of the month displays a pop-up menu that allows selecting a month from a list that spans a seven month window (three months prior to three months after the current month).

The button at the lower-left corner is used to revert to the value the calendar last displayed. The button at the lower-right corner is used to jump to the current date.

TOvcCustomCalendar Class

The TOvcCustomCalendar class is the immediate ancestor of the TOvcCalendar component. It implements all of the methods and properties used by the TOvcCalendar component and is identical to the TOvcCalendar component except that no properties are published.

TOvcCustomCalendar is provided to facilitate creation of descendent calendar components. For property and method descriptions, see “TOvcCalendar Component” on page 481.

Hierarchy

TCustomControl (VCL)

  TOvcCustomControl (OvcBase) 16

    TOvcCustomControlEx (OvcBase) 19

      TOvcCustomCalendar (OvcCal)

TOvcCalendar Component

The TOvcCalendar component provides a grid-like display of a month with each row of the grid representing one week and each column representing a day. You can specify which day is at the left of the calendar grid and the colors for the active day, current month, day names (at the top of the grid), the days of the previous and next months, and the optional header.

Navigation through the years, months, and days is accomplished through the calendar's connection to the Controller's command processor, or by using the methods and properties provided by the TOvcCalendar component. See "TOvcController Component" on page 28for more information on using the Controller.

Calendar commands

The TOvcCalendar component provides configurable key to command translation support through the TOvcController assigned to the Controller property. The supported commands are shown in Table 14.1.

Table 15.4: *Calendar commands*

Command	**Default**	**WordStar**	**Description**
ccDown	<Down>	<CtrlX>	Move the selection to the next week (this might cause the following month to display).
ccEnd	<End>	<CtrlQ><D>	Move the selection to the last day of the month.
ccFirstPage	<CtrlHome>	not assigned	Move to the previous year, keeping the same month and day selected if possible.
ccHome	<Home>	<CtrlQ><S>	Move the selection to the first day of the month.
ccLastPage	<CtrlEnd>	not assigned	Move to the following year, keeping the same month and day selected if possible.
ccLeft	<Left>	<CtrlS>	Move the selection left one day. If the currently selected day is the first day of the month, the previous month is displayed with the last day selected.
ccNextPage	<PgDn>	<CtrlC>	Move to the next month, keeping the same day selected if possible.

Table 15.4: *Calendar commands (continued)*

Command	Default	WordStar	Description
ccPrevPage	<PgUp>	<CtrlR>	Move to the previous month, keeping the same day selected if possible.
ccRight	<Right>	<CtrlD>	Move the selection right one day. If the currently selected day is the last day of the month, the following month is displayed with the first day selected.
ccUp	<Up>	<CtrlE>	Move the selection to the previous week (this might cause the previous month to display).

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase). 16

❷ TOvcCustomControlEx (OvcBase) 19

TOvcCustomCalendar (OvcCal) 480

TOvcCalendar (OvcCal)

Properties

❶ About	❷ Controller	❶ LabelInfo
AsDateTime	Date	Month
AsStDate	DateFormat	Options
❶ AttachedLabel	Day	ReadOnly
Browsing	DayNameWidth	WantDblclicks
CalendarDate	DrawHeader	WeekStarts
Colors	IntlSupport	Year

Methods

DateString	IncMonth	SetToday
DayString	IncYear	
IncDay	MonthString	

Events

❶ AfterEnter	OnDrawDate	OnGetHighlight
❶ AfterExit	OnDrawItem	❶ OnMouseWheel
OnChange	OnGetDateEnabled	

Reference Section

AsDateTime — run-time property

```
property AsDateTime : TDateTime
```

↳ Determines the value of the calendar date as a TDateTime value.

The following example assigns the current date to Cal1:

```
  Cal1.AsDateTime := SysUtils.Date;
```

AsStDate — run-time property

```
property AsStDate : TStDate
```

↳ Determines the value of the calendar date as a TStDate value.

See also: AsDateTime, CalendarDate

Browsing — run-time, read-only property

```
property Browsing : Boolean
```

↳ indicates when calendar days are being browsed.

This property is useful to detect if a change in the calendar date is due to a user selection or because of a browse action. For example, when using the navigation buttons or when using the arrow keys to position to the correct date.

CalendarDate — run-time property

```
property CalendarDate : TStDate
```

↳ Determines the day, month, and year displayed by the calendar.

This property determines the date that the calendar displays. Assigning a value to CalendarDate causes the calendar to display that date as its active date.

If an invalid date is assigned to CalendarDate, it is ignored.

Colors — property

```
property Colors : TOvcCalendarColors
```

↳ Determines the colors used for different areas of the calendar.

See “TOvcCalColors Class” on page 493 for additional information.

Date property

```
property Date : TDateTime
```

↳ Determines the date displayed in the calendar.

DateFormat property

```
property DateFormat : TOvcDateFormat
```

```
TOvcDateFormat = (dfShort, dfLong);
```

Default: dfLong

↳ Determines the display format of the date displayed at the top of the calendar.

The value of dfShort causes the date to be displayed using the short name for the month and a value of dfLong causes the date to be displayed using the full month name.

DateString method

```
function DateString(const Mask : string): string;
```

↳ Returns a string version of the current date.

DateString calls TOvcIntlSup.DateToDateString, passing Mask and the highlighted date as the parameters. See “TOvcIntlSup Class” on page 344 for information about the DateToDateString method.

Day run-time, read-only property

```
property Day : Integer
```

Default: The current day of the month

↳ Determines the day of the month displayed by the calendar.

Assigning a valid value to this property changes the calendar’s active day. If an invalid value is assigned to Day, it is ignored.

DayNameWidth property

```
property DayNameWidth : TOvcDayNameWidth
```

```
TOvcDayNameWidth = 1..3;
```

Default: 3

Determines the length of the day name displayed along the top of the month grid.

This property controls the number of characters used to display the day of the week at the top of the calendar.

DayString method

```
function DayString : string;
```

Returns the name of the active day.

This function returns a string containing the name of the calendar's active day. It does this by retrieving the day name from the LongDayNames array, defined in the VCL SysUtils unit.

See also: MonthString

DrawHeader property

```
property DrawHeader : Boolean
```

Default: True

Determines if the calendar date and day names are drawn at the top of the calendar.

IncDay method

```
procedure IncDay(Delta : Integer);
```

Changes the calendar date by the specified number of days.

IncDay changes the calendar date by Delta number of days. Delta can be either a positive or negative value.

IncMonth method

```
procedure IncMonth(Delta : Integer);
```

Changes the calendar date by the specified number of months.

IncMonth changes the calendar date by Delta number of months. Delta can be either a positive or negative value.

IncYear method

```
procedure IncYear(Delta : Integer);
```

Changes the calendar date by the specified number of years.

IncYear changes the calendar date by Delta number of years. Delta can be either a positive or negative value.

IntlSupport run-time property

```
property IntlSupport : TOvcIntlSup
```

Default: OvcIntlSup

Provides access to the international support object that is attached to the calendar component.

A global international support object is created during initialization of the OvcIntl unit and all calendar components use it by default. You can create an additional TOvcIntlSup object, tailor the international settings (see “TOvcIntlSup Class” on page 344 for details), and attach it to the calendar component by assigning it to this property.

Month run-time, read-only property

```
property Month : Integer
```

Default: the current month

Determines the month displayed by the calendar.

Assigning a valid value to this property changes the calendar’s active month. If an invalid value is assigned to Month, it is ignored.

MonthString method

```
function MonthString : string;
```

Returns the name of the active month.

This function returns a string containing the name of the calendar’s active month. It gets the month name from the LongMonthNames array, defined in the VCL’s SysUtils unit.

See also: DayString

OnChange event

```
property OnChange : TNotifyEvent
```

Defines an event handler that is called when the calendar is changed.

The method assigned to the OnChange event is called when the calendar's active day, month, or year changes. TNotifyEvent is defined in the VCL's Classes unit.

OnDrawDate event

```
property OnDrawDate : TCalendarDateEvent
```

```
TCalendarDateEvent = procedure(
  Sender : TObject; ADate : TDateTime;
  const Rect : TRect) of object;
```

Defines an event handler that is called to allow you the opportunity to draw the date at the top of the calendar.

OnDrawItem event

```
property OnDrawItem : TDrawItemEvent
```

```
TCalendarDateEvent = procedure(
  Sender : TObject; ADate : TDateTime;
  const Rect : TRect) of object;
```

Defines an event handler that is called to allow you the opportunity to handle the task of drawing each individual day on the displayed calendar.

Sender is the instance of the calendar that requires painting, ADate indicates the date that should be drawn, and Rect is the area the you are allowed to draw in.

The ExCalOD example project demonstrates this event.

OnGetDateEnabled event

```
property OnGetDateEnabled : TGetDateEnabledEvent
```

```
TGetDateEnabledEvent = procedure(
  Sender : TObject; ADate : DateTime;
  var Enabled : Boolean) of object;
```

Defines an event handler that allows you to determine if ADate can be selected.

If Enabled is set to False, the user will not be allowed to select that particular date.

OnGetHighlight event

```
property OnGetHighlight : TGetHighlightEvent
```

```
TGetHighlightEvent = procedure(
  Sender : TObject; ADate : TDateTime;
  var Color : TColor) of object;
```

↳ Defines an event handler that allows you to determine the color used to draw the text displayed in the cell for the specified date.

Options property

```
property Options : TOvcCalDisplayOptions
```

```
TOvcCalDisplayOptions = set of TOvcCalDisplayOption;
```

```
TOvcCalDisplayOption = (
  cdoShortNames, cdoShowYear, cdoShowInactive, cdoShowRevert,
  cdoShowToday, cdoShowNavBtns);
```

Default: [cdoShortNames, cdoShowYear, cdoShowInactive, cdoShowRevert,
cdoShowToday, cdoShowNavBtns];

↳ Determines several options that determine the look of the displayed calendar.

The following table describes the possible options:

Options	**Description**
cdoShortNames	Causes dates returned by calling DateString to use short month names.
cdoShowYear	Causes the year to be displayed at the top of the calendar.
cdoShowInactive	Allows display of days that are not part of the current month.
cdoShowRevert	Displays the “revert” button, which allows changing back to the date that was previously displayed.
cdoShowToday	Displays the “today” button, which allows selecting today’s date directly.
cdoShowNavBtns	Displays navigation buttons at the top of the calendar that allows moving between months and years.

ReadOnly property

```
property ReadOnly : Boolean
```

Default: False

↳ Determines if the calendar date can be changed.

If ReadOnly is True, the user can not change the displayed date.

SetToday method

```
procedure SetToday;
```

↳ Changes the calendar's active date to today's date.

WantDblClicks property

```
property WantDblClicks : Boolean
```

Default: True

↳ Determines whether the calendar accepts mouse double clicks.

If WantDblClicks is True, the method assigned to the OnDblClick event is called when the mouse is double-clicked. If WantDblClicks is False, a double mouse click is treated as two single clicks.

WeekStarts property

```
property WeekStarts : TDayType
```

```
TDayType = (
  Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);
```

Default: Sunday

↳ Determines the day of the week that begins a week.

This property determines which day of the week is drawn in the left column of the calendar.

Year run-time, read-only property

```
property Year : Integer
```

Default: the current year

↳ Determines the year displayed by the calendar.

Assigning a valid value to this property changes the calendar's active year. If an invalid value is assigned to Year, it is ignored.

TOvcDbCalendar Component

TOvcDbCalendar is a direct descendant of the TOvcCustomCalendar and inherits all of its properties and methods.

There are very few differences in the behavior of the TOvcDbCalendar and TOvcCalendar except that TOvcDbCalendar connects to a DataSource and allows visual manipulation of a date or DateTime field.

TOvcDbCalendar provides four additional properties. AutoUpdate allows you to configure the calendar to automatically update the data source whenever the calendar loses the focus. The DataField, DataSource, and Field properties are exactly the same as the like-named properties in standard data-aware components.

To use a TOvcDbCalendar component, just assign a TDataSource to the DataSource property and assign a date field name to the DataField property. The calendar automatically gets the date from the data field and optionally updates the data source with the changed data.

Hierarchy

TCustomControl (VCL)

- ❶ TOvcCustomControl (OvcBase) 16
  - ❷ TOvcCustomControlEx (OvcBase) 19
    - TOvcCustomCalendar (OvcCal) 480
      - TOvcDbCalendar (OvcDbCal)

Properties

❶ About	❷ Controller	Field
❶ AttachedLabel	DataField	❶ LabelInfo
AutoUpdate	DataSource	

Events

❶ AfterEnter	❶ AfterExit	❶ OnMouseWheel

Reference Section

AutoUpdate — property

```
property AutoUpdate : Boolean
```

Default: True

↳ Determines whether the data source is updated when the calendar date is changed.

If AutoUpdate is True, the calendar component automatically calls UpdateRecord for the attached DataSource whenever the date is changed. Set it to False if you update the data source explicitly.

DataField — property

```
property DataField : string
```

↳ Identifies the field from which the calendar component displays data.

The field assigned to DataField must be a ftDate or ftDateTime field, or an InvalidFieldType exception is raised.

DataSource — property

```
property DataSource : TDataSource
```

↳ Specifies the data source component where the calendar obtains the data to display.

Field — run-time, read-only property

```
property Field : TField
```

↳ Returns the TField object to which the editor component is linked.

Use the Field object when you want to change the value of the data in the field programmatically. Changing the Calendar date by using any of the Calendar methods or properties will not change the underlying database field value.

TOvcCalColors Class

TOvcCalColors is a class that is used to encapsulate the color settings of the TOvcCalendar. It's properties consist primarily of several TColor values.

Hierarchy

TPersistent

TOvcCalColors

Properties

ActiveDay	Days	Weekend
ColorScheme	InactiveDays	
DayNames	MonthAndYear	

Reference Section

ActiveDay property

```
property ActiveDay : TColor
```

↳ Determines the color used for the active day cell.

ColorScheme property

```
property ColorScheme : TOvcCalColorScheme
```

```
TOvcCalColorScheme = (
  cscalCustom, cscalClassic, cscalWindows, cscalGold,
  cscalOcean, cscalRose);
```

Default: cscalWindows

↳ Allows selection of one of five predefined calendar color schemes as shown in the following table:

Value	**Color Scheme**
cscalClassic	clHighlight, clWindow, clWindow, clWindow, clWindow
cscalWindows	clRed, clMaroon, clBlack, clGray, clBlue, clRed
cscalGold	clBlack, clBlack, clYellow, clGray, clBlack, clTeal
cscalOcean	clBlack, clBlack, clAqua, clGray, clBlack, clNavy
cscalRose	clRed, clRed, clFuchsia, clGray, clBlue, clTeal

DayNames property

```
property DayNames : TColor
```

↳ Determines the color used to for display of the day names.

Days property

```
property Days : TColor
```

↳ Determines the color used to display the text in the individual day cells.

InactiveDays property

```
property InactiveDays : TColor
```

↳ Determines the color used to draw the text for day cells for months other than the current month.

MonthAndYear property

```
property MonthAndYear : TColor
```

↳ Determines the color used to draw the text displayed at the top of the calendar showing the current month and year.

Weekend property

```
property Weekend : TColor
```

↳ Determines the color used to draw the text for day cells that are either Saturdays or Sundays.

Chapter 15: Calculator

Orpheus includes a full-featured calculator that you can drop into your programs easily. Unlike the CALC.EXE program that comes with Microsoft Windows, the Orpheus Calculator component lets you place a fully functional calculator directly into your application. With the Orpheus Calculator there is no need to launch an external calculator program from your application.

In addition, the calculator in Orpheus has a number of features that let you customize its appearance and function extensively. These features include an optional tape display, complete control over the color and size of the calculator's windowed display, and a selection of optional keys that can be displayed or suppressed.

Perhaps most importantly, like all Orpheus components, the calculator is customizable in a wide variety of ways. For example, there is easy access to sending and retrieving values from the Windows Clipboard. In addition, the component's event handlers can inform you of specific key presses thereby giving you control over such things as how the results of calculations are to be handled.

TOvcCustomCalculator Class

TOvcCustomCalculator class is the immediate ancestor of the TOvcCalculator component. It implements all the methods and properties used by the TOvcCalculator component and is identical to TOvcCalculator except that no properties are published.

The TOvcCustomCalculator is provided to facilitate creation of descendent calculator components. For property and method descriptions, see “TOvcCalculator Component” on page 499.

Hierarchy

TCustomControl (VCL)

    TOvcCustomControl (OvcBase) 16

        TOvcCustomCalculator (OvcCalc)

TOvcCalculator Component

To add a calculator to your form, simply select the Orpheus Calculator Icon from the Component Palette and drop it on the form. The Calculator Component is shown in Figure 15.1.

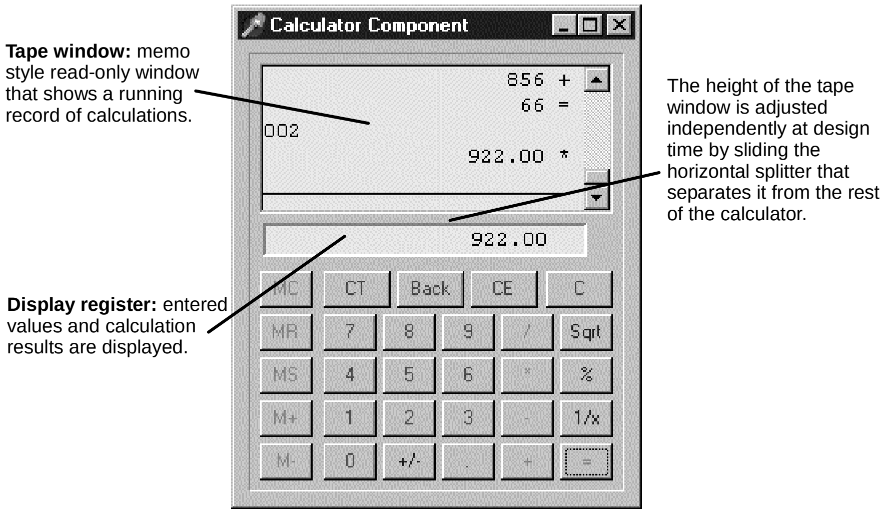

Tape window: memo style read-only window that shows a running record of calculations.
Display register: entered values and calculation results are displayed.
The height of the tape window is adjusted independently at design time by sliding the horizontal splitter that separates it from the rest of the calculator.
Calculator Component
856 +
66 =
002
922.00 *
922.00
MC
CT
Back
CE
C
MR
7
8
9
/
Sqrt
MS
4
5
6
*
%
M+
1
2
3
-
1/x
M-
0
+/-
.
+
=

Figure 15.1: Calculator Component

You can resize the calculator to almost any size, and the buttons are automatically resized to maintain the proper position and size ratio.

You can use the calculator by clicking on the desired buttons using the mouse or from the keyboard. The column of buttons on the left is used to manipulate the memory value. The <MC> button clears the memory contents (sets it to zero) and is only active when the memory value is not zero. The <MR> button recalls the contents of the memory and places it in the display. The <MS> button stores the current display value in the memory. The <M+> and <M-> buttons add or subtract the value in the display from the current memory value and place the result in the memory.

The <CT> button clears the tape window. The <Back> button deletes the last entered number or decimal point. The <CE> button clears the display, but leaves any pending operation active. The <C> button clears the display, the operation, and any pending errors.

The <Sqrt> button calculates the square root of the display value and places the result in the display. The <1/x> button calculates multiplicative inverse of the display value and places the result in the display. The <%> button causes the value in the display to be treated as a percentage, (i.e., display value = display value / 100.) The <+/-> button changes the sign of the display value.

The character displayed on the decimal button is determined by the DecimalSeparator variable in the SysUtils unit of your compiler. By default, DecimalSeparator is set to the decimal character defined in the current Windows regional settings.

Most of the calculator functions can be performed using the numeric keypad or equivalent keys on the keyboard. For functions that are not on the keyboard, keystroke equivalents are defined which are compatible with the standard Windows calculator (CALC.EXE). Table 15.1 lists the keystroke equivalents for the calculator buttons.

Table 15.5: *Calculator buttons and corresponding keystrokes*

Calculator Button	**Keystroke**
0-9	0-9
*	*
/	/
-	-
+	+
=	=
%	%
.	. or ,
Copy	<CtrlC>
Paste	<CtrlV>
MC (memory clear)	<CtrlL>
MR (memory recall)	<CtrlR>
S (memory store)	<CtrlM>
M+ (memory add)	<CtrlP>
M- (memory subtract)	<CtrlS>
Back	<Backspace>
CE (clear entry)	<Del>
C (clear)	<Esc>
CT (clear tape)	<Ctrl T>

Table 15.5: *Calculator buttons and corresponding keystrokes (continued)*

Calculator Button	Keystroke
+/- (change sign)	<F9>
Sqrt (square root)	<@>
1/x (inverse)	<R>

The memory functions and tape window are optional features that can be removed from your calculator via the Options property, as shown in Figure 15.2.

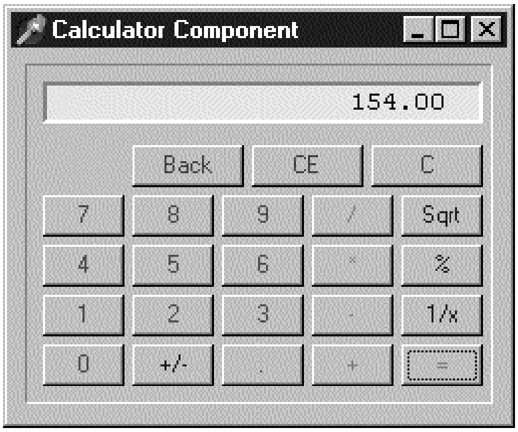


Figure 15.2: Simple calculator component

The colors used in the calculator can be customized to fit your needs. You can set any of the colors individually, or you can choose a predefined color scheme of coordinated colors. See "TOvcCalcColors Class" on page 509 for more information.

Hierarchy

TCustomControl (VCL)

- ❶ TOvcCustomControl (OvcBase) 16
  - TOvcCustomCalculator (OvcCalc) 498
    - TOvcCalculator (OvcCalc)

Properties

- ❶ About
- ❶ AttachedLabel
- Colors
- Decimals
- DisplayStr
- DisplayValue
- ❶ LabelInfo
- LastOperand
- MaxPaperCount
- Memory
- Operand
- Options
- Tape
- TapeFont
- TapeHeight
- TapeSeparatorChar

Methods

- CopyToClipboard
- PasteFromClipboard
- PressButton

Events

- ❶ AfterEnter
- ❶ AfterExit
- OnButtonPressed
- ❶ OnMouseWheel

Reference Section

Colors — property

```
property Colors : TOvcCalcColors
```

↳ Determines the colors used in the calculator.

Colors is a TOvcCalcColors class that allows you to set all the calculator colors. The properties of the TOvcCalcColors are used to set the colors for the DisabledMemoryButtons, Display, DisplayTextColor, EditButtons, FunctionButtons, MemoryButtons, NumberButtons, and OperatorButtons. You can set any of these properties individually to the desired color. Alternatively, you can choose a color scheme for all of the calculator colors. A color scheme is a pre-defined set of coordinated colors that you can choose by setting the ColorScheme property of the TOvcCalcColors class.

See "TOvcCalcColors Class" on page 509 for more information about the calculator colors.

CopyToClipboard — method

```
procedure CopyToClipboard;
```

↳ Copies the value in the calculator's display to the Window clipboard.

Decimals — property

```
property Decimals : Integer
```

Default: 2

↳ The number of digits that are displayed to the right of the decimal point.

If Decimals is non-negative, then the calculator displays a figure in floating point format, and the decimal point is not displayed if the fractional portion is zero. If Decimals is negative, then the calculator displays a figure in fixed point format. The number of decimal places is given by the absolute value of Decimals, except when Decimals equals zero. When Decimals is zero, the number of decimal places displayed are determined according to the digits entered.

DisplayStr run-time property

```
property DisplayStr : string
```

↳ Represents the contents of the calculator's display.

Assigning a value to DisplayStr is similar to pasting from the Windows clipboard. Characters are entered into the calculator just as if they were entered using the mouse or the keyboard.

DisplayValue run-time property

```
property DisplayValue : Extended
```

↳ Contains the value of the calculator's display.

If you assign a value to DisplayValue, the contents of the display are changed to that value.

LastOperand run-time property

```
property LastOperand : Extended
```

↳ Contains the second value entered for a two number operation.

In a two number operation (+, -, *, /) that require two operands, Operand contains the value of the first operand which may be the result of preceding operations and LastOperand contains the second operand. Single number operations (1/x, Sqrt, and %) operate on LastOperand. When a value is keyed in, LastOperand is updated to reflect the current number shown in the display. When an operator button is pressed, any pending two number operation is executed placing the result in Operand and the display is updated.

LastOperand is cleared by the clear entry <CE> button.

See also: Operand

MaxPaperCount property

```
property MaxPaperCount : Integer
```

Default: 9999

↳ The maximum number of tape entries to record.

MaxPaperCount specifies the number of tape entries that are recorded before displaying the red "out of paper" stripe on the tape window.

Memory — run-time property

```
property Memory : Extended
```

Contains the value of the calculator's memory.

If you assign a value to Memory, the contents of the memory are changed to that value.

OnButtonPressed — event

```
property OnButtonPressed : TOvcCalcButtonPressedEvent
```

```
TOvcCalcButtonPressedEvent = procedure(
  Sender : TObject; Button : TOvcCalculatorButton) of object;
```

```
TOvcCalculatorButton = (cbNone, cbTape, cbBack, cbClearEntry,
  cbClear, cbAdd, cbSub, cbMul, cbDiv, cb0, cb1, cb2, cb3, cb4,
  cb5, cb6, cb7, cb8, cb9, cbDecimal, cbEqual, cbInvert,
  cbChangeSign, cbPercent, cbSqrt, cbMemClear, cbMemRecall,
  cbMemStore, cbMemAdd, cbMemSub, cbSubTotal);
```

Defines an event handler that is called when a calculator button is pressed.

Sender is the instance of the calculator. Button is the button that was pressed.

For example, OnButtonPressed could be used to detect when the Equal button is pressed so that the result of the calculation can be retrieved for use elsewhere in the application.

Operand — run-time property

```
property Operand : Extended
```

Contains the result of the current chain of two number operations.

In a two number operation (+, -, *, /) that require two operands, Operand contains the value of the first operand which may be the result of preceding operations and LastOperand contains the second operand.

Single number operations (1/x, Sqrt, and %) do not operate on Operand. When an operator button is pressed, any pending two number operation is executed and the result is placed in Operand and the display is updated. The < button causes the last pending two number operation to be executed.

Operand is cleared by the clear <C> button.

See also: LastOperand

```
property Options : TOvcCalculatorOptions
```

```
TOvcCalculatorOption = (
  coShowItemCount, coShowMemoryButtons, coShowClearTapeButton,
  coShowTape, coShowSeparatePercent);
```

```
TOvcCalculatorOptions = set of TOvcCalculatorOption;
```

Default: [coShowItemCount, coShowMemoryButtons]

Is a set of flags that determine the optional features.

The possible values for Options are:

Value	Description
coShowItemCount	Display the number of items in each tape page.
coShowMemoryButtons	Display memory (<MC>, <MR>, <MS>, <M+>, <M->) buttons.
coShowClearTapeButton	Display the clear tape <CT> button.
coShowTape	Display tape window.
coShowSeparatePercent	Display a separate entry on the tape for the decimal value produced by the % button.

PasteFromClipboard method

```
procedure PasteFromClipboard;
```

↳ Copies the contents of the Windows clipboard to the calculator's display.

PressButton method

```
procedure PressButton (Button : TOvcCalculatorButton);
```

```
TOvcCalculatorButton = (
  cbNone, cbTape, cbBack, cbClearEntry, cbClear, cbAdd, cbSub,
  cbMul, cbDiv, cb0, cb1, cb2, cb3, cb4, cb5, cb6, cb7, cb8, cb9,
  cbDecimal, cbEqual, cbInvert, cbChangeSign, cbPercent, cbSqrt,
  cbMemClear, cbMemRecall, cbMemStore, cbMemAdd, cbMemSub,
  cbSubTotal);
```

↳ Causes a button on the calculator to be pressed.

The button specified by Button is pressed on the calculator just as if that button were pressed by the mouse.

The PasteFromClipboard method uses PressButton internally.

Tape run-time property

```
property Tape : TStrings
```

↳ Contains the strings displayed in the tape window.

TapeFont property

```
property TapeFont : TFont
```

↳ The font used by the tape window.

Use TapeFont when you wish to change the tape window display font.

TapeHeight property

```
property TapeHeight : Integer
```

Default: 46

↳ The height of the tape window.

TapeHeight determines the height (in pixels) of the tape window. Changing TapeHeight does not change the height of the calculator but forces the buttons and calculator display to rescale.

TapeSeparatorChar property

```
property TapeSeparatorChar : Char
```

Default: ‘_’

Specifies the character used in the separation line

The tape separation line is used to denote the end of a chain of numeric operations on the tape. A chain of operations is terminated when the equal < button is pressed.

TOvcCalcColors Class

The TOvcCalcColors class encapsulates the TColor properties used by the TOvcCalculator component. Its function is to group the color properties into one area of the object inspector.

The TOvcCalcColors class also implements pre-defined sets of coordinated colors for a calculator (see the ColorScheme property on page 510.)

Hierarchy

TPersistent (VCL)

  TOvcCalcColors (OvcCalc)

Properties

ColorScheme	DisplayTextColor	MemoryButtons
DisabledMemoryButtons	EditButtons	NumberButtons
Display	FunctionButtons	OperatorButtons

Reference Section

ColorScheme property

```
property ColorScheme : TOvcCalcColorScheme
```

```
TOvcCalcColorScheme = (cscalcCustom, cscalcWindows, cscalcDark,
  cscalcOcean, cscalcPlain);
```

Default: cscalcCustom

↳ Is a pre-defined set of colors for the calculator.

The following values are available:

Value
`cxcalcCustom`
`cscalcWindows`
`cscalcDark`
`cscalcOcean`
`cscalcPlain`

ColorScheme allows you to choose a pre-defined set of coordinated colors for all the calculator color properties. If you change any of the color properties individually, ColorScheme changes to cscalcCustom.

DisabledMemoryButtons property

```
property DisabledMemoryButtons : TColor
```

Default: clGray

↳ The color of the memory buttons when they are disabled.

See also: MemoryButtons

Display property

```
property Display : TColor
```

Default: clWindow

↳ The background color of the calculator's display area.

See also: DisplayTextColor

DisplayTextColor property

```
property DisplayTextColor : TColor
```

Default: clWindowText

↳ The color of the text in the calculator's display area.

See also: Display

EditButtons property

```
property EditButtons : TColor
```

Default: clMaroon

↳ The color of the calculator's edit buttons.

The edit buttons are the <C> (clear), <CE> (clear entry), <CT> (clear tape), and <Back> (backspace) buttons.

FunctionButtons property

```
property FunctionButtons : TColor
```

Default: clNavy

↳ The color of the calculator's function buttons.

The function buttons are the <+/->, <1/x>, <%>, and <Sqrt> buttons.

MemoryButtons property

```
property MemoryButtons : TColor
```

Default: clRed

↳ The color of the calculator's memory buttons.

The memory buttons are <MC> (memory clear), <MR> (memory recall), <MS> (memory store), <M+> (memory add), and <M-> (memory minus) buttons.

NumberButtons property

```
property NumberButtons : TColor
```

Default: clBlue

↳ The color of the calculator's number buttons.

The number buttons are the <0> through <9> and decimal point <.> buttons.

OperatorButtons property

```
property OperatorButtons : TColor
```

Default: clRed

↳ The color of the calculator's operator buttons.

The operator buttons are the <+>, <->, <*>, </>, and < buttons.

Chapter 16: Clock Components

TOvcClock is an analog clock that can display the system's current time, updated every second or a time as determined by the Time property to display schedules, meeting times, and so on. Both the clock face and the clock hands are customizable. A sample clock is shown in Figure 16.1.

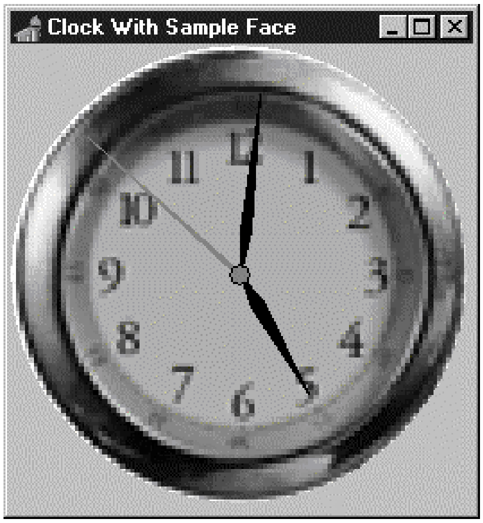


Figure 16.1: Sample clock face

TOvcCustomClock Class

The TOvcCustomClock class is the immediate ancestor of the TOvcClock component. It implements all the methods and properties used by the TOvcClock component and is identical to the TOvcClock component except that no properties are published.

TOvcCustomClock is provided to facilitate creation of descendent clock components. For property and method descriptions, see 515.

Hierarchy

TCustomControl (VCL)

  TOvcCustomControl (OvcBase) 16

    TOvcCustomControlEx (OvcBase) 19

      TOvcCustomClock (OvcClock)

TOvcClock Component

The TOvcClock component displays an analog clock on a form. The clock can be used as a real-time clock or it can be used as a static time display. When used as a real-time clock, the clock displays the system's current time, updated every second. When used as a static time display, the clock displays a time as determined by the Time property. A static time display can be used to display schedules, meeting times, and so on.

Both the clock face and the clock hands are customizable. The clock face is determined by the ClockFace property. If ClockFace is not assigned, the clock will be drawn with a default clock face. To display a picture on a clock face, assign a bitmap to the ClockFace property. The bitmap supplied will be used as the clock face with the hands drawn on top of the clock face. Orpheus comes with several clock faces that you can use in your applications. These bitmaps can be found in the Examples directory. The clock hands can be customized to modify the hand length, color, and fill style. The second hand can be shown or hidden as desired.

Figure 16.2 illustrates the use of several clock faces. Each clock shows a different time zone (the time zone offset can be set by assigning values to the TimeOffset property).

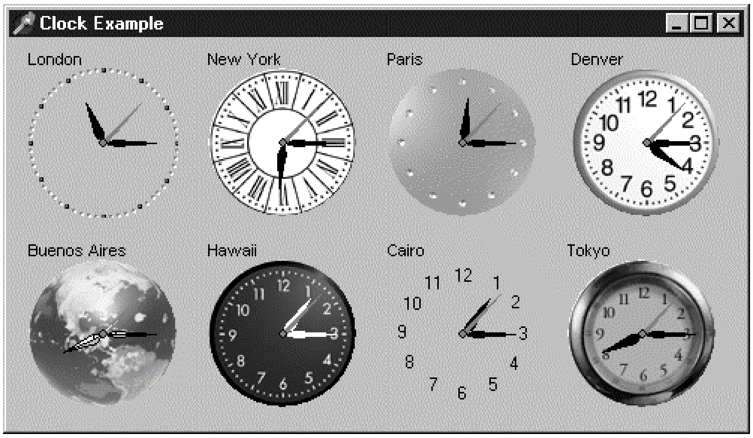


Figure 16.2: Examples of several clock faces

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase) 16

❷ TOvcCustomControlEx (OvcBase) 19

TOvcCustomClock (OvcClock) 514

TOvcClock (OvcClock)

Properties

- ❶ About
- Active
- ❶ AttachedLabel
- ClockFace
- ClockMode
- ❷ Controller
- DisplayMode
- DrawMarks
- ElapsedDays
- ElapsedHours
- ElapsedMinutes
- ElapsedSeconds
- ElapsedSecondsTotal
- HandOptions
- ❶ LabelInfo
- Time
- TimeOffset

Events

- ❶ AfterEnter
- ❶ AfterExit
- OnCountdownDone
- OnHourChange
- OnMinuteChange
- ❶ OnMouseWheel
- OnSecondChange

Reference Section

Active — property

```
property Active : Boolean
```

Default: False

Determines whether or not the clock shows the system time in real time.

The behavior of the Active property depends on the current ClockMode.

cmClock: Set Active to True to display the system time in real-time. Set Active to False if you want the clock to display a static time. Assign a TDateTime value to the Time property to set the static time.

cmTimer: Setting Active to True resets the timer at 12:00 and starts the timer counting up. Setting Active to False stops the timer. Read the ElapsedTime property to determine the elapsed time of the clock in cmTimer mode.

cmCountdownTimer: Set the start time with TimeOffset and MinuteOffset. Setting Active to True sets the timer at the time specified by TimeOffset and MinuteOffset, and begins counting down to zero. Setting Active to False before the countdown timer reaches zero will stop the timer. The timer will automatically set Active to False when it reaches zero.

See also: ClockMode, ElapsedTime, Time

ClockFace — property

```
property ClockFace : TBitmap
```

A bitmap that is displayed for the clock face.

Assign a bitmap to ClockFace to change the appearance of the clock. You can load a bitmap at design time using the ClockFace property editor, or at run time by assigning a TBitmap to ClockFace. The clock hands are drawn on top of the clock face. For aesthetic reasons you may want to set the DrawMarks property to False when supplying a clock face. Several clock faces are provided for you. You will find the clock face bitmaps in the Examples directory.

See also: DrawMarks, HandOptions

ClockMode property

```
property ClockMode : TOvcClockMode
```

```
TOvcClockMode = (cmClock, cmTimer, cmCountdownTimer);
```

Default: cmClock

↳ The mode in which the clock operates.

cmClock: The clock displays a time of day.

cmTimer: The clock shows the current elapsed time of a timer. To start a timer set ClockMode to cmTimer and the Active property to True. To stop the timer, set Active to False. Time is automatically set to 12:00 AM when the Active property is set to True if ClockMode is cmTimer. The ElapsedSeconds property can be used to determine the elapsed time when the clock is used in timer mode.

cmCountdownTimer: The clock shows the amount of time left on the timer. Set the amount of time for the countdown timer with TimeOffset and MinuteOffset properties. When the Active property set to True the clock's time is set to the defined start time and the clock begins to tick off time backwards toward zero. If the Active property is set to False before the clock reaches zero then the clock stops. When the clock reaches zero, the Active property is set to False automatically. Elapsed time can be read at any time

See also: Active, ElapsedSecondsTotal, Time

DigitalOptions property

```
property DigitalOptions : TOvcClockDigitalOptions
```

↳ Determine how the clock appears when it DisplayMode is dmDigital.

DigitalOptions is an instance of the TOvcClockDigitalOptions class. Options include the LED segment off color and LED segment on color, display back color, whether to show seconds, and whether the separator flashes with each second. See the TOvcClockDigitalOptions reference in the following section for more information on DigitalOptions.

See also: TOvcClockDigitalOptions

DisplayMode property

```
property DisplayMode : TOvcClockDisplayMode;
```

```
TOvcClockDisplayMode = (dmAnalog, dmDigital);
```

Default: dmAnalog

↳ Determines whether the clock will be displayed as a digital or analog clock.

When DisplayMode is dmDigital then the clock uses DigitalOptions to determine how it appears. Otherwise it uses HandOptions in conjunction with ClockFace and DrawMarks to determine how the clock appears.

See also: DigitalOptions

DrawMarks property

```
property DrawMarks : Boolean
```

Default: True

↳ Determines whether or not minute and hour marks are drawn around the clock face.

When DrawMarks is True, markers are drawn around the clock border. The markers consists of dots at the minute points and blue squares at the hour points. Generally you will set DrawMarks to False if you are using a bitmap as the clock face.

See also: ClockFace

ElapsedDays run-time, read-only property

```
property ElapsedDays : Integer
```

↳ Returns the number of elapsed days when ClockMode is cmTimer or cmCountdownTimer.

Read ElapsedDays to determine the number of elapsed days since the timer was started. The elapsed time (in milliseconds) is stored internally in an unsigned long integer variable so the number of elapsed days will reset to 0 after 49.7 days have elapsed.

ElapsedHours — run-time, read-only property

```
property ElapsedHours : Integer
```

↳ Returns the number of elapsed hours when ClockMode is cmTimer or cmCountdownTimer.

ElapsedHours does not return the total elapsed hours, but rather is the hours component of elapsed days, hours, minutes, and seconds. To determine the total elapsed time in hours, read the ElapsedSecondsTotal property and divide by 360.

See also: ElapsedDays, ElapsedMinutes, ElapsedSeconds, ElapsedSecondsTotal

ElapsedMinutes — run-time, read-only property

```
property ElapsedMinutes : LongInt
```

↳ Returns the number of elapsed minutes when ClockMode is cmTimer or cmCountdownTimer.

ElapsedMinutes does not return the total elapsed minutes, but rather is the minutes component of elapsed days, hours, minutes, and seconds. To determine the total elapsed time in minutes, read the ElapsedSecondsTotal property and divide by 60.

See also: ElapsedDays, ElapsedHours, ElapsedSeconds, ElapsedSecondsTotal

ElapsedSeconds — run-time, read-only property

```
property ElapsedSeconds : LongInt
```

↳ Returns the number of elapsed seconds when ClockMode is cmTimer or cmCountdownTimer.

ElapsedSeconds does not return the total elapsed seconds, but rather is the seconds component of elapsed days, hours, minutes, and seconds. To determine the total elapsed time in seconds, read the ElapsedSecondsTotal property. The elapsed time (in milliseconds) is stored internally in an unsigned long integer variable so the number of elapsed seconds will reset to 0 after 4,294,967 seconds have elapsed (49.7 days).

See also: ElapsedDays, ElapsedHours, ElapsedMinutes, ElapsedSecondsTotal

ElapsedSecondsTotal — run-time, read-only property

```
property ElapsedSecondsTotal : LongInt
```

↳ Returns the total number of elapsed seconds when ClockMode is cmTimer or cmCountdownTimer.

The elapsed time (in milliseconds) is stored internally in an unsigned long integer variable so the total number of elapsed seconds will reset to 0 after 4,294,967 seconds have elapsed (49.7 days).

HandOptions — property

```
property HandOptions : TOvcHandOptions
```

↳ Determine how the hands are drawn on the clock.

HandOptions is an instance of the TOvcHandOptions class. Options include the hand color, hand length, and whether the hand is drawn filled or hollow. See the TOvcHandOptions reference in the following section for more information on hand options.

See also: TOvcHandOptions

OnCountdownDone — event

```
property OnCountdownDone : TNotifyEvent
```

↳ Defines an event that is generated when the timer reaches zero in cmCountdownTimer mode.

Provide an event handler for the OnCountdownDone event if you want to be notified each time the countdown timer reaches zero.

OnHourChange — event

```
property OnHourChange : TNotifyEvent
```

↳ Defines an event handler that is generated when the hour changes.

Provide an event handler for the OnHourChange event if you want to be notified at the top of each hour.

See also: OnMinuteChange, OnSecondChange

OnMinuteChange event

```
property OnMinuteChange : TNotifyEvent
```

↳ Defines an event that is generated when the minute changes.

Provide an event handler for the OnMinuteChange event if you want to be notified when the minute changes.

See also: OnHourChange, OnSecondChange

OnSecondChange event

```
property OnSecondChange : TNotifyEvent
```

↳ Defines an event that is generated each second.

Provide an event handler for the OnSecondChange event if you want to be notified each time the clock's second hand changes.

See also: OnHourChange, OnMinuteChange

Time property

```
property Time : TDateTime
```

↳ The clock's current time.

Read Time to determine the clock's current time. Set Time to a TDateTime value to display a static time on the clock when the Active property is False. The following example sets the clock's time to 1:30 PM:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  OvcClock1.Active := False;
  OvcClock1.Time := EncodeTime(13, 30, 0, 0);
end;
```

See also: Active

TimeOffset property

```
property TimeOffset : Integer
```

Default: 0

↳ The hour offset from the current time zone.

The clock's time is taken from the system's current time. When TimeOffset is 0, the current local time is displayed. To display times in other time zones, set TimeOffset to the relative time zone you wish to display. If, for example, you are in the United States' Mountain time zone and you want to display a clock showing Pacific time you would set TimeOffset to –1. To display Eastern time (two time zones earlier) you would set TimeOffset to 2.

See also: Time

TOvcDbClock Component

TOvcDbClock is a direct descendant of the TOvcCustomClock and inherits all of its properties and methods, which are documented in the TOvcClock section.

There are very few differences in the behavior of the TOvcDbClock and TOvcClock except that TOvcDbClock is capable of connecting to a data source and displaying the time values of a field within the attached database.

TOvcDbClock provides three additional properties. The DataField, DataSource, and Field properties are exactly the same as the like-named properties in VCL's standard data-aware components.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase). 16

❷ TOvcCustomControlEx (OvcBase) 19

TOvcCustomClock (OvcClock). 514

TOvcDbClock (OvcDbClk)

Properties

❶ About	DataField	❶ LabelInfo
❶ AttachedLabel	DataSource	
❷ Controller	Field	

Events

❶ AfterEnter	❶ AfterExit	❶ OnMouseWheel

Reference Section

DataField property

```
property DataField : string
```

↳ Identifies the field (in the data source component) from which the clock component displays data.

DataSource property

```
property DataSource : TDataSource
```

↳ Specifies the data source component where the clock obtains the data to display.

Field property

```
property Field : TField
```

↳ Returns the TField object to which the clock component is linked.

Use the Field object when you want to change the value of the data in the field programmatically.

TOvcHandOptions Class

TOvcHandOptions determines how a TOvcClock's hands are drawn. Clock hand options are surfaced in the TOvcClock component's HandOptions property. The options allow you to control the appearance of the second hand, the minute hand, and the hour hand. You can set the hand colors, their length and width, and whether they are filled or outlined. By default the hour and minute hands are black and the second hand is red. The hand length is automatically adjusted when the size of the clock changes.

Hierarchy

TPersistant (VCL)

  TOvcHandOptions (OvcClock)

Properties

HourHandColor	MinuteHandLength	SecondHandWidth
HourHandLength	MinuteHandWidth	ShowSecondHand
HourHandWidth	SecondHandColor	SolidHands
MinuteHandColor	SecondHandLength	

Reference Section

HourHandColor property

```
property HourHandColor : TColor
```

Default: clBlack

↳ Determines the color of the hour hand.

HourHandLength property

```
property HourHandLength : TOvcPercent
```

```
TOvcPercent = 0..100;
```

Default: 60

↳ Determines the length of the hour hand as a percentage of the clock's diameter.

Once the hand length is set it will retain it's length relative to the clock face if the clock is resized.

HourHandWidth property

```
property HourHandWidth : Integer
```

Default: 4

↳ Determines the width of the clock's hour hand.

MinuteHandColor property

```
property MinuteHandColor : TColor
```

Default: clBlack

↳ Determines the color of the clock's minute hand.

MinuteHandLength property

```
property MinuteHandLength : TOvcPercent
```

```
TOvcPercent = 0..100;
```

Default: 80

Determines the length of the minute hand as a percentage of the clock's diameter.

Once the hand length is set it will retain it's length relative to the clock face if the clock is resized.

MinuteHandWidth property

```
property MinuteHandWidth : Integer
```

Default: 3

Determines the width of the clock's minute hand.

SecondHandColor property

```
property SecondHandColor : TColor
```

Default: clRed

Determines the color of the clock's second hand.

SecondHandLength property

```
property SecondHandLength : TOvcPercent
```

```
TOvcPercent = 0..100;
```

Default: 90

Determines the length of the second hand as a percentage of the clock's diameter.

Once the hand length is set it will retain it's length relative to the clock face if the clock is resized. To eliminate the second hand, set ShowSecondHand to False.

See also: ShowSecondHand

SecondHandWidth property

```
property SecondHandWidth : Integer
```

Default: 1

Determines the width of the clock's second hand.

ShowSecondHand property

```
property ShowSecondHand : Boolean
```

Default: True

Determines whether the clock's second hand is displayed.

SolidHands property

```
property SolidHands : Boolean
```

Default: True

Determines the drawing style used to paint the clock hands.

If SolidHands is True, hands are filled with the current hand color. If SolidHands is False, clock hands are outlined with the hand color and the interior of the hand is drawn in a 3D style.

TOvcClockDigitalOptions Class

TOvcClockDigitalOptions determines how the clock is drawn in dmDigital mode. DigitalOptions are surfaced in the OvcClock component's DigitalOptions property. The options allow you to control the LED segment colors, display back color, whether to show seconds, and whether the separator flashes with each second.

Hierarchy

TPersistant (VCL)

TOvcClockDigitalOptions (OvcClock)

Properties

LEDOnColor	BackColor	FlashSeparator
LEDOffCOlor	ShowSeconds	

Reference Section

LEDOnColor property

```
property LEDOnColor : TColor
```

Default: clLime

Determines the color of the lit LED segments.

LEDOffColor property

```
property LEDOffColor : TColor
```

Default: $00274026 (a really dark green)

Determines the color of the unlit LED segments.

BackColor property

```
property BackColor : TColor
```

Default: clBlack

Determines the color of the background area behind the LED segments.

ShowSeconds property

```
property ShowSeconds : Boolean
```

Default: True

Determines whether the seconds are visible on the display.

FlashSeparator property

```
property FlashSeparator : Boolean
```

Default: True

Determines whether the separator will flash every second.

Chapter 17: Combo Box Components

Orpheus provides several ComboBox components that provide an easy mechanism for selecting fonts, icons, printers, directories and more, from drop-down lists.

- The TOvcAssociationComboBox displays a list of the registered file associations found in the Windows registry or WIN.INI.
- The TOvcColorComboBox provides an easy method for selecting one of sixteen predefined colors. The data-aware version, TOvcDbColorComboBox, connects to a data source and allows visual manipulation of an index for a color in the drop-down list of colors.
- The TOvcComboBox provides all the functionality of the TComboBox component plus an auto-search facility, the ability to keep track and display a list of the most-recently- used (MRU) items, and the ability to create an attached label.
- The TOvcDbAliasComboBox displays a list of database alias names defined by the BDE.
- The TOvcDbFieldComboBox connects to a data source and displays a filtered list of the field names of an active data set.
- The TOvcDbTableNameComboBox displays a list of tables associated with a specified database.
- The TOvcDirectoryComboBox reads the directory structure of the current drive.
- The TOvcDriveComboBox displays a list of available drives.
- The TOvcFileComboBox displays a filtered list of files in a specified directory.
- The TOvcFontComboBox displays a list of installed fonts.
- The TOvcHistoryComboBox implements a history selection list.
- The TOvcPrinterComboBox displays a list of the printers installed in Windows and optionally allows the user to select a printer.

Each of the Orpheus ComboBox components descends from a common class, TOvcBaseComboBox. TOvcBaseComboBox implements the general look of the combobox and defines some basic functionality such as AutoSearch and a most-recently-used (MRU) list.

The ExCbx example program demonstrates the components described in this section.

TOvcBaseComboBox Class

TOvcBaseComboBox class is an abstract class derived from the VCL TCustomComboBox class. It serves as the ancestor class for the other Orpheus combo box components and contains their common properties and methods. It is identical to TOvcComboBox (see page 554) except that no properties are published.

The TOvcBaseComboBox provides all the functionality of the TComboBox component plus it provides an auto-search facility, the ability to keep track and display a list of the most-recently-used (MRU) items, and the ability to create an attached label.

The auto-search facility allows the TOvcBaseComboBox to automatically search its drop-down list as the user enters text into the edit box, and appends the remaining characters of the matched item string. The appended characters are highlighted to make it easy to see the current match and continue entering additional text.

The TOvcBaseComboBox keeps track of the items most-recently-used (MRU) and displays them at the top of the drop-down list. The number of items tracked is configurable and the background color of the MRU list can be set independently to distinguish these items from the rest of the drop-down list.

An attached label is a label that maintains its position relative to the TOvcBaseComboBox. This feature is most useful at design-time when you are moving components to arrange the look of the form.

Caution: The MRU list is included at the beginning of the Items list, which will cause the index of an item in the "normal" portion of the drop-down list to vary depending upon how many items are in the MRU list. For this reason, two additional properties, List and ListIndex, are provided which reference only the items of the "normal" drop-down and thus should be used instead of Items and ItemIndex to reference an item relative to the top of the "normal" list.

The TOvcBaseComboBox is provided to facilitate the creation of descendent combo box components.

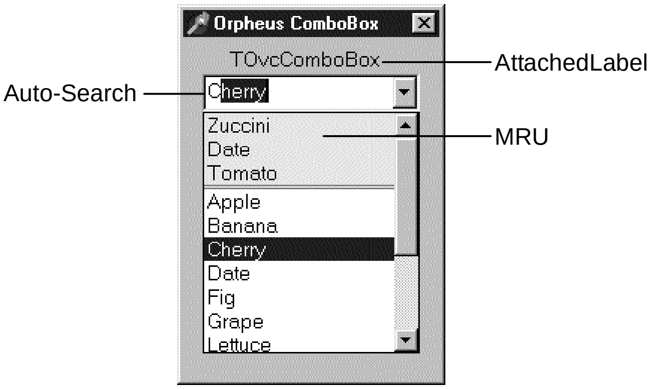


Figure 17.1: Orpheus ComboBox

Hierarchy:

TCustomComboBox (VCL)

  TOvcBaseComboBox (OvcCmBx)

Properties

About	HotTrack	ListIndex
AttachedLabel	ItemHeight	MRUListColor
AutoSearch	KeyDelay	MRUListCount
Controller	LabelInfo	Style
DroppedWidth	List	

Methods

AddItem	ClearMRUList	SelectionChanged
AssignItems	InsertItem	
ClearItems	RemoveItem	

Events

AfterEnter	OnMouseWheel
AfterExit	OnSelectionChange

Reference Section

About — read-only property

```
property About : string
```

Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

AddItem — method

```
function AddItem(
  const Item : string; AObject : TObject) : Integer;
```

AddItem adds an item to the drop-down list.

AddItem adds the Item to the list and returns its new index. Use AddItem to add an item to the drop-down list at run time rather than the AddObject method of the Items property.

See also: List

AfterEnter — event

```
property AfterEnter : TNotifyEvent
```

Defines an event handler that is called after the component receives the focus.

Unlike the OnEnter event (in some versions of Delphi and C++Builder), which is called during the focus change, AfterEnter is not called until after the focus change is complete

TNotifyEvent is defined in the VCL's Classes unit.

AfterExit — event

```
property AfterExit : TNotifyEvent
```

Defines an event handler that is called after the component receives the focus.

Unlike the OnExit event (in some versions of Delphi and C++Builder), which is called during the focus change, AfterExit is not called until after the focus change is complete

TNotifyEvent is defined in the VCL's Classes unit.

AssignItems method

```
procedure AssignItems(Source : TPersistent)
```

↳ Sets the strings and associated objects in the drop-down list.

AssignItems sets the value of the List object from another object and clears the most-recently- used list. If associated objects are supported, any associated objects are copied from the Source as well. Use AssignItems to set the drop-down list at run time rather than the Items Assign method.

See also: List

AttachedLabel run-time, read-only property

```
property AttachedLabel : TOvcAttachedLabel
```

↳ Provides access to the TOvcAttachedLabel component.

The TOvcAttachedLabel is normally accessed only at design time, but this property makes it possible to alter the attached label at run time if necessary.

See page 762 for more information on the TOvcAttachedLabel component.

AutoSearch property

```
property AutoSearch : Boolean
```

Default: True

↳ Determines whether auto-search of the drop-down list is enabled.

If AutoSearch is True, the drop-down list is searched for the entered text as soon as KeyDelay milliseconds have passed since the last character was entered.

See also: KeyDelay

ClearItems method

```
procedure ClearItems;
```

↳ Deletes the items from the drop-down list not including the MRU list.

Use ClearItems if you wish to clear the normal drop-down list, but not the MRU list. To clear all items in the drop-down list use the Clear method.

ClearMRUList method

```
procedure ClearMRUList;
```

↳ Deletes the items from the MRU list.

Use ClearMRUList to clear the MRU list portion of the drop-down list leaving the “normal” drop-down list intact.

Controller property

```
property Controller : TOvcController
```

Default: The first TOvcController object on the form.

↳ The TOvcController object that is attached to this component.

If this property is not assigned, some or all of the features provided by the component will not be available.

DroppedWidth property

```
property DroppedWidth : Integer
```

Default: -1

↳ The maximum allowable width, in pixels, of the drop-down list box.

Use DroppedWidth to change the width of the list box. When DroppedWidth is -1, the maximum width of the drop-down list is determined by Width.

HotTrack property

```
property HotTrack: Boolean
```

Default: False

↳ Determines whether the combo box is drawn with a flat-style look.

When HotTrack is set to True, the combo box has a flat style appearance until the mouse passes over it when it appears in 3-D.

InsertItem method

```
procedure InsertItem(
  Index : Integer; const Item : string; AObject : TObject);
```

↳ Inserts an item into the drop-down list.

InsertItem inserts an item into the list at the specified index. Use InsertItem to insert an item into the drop-down list at run time rather than the InsertObject method of the Items property.

See also: List

ItemHeight property

```
property ItemHeight: Integer
```

Default: 14

↳ The height, in pixels, of the items in the drop-down list.

By changing ItemHeight you can control the height of the rectangle used to display each item in the drop-down list box.

KeyDelay property

```
property KeyDelay : Integer
```

Default: 500

↳ The number of milliseconds to wait after keyboard entry before performing an auto-search.

This property is used only if AutoSearch is True.

See also: AutoSearch

LabelInfo property

```
property LabelInfo : TOvcLabelInfo
```

↳ Provides access to the status of the attached label.

TOvcLabelInfo (see page 764) groups the Visible, OffsetX, and OffsetY properties that determine whether the attached label is visible and where it is positioned.

List — read-only property

```
property List : TStrings
```

↳ Provides access to the items in the drop-down list not including the MRU list.

Use List to access items in the regular drop-down list at run time rather than the VCL's Items property since Items also includes the most-recently-used list. An item's location in the Items string list will vary depending upon the size of the MRU list. List contains only those items in the normal portion of the drop-down list and hence an item's location in the list remains constant until it is altered by the insertion or deletion of an item. The currently selected item in List is given by ListIndex. At run time, the drop-down list should only be updated by using the AddItem, AssignItems, ClearItems, InsertItem, and RemoveItem methods. Otherwise, you may get unpredictable results depending on the size of the MRU list.

See also: AddItem, AssignItems, ClearItems, InsertItem, ListIndex, RemoveItem

ListIndex — property

```
property ListIndex : Integer
```

↳ The ordinal number of the selected item in List

Use ListIndex to obtain or set the index of the selected item in the regular drop-down list at run time rather than ItemIndex. ItemIndex references the VCL's Items property, which includes the most-recently-used items list. If no item is selected, the value of ListIndex is -1.

See also: List

MRUListColor — property

```
property MRUListColor : TColor
```

Default: clWindow

↳ Determines the background color of the most-recently-used items.

MRUListColor can be used to highlight the most-recently-used items that are shown in the list portion of the combo box. By default, MRUListColor is the same as the color used by the normal part of the combo box control.

See also: MRUListCount, ClearMRUList

MRUListCount property

```
property MRUListCount : Integer
```

Default: 3

Determines the maximum number of items kept in the MRU list.

OnMouseWheel event

```
property OnMouseWheel : TMouseWheelEvent
```

Defines an event handler that is called when a mouse wheel message is received.

OnSelectionChange event

```
property OnSelectionChange : TNotifyEvent
```

Defines an event handler that is called when a new item is selected.

By specifying a handler for this event, you can be notified whenever a selection change is made.

RemoveItem method

```
procedure RemoveItem(const Item : string);
```

Removes an item from the drop-down list and updates the MRU list.

If the specified item is currently selected, the SelectionChanged event is fired. Use RemoveItem to delete an item from the drop-down list at run time rather than the Delete method of the VCL's Items property.

SelectionChanged virtual method

```
procedure SelectionChanged; virtual;
```

Updates the MRU list and fires the OnSelectionChanged event.

See also: OnSelectionChanged

Style property

```
property Style : TOvcComboStyle
```

```
TOvcComboStyle = (ocsDropDown, ocsDropDownList);
```

Default: ocsDropDown

↳ Determines whether an edit box is created to allow typing in the field.

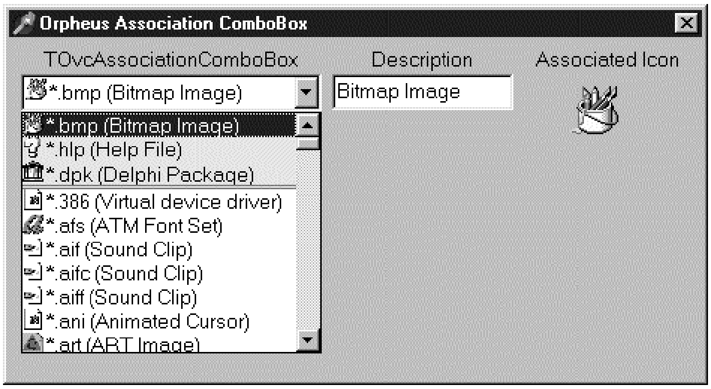
When the value of Style is ocsDropDown, the user can select an item from the drop-down list and change the value of the field or type a new value in the edit box. The AutoSearch facility only applies when Style is ocsDropDown. When Style is ocDropDownList, the user can change the contents of the field only by selecting one of the items from the drop-down list.

See also: AutoSearch

TOvcAssociationComboBox Component

The TOvcAssociationComboBox is a specialized combo box that presents the user with a list of the registered file types found in the Windows registry (HKEY_CLASSES_ROOT) or the WIN.INI ([Extensions]). Each item in the list includes file extension, file type description, and default icon.

The TOvcAssociationComboBox also includes auto-search facility, most-recently-used items, and attached label.



*.bmp (Bitmap Image)

Bitmap Image

*.bmp (Bitmap Image)
*.hlp (Help File)
*.dpk (Delphi Package)
*.386 (Virtual device driver)
*.afs (ATM Font Set)
*.aif (Sound Clip)
*.aifc (Sound Clip)
*.aiff (Sound Clip)
*.ani (Animated Cursor)
*.art (ART Image)

Figure 17.2: Orpheus Association ComboBox

Hierarchy

TCustomComboBox (VCL)

❶ TOvcBaseComboBox (OvcCmBx). 534

TOvcAssociationComboBox (OvcFMCbx)

17

Properties

- ❶ About
- AssociatedIcon
- ❶ AttachedLabel
- ❶ AutoSearch
- ❶ Controller
- Description
- ❶ DroppedWidth
- ❶ HotTrack
- ❶ ItemHeight
- ❶ KeyDelay
- ❶ LabelInfo
- ❶ List
- ❶ ListIndex
- ❶ MRUListColor
- ❶ MRUListCount
- ❶ Style

Methods

- ❶ AddItem
- ❶ AssignItems
- ❶ ClearItems
- ❶ ClearMRUList
- ❶ InsertItem
- ❶ RemoveItem
- ❶ SelectionChanged

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❶ OnMouseWheel
- ❶ OnSelectionChange

Reference Section

AssociatedIcon — read-only property

```
property AssociatedIcon : HIcon
```

Contains the handle for the default icon of the selected item.

Use AssociatedIcon to obtain the icon handle of the default icon for the selected file extension. For example, you could display the associated icon with a TImage component when the selection changes by the following:

```
procedure
  TForm1.OvcAssociationSelectionChange(Sender : TObject);
begin
  Image1.Picture.Icon.Handle := OvcAssociation.AssociatedIcon;
end;
```

Description — read-only property

```
property Description : string
```

Contains the file type description of the selected file extension.

Use Description to obtain the file type description for the selected file extension. For example, you could display the description in a form's caption when the selection changes by the following:

```
procedure
  TForm1.OvcAssociationSelectionChange(Sender : TObject);
begin
  Caption := OvcAssociation.Description;
end;
```

TOvcCustomColorComboBox Class

The TOvcCustomColorComboBox class is the immediate ancestor of the TOvcColorComboBox component. It implements all of the methods and properties used by the TOvcColorComboBox component and is identical to the TOvcColorComboBox component except that no properties are published.

TOvcCustomColorComboBox is provided to facilitate creation of descendent Color Combo Box components. For property and method descriptions, see “TOvcColorComboBox Component” on page 547.

Hierarchy

TCustomComboBox (VCL)

❶ TOvcBaseComboBox (OvcCmBx). 534

TOvcCustomColorComboBox (OvcClrCb)

TOvcColorComboBox Component

The TOvcColorComboBox component is derived from TOvcCustomComboBox and inherits its properties, methods, and events. It provides an easy method for selecting one of sixteen predefined colors: Black, Maroon, Green, Olive, Navy, Purple, Teal, Gray, Silver, Red, Lime, Yellow, Blue, Fuchsia, Aqua, and White.

Example

This example uses a Color Combo Box to select a color. The background color of a label is changed to the color selected from the Color Combo Box.

Create a new project, add components, and set the property values as indicated in Table 17.1.

Component	**Property**	**Value**
`TOvcColorComboBox`		
`TLabel`	`Caption`	`‘Selected color:’`
`TLabel`		

Click on the TOvcColorComboBox to select it. Double-click on the OnChange event in the Events tab of the Object Inspector to add the following event handler:

```
procedure TForm1.OvcColorComboBox1Change(Sender : TObject);
begin
  Label2.Color := OvcColorComboBox1.SelectedColor;
end;
```

Compile and run the project. It should look something like Figure 17.3.

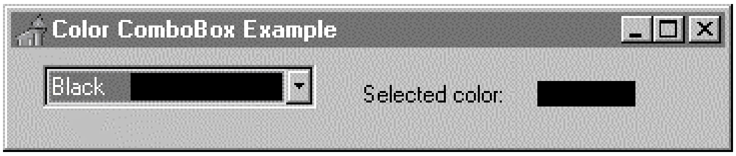


Figure 17.3: Color ComboBox Example

When you select a color, the label is displayed in the new color.

Hierarchy

TCustomComboBox (VCL)

❶ TOvcBaseComboBox (OvcCmBx). 534

TOvcCustomColorComboBox (OvcClrCb). 546

TOvcColorComboBox (OvcClrCb)

Properties

- ❶ About
- ❶ AttachedLabel
- ❶ AutoSearch
- ❶ Controller
- ❶ DroppedWidth
- ❶ HotTrack
- ❶ ItemHeight
- ❶ KeyDelay
- ❶ LabelInfo
- ❶ List
- ❶ ListIndex
- ❶ MRUListColor
- ❶ MRUListCount
- SelectedColor
- ShowColorNames
- ❶ Style

Methods

- ❶ AddItem
- ❶ AssignItems
- ❶ ClearItems
- ❶ ClearMRUList
- ❶ InsertItem
- ❶ RemoveItem
- ❶ SelectionChanged

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❶ OnMouseWheel
- ❶ OnSelectionChange

Reference Section

SelectedColor property

```
property SelectedColor : TColor
```

The color that is selected in the combo box.

If you assign a TColor value to SelectedColor, the corresponding entry in the list is selected. If the specified TColor is not in the Color Combo Box list, the assignment is ignored and the current selection is unchanged.

ShowColorNames property

```
property ShowColorNames : Boolean
```

Default: True

Determines whether the names of the colors are displayed to the left of the colors in the Color Combo Box.

TOvcDbColorComboBox Component

The TOvcDbColorComboBox component is derived from TOvcCustomColorComboBox and inherits its properties, methods and events. It provides an easy method for selecting one of sixteen predefined colors: Black, Maroon, Green, Olive, Navy, Purple, Teal, Gray, Silver, Red, Lime, Yellow, Blue, Fuchsia, Aqua, and White.

There are very few differences in the behavior of the DbColor Combo Box and Custom Color Combo Box except that DbColor Combo Box connects to a data source and allows visual manipulation of an index for a color in the drop-down list of colors.

DbColor Combo Box provides three additional properties. The DataField, DataSource, and Field properties are exactly the same as the like-named properties in VCL's standard data-aware components.

To use a TOvcDbColorComboBox component, just assign a TDataSource to the DataSource property and assign a numeric field name to the DataField property. The combo box automatically gets the color index from the data field and updates the data source with the changed data.

Hierarchy

TCustomComboBox (VCL)

❶ TOvcBaseComboBox (OvcCmBx). 534

TOvcCustomColorComboBox (OvcClrCb) 546

TOvcDbColorComboBox (OvcClrCb)

Properties

- ❶ About
- ❶ AttachedLabel
- ❶ AutoSearch
- ❶ Controller
- DataField
- DataSource
- ❶ DroppedWidth
- Field
- ❶ HotTrack
- ❶ ItemHeight
- ❶ KeyDelay
- ❶ LabelInfo
- ❶ List
- ❶ ListIndex
- ❶ MRUListColor
- ❶ MRUListCount
- ❶ Style

Methods

- ❶ AddItem
- ❶ AssignItems
- ❶ ClearItems
- ❶ ClearMRUList
- ❶ InsertItem
- ❶ RemoveItem
- ❶ SelectionChanged

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❶ OnMouseWheel
- ❶ OnSelectionChange

Reference Section

DataField property

```
property DataField : string
```

Identifies the field from which the Color Combo Box component displays data.

The field assigned to DataField must be a numeric field, or an InvalidFieldType exception is raised.

DataSource property

```
property DataSource : TDataSource
```

Specifies the data source component where the Color Combo Box obtains the data to display.

Field run-time, read-only property

```
property Field : TField
```

Returns the TField object to which the Color Combo Box component is linked.

Use the Field object when you want to change the value of the data in the field programmatically.

TOvcComboBox Component

The TOvcComboBox (shown in Figure 17.4) is a general purpose combo box that includes an auto-search facility to automatically search the drop-down list as the user enters text, a most-recently-used (MRU) items list that is displayed at the top of the drop-down list, and an attached label that maintains its position relative to the TOvcComboBox. All of the properties, methods and events in TOvcComboBox are inherited from the TOvcBaseComboBox Class (see page 534).

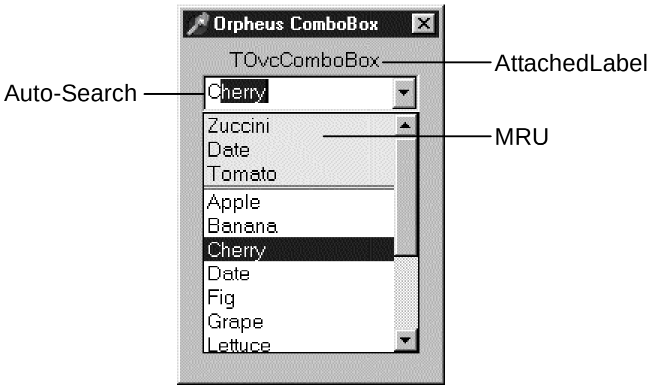


Figure 17.4: Orpheus ComboBox

Hierarchy

TCustomComboBox (VCL)

❶ TOvcBaseComboBox (OvcCmBx)..534

TOvcComboBox (OvcCmBx)

TOvcDbAliasComboBox Component

The TOvcDbAliasComboBox (shown in Figure 17.5) is a specialized data-aware combo box that displays a list of BDE-defined database alias names. The TableNameComboBox property allows you to synchronize the Alias Combo Box with a TOvcDbTableNameComboBox (see page 563).

The TOvcDbAliasComboBox also includes auto-search facility, most-recently-used items, and attached label.

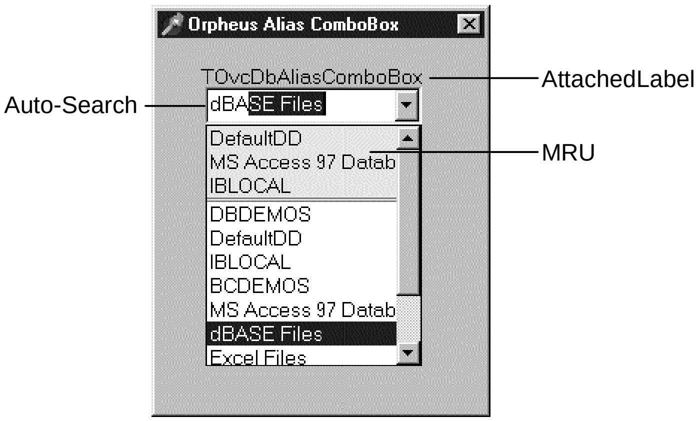


Figure 17.5: Orpheus Alias ComboBox

Hierarchy

TCustomComboBox (VCL)

❶ TOvcBaseComboBox (OvcCmBx). 534

TOvcDbAliasComboBox (OvcDbACb)

Properties

- ❶ About
- AliasName
- ❶ AttachedLabel
- ❶ AutoSearch
- ❶ Controller
- DbEngineHelper
- DriverName
- ❶ DroppedWidth
- ❶ HotTrack
- ❶ ItemHeight
- ❶ KeyDelay
- ❶ LabelInfo
- ❶ List
- ❶ ListIndex
- ❶ MRUListColor
- ❶ MRUListCount
- Path
- ❶ Style
- TableNameComboBox

Methods

- ❶ AddItem
- ❶ AssignItems
- ❶ ClearItems
- ❶ ClearMRUList
- ❶ InsertItem
- ❶ RemoveItem
- ❶ SelectionChanged

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❶ OnMouseWheel
- ❶ OnSelectionChange

Reference Section

AliasName property

```
property AliasName : string
```

↳ Contains the selected database alias.

DbEngineHelper property

```
property DbEngineHelper : TOvcDbEngineHelperBase
```

↳ Identifies a component that acts as source for information specific to different database systems.

This component uses the component assigned to DbEngineHelper to assist in obtaining information from the database system in a generic way. It's primary purpose is to allow this and other data-aware Orpheus components to work with third-part database systems.

If you are using the BDE, place a TOvcDbBDEHelper component on the form and assign its name to this property. Additional "Helper" components designed to support other database systems are available. See the DBHELPER.HLP in the main Orpheus directory for additional information.

A value must be assigned to this property.

DriverName property

```
property DriverName : string
```

↳ Contains the BDE driver for the selected database alias.

Path property

```
property Path : string
```

↳ Contains the directory pathname of the selected database alias.

TableNameComboBox property

```
property TableNameComboBox : TOvcTableNameComboBox
```

Specifies a synchronized Table Name Combo Box.

The TableNameComboBox property provides the mechanism to associate the Alias Combo Box with an Orpheus Table Name Combo Box. If a Table Name Combo Box is specified then when the alias selection changes, the AliasName property of the Table Name Combo Box will be updated as well so it can display the table names associated with the selected database.

See also: TOvcTableNameComboBox

TOvcDbFieldComboBox Component

The TOvcDbFieldComboBox (shown in Figure 17.6) is a specialized data-aware combo box that is capable of connecting to a data source and displaying a filtered list of the field names of an active data set. The selected field name is accessed via the FieldName property. The drop-down list can be filtered by specifying the type of data fields to omit via the OmitFields property. Hidden field names can optionally be included in the drop-down list.

The TOvcDbFieldComboBox also includes auto-search facility, most-recently-used items, and attached label.

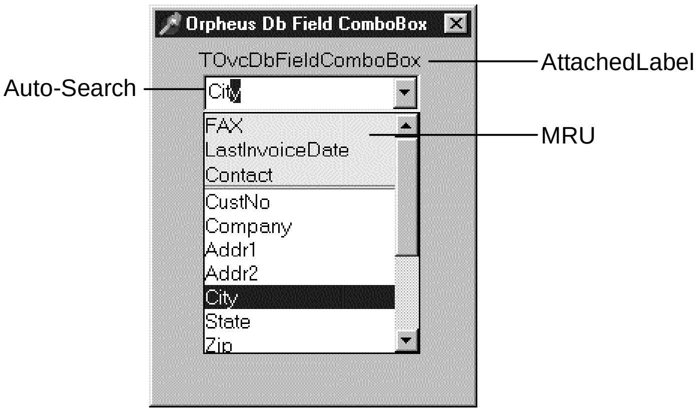


Figure 17.6: Orpheus DB Field ComboBox

Hierarchy

TCustomComboBox (VCL)

❶ TOvcBaseComboBox (OvcCmBx). 534

TOvcDbFieldComboBox (OvcDbFCb)

17

Properties

- ❶ About
- ❶ AttachedLabel
- ❶ AutoSearch
- ❶ Controller
- DataSource
- ❶ DroppedWidth
- FieldName
- ❶ HotTrack
- ❶ ItemHeight
- ❶ KeyDelay
- ❶ LabelInfo
- ❶ List
- ❶ ListIndex
- ❶ MRUListColor
- ❶ MRUListCount
- OmitFields
- ShowHiddenFields
- ❶ Style

Methods

- ❶ AddItem
- ❶ AssignItems
- ❶ ClearItems
- ❶ ClearMRUList
- ❶ InsertItem
- ❶ RemoveItem
- ❶ SelectionChanged

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❶ OnMouseWheel
- ❶ OnSelectionChange

Reference Section

DataSource property

```
property DataSource : TDataSource
```

↳ Specifies the data source component where data field names are obtained.

FieldName property

```
property FieldName : string
```

↳ Contains the name of the data field currently selected.

FieldName can be used to specify which field of the active record is of interest. For example, you could select a field and display its contents within the active record with the following OnSelectionChanged event handler:

```
  procedure TForm1.OvcDbFieldCBxChange(Sender : TObject);
  begin
    with OvcDbFieldComboBox1 do
      ShowMessage(Table1.FieldByName(FieldName).AsString);
  end;
```

OmitFields property

```
property OmitFields : TOvcFieldTypeSet
```

```
TOvcFieldTypeSet = set of TFieldType;
```

Default: Emtpy set

↳ Specifies the data field types to omit from the list.

Use OmitFields to filter the drop-down list.

The following example eliminates database fields of type ftCurrency from the drop-down list:

```
    with OvcDbFieldComboBox1 do
      OmitFields := OmitFields + [ftCurrency];
```

```
property ShowHiddenFields : Boolean
```

Default: False

↳ Specifies whether hidden fields are included in the drop-down list.

If ShowHiddenFields is True, TFields that have their Visible property set to False are still displayed. Otherwise, TFields that are not visible are not shown.

TOvcDbTableNameComboBox Component

The TOvcDbTableNameComboBox (shown in Figure 17.7) is a specialized data-aware combo box that displays a list of the tables for a specified database. The TOvcDbTableNameComboBox can be synchronized with a TOvcDbAliasComboBox (see page 555) so that the list of table names is updated when the alias selection changes.

The TOvcDbTableNameComboBox also includes auto-search facility, most-recently-used items, and attached label.

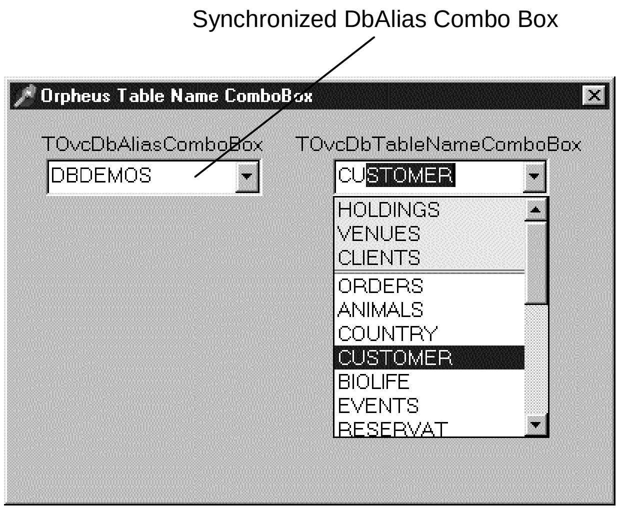


Figure 17.7: Orpheus Table Name Combo Box

Hierarchy

TCustomComboBox (VCL)

➊ TOvcBaseComboBox (OvcCmBx) 534

TOvcDbTableNameComboBox (OvcDbACb)

Properties

➊ About	➊ DroppedWidth	➊ ListIndex
AliasName	➊ HotTrack	➊ MRUListColor
➊ AttachedLabel	➊ ItemHeight	➊ MRUListCount
➊ AutoSearch	➊ KeyDelay	➊ Style
➊ Controller	➊ LabelInfo	TableName
DbEngineHelper	➊ List	

Methods

➊ AddItem	➊ ClearMRUList	➊ SelectionChanged
➊ AssignItems	➊ InsertItem	
➊ ClearItems	➊ RemoveItem	

Events

➊ AfterEnter	➊ OnMouseWheel
➊ AfterExit	➊ OnSelectionChange

Reference Section

AliasName — property

```
property AliasName : string
```

Specifies the database.

Use AliasName to select a BDE database to list. If the DbTable Name Combo Box is synchronized with a DbAlias Combo Box then AliasName will be updated with the alias selected by DbAlias Combo Box.

DbEngineHelper — property

```
property DbEngineHelper : TOvcDbEngineHelperBase
```

Identifies a component that acts as source for information specific to different database systems.

This component uses the component assigned to DbEngineHelper to assist in obtaining information from the database system in a generic way. Its primary purpose is to allow this and other data-aware Orpheus components to work with third-party database systems.

If you are using the BDE, place a TOvcDbBDEHelper component on the form and assign its name to this property. Additional “Helper” components designed to support other database systems are available. See the DBHELPER.HLP in the main Orpheus directory for additional information.

A value must be assigned to this property.

TableName — read-only property

```
property TableName : string
```

Contains name of the selected BDE database table.

TOvcDirectoryComboBox Component

The TOvcDirectoryComboBox (shown in Figure 17.8) is a specialized combo box that is aware of the directory structure of the current drive. The drop-down directory list is displayed in a hierarchical tree starting with the root directory of the current drive and proceeding down through the current selected directory path to its subdirectories. The selected directory name is accessed via the Directory property. The directory list can be optionally filtered by setting the Mask property. The FileComboBox property allows you to synchronize the Directory Combo Box with a TOvcFileComboBox.

The TOvcDirectoryComboBox also includes auto-search facility, most-recently-used items, and attached label.

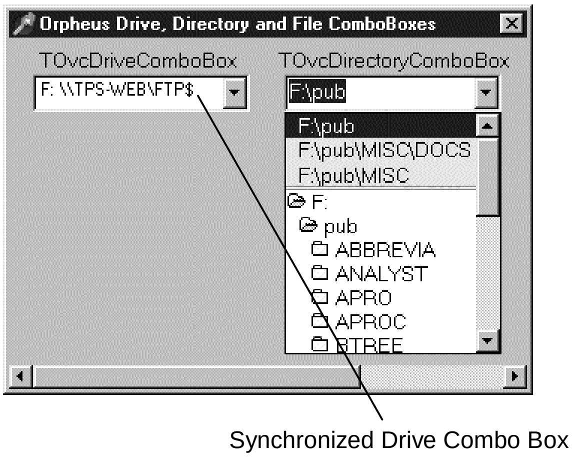


Figure 17.8: Orpheus Drive, Directory, and File Combo Box

17

Hierarchy

TCustomComboBox (VCL)

❶ TOvcBaseComboBox (OvcCmBx). 534

TOvcDirectoryComboBox (OvcDRCbx)

Properties

❶	About	❶	DroppedWidth	❶	List
❶	AttachedLabel		FileComboBox	❶	ListIndex
❶	AutoSearch	❶	HotTrack		Mask
❶	Controller	❶	ItemHeight	❶	MRUListColor
	Directory	❶	KeyDelay	❶	MRUListCount
	Drive	❶	LabelInfo	❶	Style

Methods

❶	AddItem	❶	ClearMRUList	❶	SelectionChanged
❶	AssignItems	❶	InsertItem		
❶	ClearItems	❶	RemoveItem		

Events

❶	AfterEnter	❶	OnMouseWheel
❶	AfterExit	❶	OnSelectionChange

Reference Section

Directory property

```
property Directory : string
```

Default: Current directory

↳ Contains the path name of the active directory.

Selecting a directory from the drop-down list updates this property to reflect the active directory path. By setting Directory to a valid directory path you can specify the active directory and hence the hierarchical tree displayed in the drop-down list.

Drive property

```
property Drive : Char
```

Default: Current drive

↳ Contains the drive letter of the active directory.

Set the Drive property when you wish to change drives. If the TOvcDirectoryComboBox is synchronized with a TOvcDriveComboBox via it's DirectoryComboBox property, Drive will be automatically set whenever a new drive is selected. See page 569 for more information on the TOvcDriveComboBox.

FileComboBox property

```
property FileComboBox : TOvcFileComboBox
```

↳ Specifies a synchronized File Combo Box.

The FileComboBox property provides the mechanism to associate the Directory Combo Box with an Orpheus File Combo Box (see page 572). If a File Combo Box is specified then when the directory selection changes, the Directory property of the File Combo Box will be updated as well so it can display the files in the selected directory.

Mask property

```
property Mask : string
```

Default: '*.*'

↳ Specifies which directories are included in the drop-down list.

Mask is a directory name that can include the wild card characters '*' and '?'. Only directories that match the mask are displayed in the drop-down list.

TOvcDriveComboBox Component

The TOvcDriveComboBox (see Figure 17.9) is a specialized combo box that is aware of the available disk drives. The drop-down list displays the drive letter and volume name for each available drive. The selected drive letter is accessed via the Drive property. The DirectoryComboBox property allows you to synchronize the Drive Combo Box with a TOvcDirectoryComboBox so that the Directory Combo Box will automatically update its directory list when the drive selection changes.

The TOvcDriveComboBox also includes most-recently-used items, and attached label.

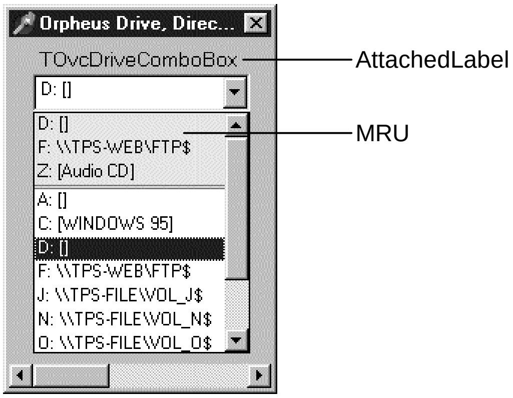


Figure 17.9: Drive Combo Box

Hierarchy

TCustomComboBox (VCL)

❶ TOvcBaseComboBox (OvcCmBx). 534

TOvcDriveComboBox (OvcDVCbx)

17

Properties

- ❶ About
- ❶ AttachedLabel
- ❶ AutoSearch
- ❶ Controller
- DirectoryComboBox
- Drive
- ❶ DroppedWidth
- FileComboBox
- ❶ HotTrack
- ❶ ItemHeight
- ❶ KeyDelay
- ❶ LabelInfo
- ❶ List
- ❶ ListIndex
- ❶ MRUListColor
- ❶ MRUListCount
- ❶ Style
- VolumeName

Methods

- ❶ AddItem
- ❶ AssignItems
- ❶ ClearItems
- ❶ ClearMRUList
- ❶ InsertItem
- ❶ RemoveItem
- ❶ SelectionChanged

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❶ OnMouseWheel
- ❶ OnSelectionChange

Reference Section

DirectoryComboBox property

```
property DirectoryComboBox : TOvcDirectoryComboBox
```

↳ Specifies a synchronized Directory Combo Box.

The DirectoryComboBox property provides the mechanism to associate the Drive Combo Box with an Orpheus Directory Combo Box (see page 566). If a Directory Combo Box is specified then when the drive selection changes, the Drive property of the Directory Combo Box will be updated as well so it can display directories on the selected drive.

Drive run-time property

```
property Drive : Char
```

↳ Contains the drive letter of the selected drive.

The selected drive can be changed by setting this property. Selecting a drive from the drop-down list updates this property.

VolumeName read-only property

```
property VolumeName : string
```

↳ Contains the volume name of the selected drive.

TOvcFileComboBox Component

The TOvcFileComboBox (as shown in Figure 17.10) is a specialized combo box that presents the user with a filtered list of files in a specified directory. Each item in the list can also include the icon associated with that file type. The TOvcFileComboBox can be synchronized with a TOvcDirectoryComboBox so that the File Combo Box will automatically update its file list when the directory selection changes. (See “TOvcDirectoryComboBox Component” on page 566)

The TOvcFileComboBox also includes auto-search facility, most-recently-used items, and attached label.

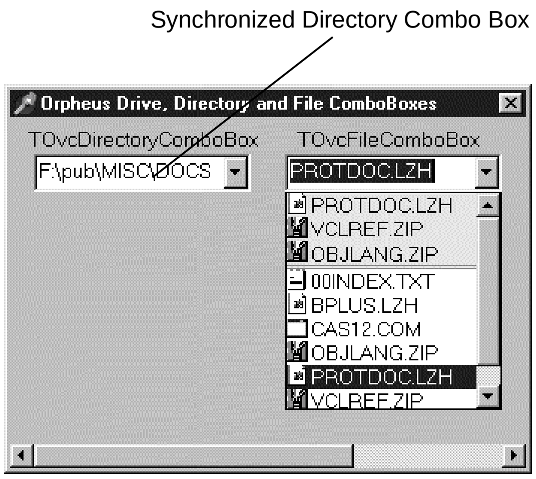


Figure 17.10: File Combo Box

Hierarchy

TCustomComboBox (VCL)

❶ TOvcBaseComboBox (OvcCmBx) 534

TOvcFileComboBox (OvcFLCbx)

Properties

❶	About	❶	DroppedWidth	❶	List
❶	AttachedLabel		FileMask	❶	ListIndex
	Attributes	❶	HotTrack	❶	MRUListColor
❶	AutoSearch	❶	ItemHeight	❶	MRUListCount
❶	Controller	❶	KeyDelay		ShowIcons
	Directory	❶	LabelInfo	❶	Style

Methods

❶	AddItem	❶	ClearMRUList	❶	SelectionChanged
❶	AssignItems	❶	InsertItem		
❶	ClearItems	❶	RemoveItem		

Events

❶	AfterEnter	❶	OnMouseWheel
❶	AfterExit	❶	OnSelectionChange

Reference Section

Attributes — property

```
property Attributes : TOvcCbxFileAttributes
```

```
TOvcCbxFileAttribute = (
  cbxReadOnly, cbxHidden, cbxSysFile, cbxArchive, cbxAnyFile);
```

```
TOvcCbxFileAttributes = set of TOvcCbxFileAttribute;
```

Default: [cbxAnyFile]

Specifies the type of files to be included in the drop-down list.

Use Attributes to filter the drop-down list according to file attributes.

The following example would include only hidden files in the drop-down list:

```
    OvcFileComboBox1.Attributes := [cbxHidden];
```

Directory — property

```
property Directory : string
```

Default: ‘.’

The directory pathname that contains all the files listed in the drop-down list.

If the TOvcFileComboBox is synchronized with a TOvcDirectoryComboBox, this property will be automatically set whenever a new directory is selected. See page 566 for more information on the Orpheus Directory Combo Box.

FileMask — property

```
property FileMask : string
```

Default: ‘*.*’

Specifies which files are included in the drop-down list.

FileMask is a file name that can include the wild card characters ‘*’ and ‘?’. Only files that match the mask are displayed in the drop-down list.

ShowIcons property

```
property ShowIcons : Boolean
```

Default: True

Specifies whether icons appear next to the file names in the drop-down list.

When ShowIcons is set to True, each file name in the drop-down list is displayed with the icon that is associated with the file extension.

TOvcFontComboBox Component

The TOvcFontComboBox (as shown in Figure 17.11) is a specialized combo box that presents the user with a list of installed fonts. By default, the font names in the drop-down list are displayed with their own font for quick preview. Each item in the list includes an icon that identifies the corresponding font family. The drop-down list can be filtered according to font family and pitch style. The PreviewControl property allows you to synchronize an separate preview edit control.

The TOvcFontComboBox also includes auto-search facility, most-recently-used items, and an attached label.

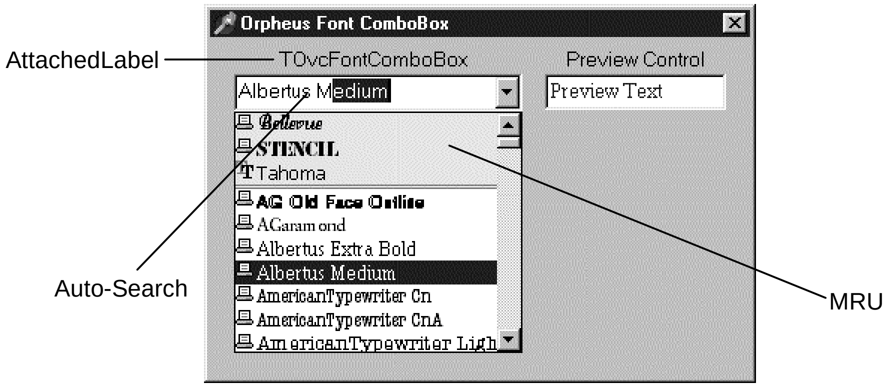


Figure 17.11: Font Combo Box

Hierarchy

TCustomComboBox (VCL)

❶ TOvcBaseComboBox (OvcCmBx). 534

TOvcFontComboBox (OvcFTCbx)

Properties

- ❶ About
- ❶ AttachedLabel
- ❶ AutoSearch
- ❶ Controller
- ❶ DroppedWidth
- FontCategories
- FontName
- FontPitchStyles
- ❶ HotTrack
- ❶ ItemHeight
- ❶ KeyDelay
- ❶ LabelInfo
- ❶ List
- ❶ ListIndex
- ❶ MRUListColor
- ❶ MRUListCount
- PreviewControl
- PreviewFont
- ❶ Style

Methods

- ❶ AddItem
- ❶ AssignItems
- ❶ ClearItems
- ❶ ClearMRUList
- ❶ InsertItem
- ❶ RemoveItem
- ❶ SelectionChanged

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❶ OnMouseWheel
- ❶ OnSelectionChange

Reference Section

FontCategories — property

```
property FontCategories : TFontCategories
```

```
TFontCategories = (fcAll, fcDevice, fcTrueType);
```

Default: fcAll

Determines which font families are included in the drop-down list.

The possible values for FontCategories are as follows:

Value	Description
fcAll	Device and TrueType fonts are included.
fcDevice	Only Device fonts are included.
fcTrueType	Only TrueType fonts are included.

See also: FontPitchStyles

FontName — run-time property

```
property FontName : string
```

Contains the name of the currently selected font.

FontName can be used to read the name of the selected font. If no font is selected then FontName is null. The font selection can be changed by setting FontName to any installed font name.

FontPitchStyles property

```
property FontPitchStyles : TFontPitches
```

```
TFontPitches = (fpAll, fpFixed, fpVariable);
```

Default: fpAll

Determines which pitch styles are included in the drop-down list.

The possible values for FontPitchStyles are as follows:

Value	**Description**
fpAll	Fixed pitch and variable pitch fonts are included.
fpFixed	Only fixed pitch fonts are included.
fpVariable	Only variable pitch fonts are included.

See also: FontCategories

PreviewControl property

```
property PreviewControl : TControl
```

Allows you to specify a component to preview the selected font.

The font of the component specified in PreviewControl is set to the SelectedFont and is automatically updated when SelectedFont changes.

See also: SelectedFont

PreviewFont property

```
property PreviewFont : Boolean
```

Default: True

Determines whether the font names in the drop-down list are displayed using their own font.

PreviewFont does not affect the PreviewControl, only the drop-down list. If PreviewFont is False then the list is displayed with the font specified in the Font property.

See also: PreviewControl

TOvcHistoryComboBox Component

The TOvcHistoryComboBox is a specialized combo box that can be used to implement a history selection list. It functions similarly to the MRU list portion of the TOvcComboBox component. If a new item is entered in the edit control, it is inserted at the top of the drop-down list and the last item is removed if it exceeds a specified limit. Unlike the MRU list, however, if the item already exists in the list, it is not brought to the top. The number of history items maintained by the combo box can be changed by setting the MaxHistory property.

The TOvcHistoryComboBox also includes the auto-search facility, and attached label.

Hierarchy

TCustomComboBox (VCL)

    ❶ TOvcBaseComboBox (OvcCmBx). 534

        TOvcHistoryComboBox (OvcHsCBx)

Properties

❶ About
❶ AttachedLabel
❶ AutoSearch
❶ Controller
❶ DroppedWidth
❶ HotTrack
❶ ItemHeight
❶ KeyDelay
❶ LabelInfo
❶ List
❶ ListIndex
MaxHistory
❶ MRUListColor
❶ MRUListCount
❶ Style

Methods

❶ AddItem
❶ AssignItems
❶ ClearItems
❶ ClearMRUList
❶ InsertItem
❶ RemoveItem
❶ SelectionChanged

Events

❶ AfterEnter
❶ AfterExit
❶ OnMouseWheel
❶ OnSelectionChange

Reference Section

MaxHistory property

```
property MaxHistory : Integer
```

Default: 3

↳ Determines the number of items that are kept in the list.

17

TOvcPrinterComboBox Component

The TOvcPrinterComboBox (as shown in Figure 17.12) is a specialized combo box that presents the user with a list of printers installed in Windows and optionally allows the user to specify which printer is selected for printing. This component uses the VCL's Printer object to obtain the list of installed printers and to change printer selections. See TPrinter in the VCL documentation for more information.

The TOvcPrinterComboBox also includes auto-search facility, most-recently-used items, and attached label.

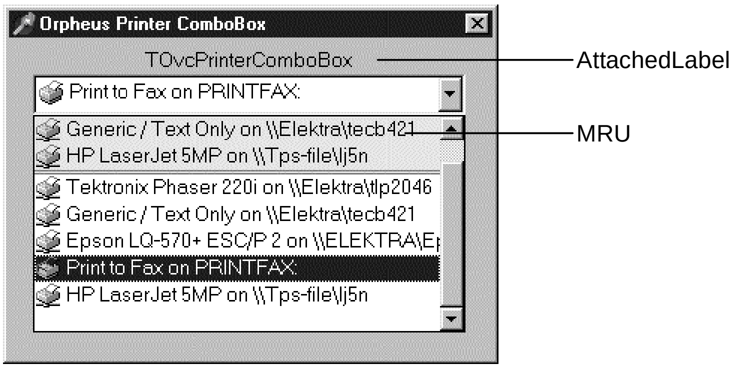

Orpheus Printer ComboBox
TOvcPrinterComboBox
AttachedLabel
Print to Fax on PRINTFAX:
Generic / Text Only on \\Elektra\tecb421
MRU
HP LaserJet 5MP on \\Tps-file\lj5n
Tektronix Phaser 220i on \\Elektra\tlp2046
Generic / Text Only on \\Elektra\tecb421
Epson LQ-570+ ESC/P 2 on \\ELEKTRA\Ep
Print to Fax on PRINTFAX:
HP LaserJet 5MP on \\Tps-file\lj5n

Figure 17.12: Printer Combo Box

Hierarchy

TCustomComboBox (VCL)

❶ TOvcBaseComboBox (OvcCmBx) 534

TOvcPrinterComboBox (OvcPrCbx)

Properties

- ❶ About
- ❶ AttachedLabel
- ❶ AutoSearch
- ❶ Controller
- ❶ DroppedWidth
- ❶ HotTrack
- ❶ ItemHeight
- ❶ KeyDelay
- ❶ LabelInfo
- ❶ List
- ❶ ListIndex
- ❶ MRUListColor
- ❶ MRUListCount
- SelectPrinter
- ShowIcons
- ❶ Style

Methods

- ❶ AddItem
- ❶ AssignItems
- ❶ ClearItems
- ❶ ClearMRUList
- ❶ InsertItem
- ❶ RemoveItem
- ❶ SelectionChanged

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❶ OnMouseWheel
- ❶ OnSelectionChange

Reference Section

SelectPrinter — property

```
property SelectPrinter : Boolean
```

Default: True

Specifies whether a selection will update Printer.PrinterIndex and make that printer the active printer.

If SelectPrinter is True, then the printer displayed in the edit field will be selected for printing. If False, then the combo box merely displays the list of installed printers and the selected printer name can be accessed via the Text property.

ShowIcons — property

```
property ShowIcons : Boolean
```

Default: False

Specifies whether icons appear next to the printer names in the drop-down list.

If ShowIcons is True, then an icon will be displayed in the drop-down list indicating whether the printer is local or a network printer.

Chapter 18: Dialog Components

Dialog components are non-visual components (like the standard file open and save dialogs). Orpheus provides seven dialog components: TOvcCalendarDialog, TOvcCalculatorDialog, TOvcTimeDialog, TOvcMemoDialog, TOvcDBMemoDialog, TOvcAliasDialog, and TOvcSplashDialog. The Orpheus Dialog components provide an easy mechanism for working with complex components in a fast and easy way. The general look and feel for each of the Orpheus dialog components is very similar. This allows you to present a consistent user interface within your applications.

Each of the Orpheus dialog components descends from a common class: TOvcBaseDialog. TOvcBaseDialog implements the general look of the form and defines some basic options such as form placement and style. Each of the Orpheus dialog components has an Execute method and a form with options for the visual component that they contain. See the appropriate section in Orpheus or the VCL reference for details specific to the contained component. In most cases, the dialog component has a public property that allows access (at run time) to the underlying visual component.

The ExDlg example program demonstrates the components described in this section.

TOvcBaseDialog Class

TOvcBaseDialog serves as the ancestor class for the other Orpheus dialog components. It introduces the properties and methods common to all of the Orpheus dialog components.

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase). 21

TOvcBaseDialog (OvcDlg)

Properties

❶	About	Font	Placement
	Caption	Options	

Methods

Execute

Events

OnHelpClick

Reference Section

Caption property

```
property Caption : string
```

Determines the title of the displayed form.

Execute virtual method

```
function Execute : Boolean; virtual;
```

Creates and displays the dialog form. The function returns True if the dialog is closed using the OK button. Otherwise, False is returned.

Font property

```
property Font : TFont
```

Determines the font used by the form and the components on the form.

OnHelpClick event

```
property OnHelpClick : TNotifyEvent
```

Defines an event handler that is called when the help button is clicked.

See also: Options

Options property

```
property Options : TOvcDialogOptions

TOvcDialogOption = (doShowHelp, doSizeable);

TOvcDialogOptions = set of TOvcDialogOption;
```

Determines some of the form’s properties.

doShowHelp determines if a help button is displayed and doSizeable determines if the form can be resized. If the Help button is visible, the method assigned to the OnHelp event handler is fired when the help button is pressed.

See also: OnHelpClick

Placement property

```
property Placement : TOvcDialogPlacement

TOvcDialogPlacement = class(TPersistent)
published
  property Position : TOvcDialogPosition
    read FPosition write FPosition;
  property Top : Integer read FTop write FTop;
  property Left : Integer read FLeft write FLeft;
  property Height : Integer read FHeight write FHeight;
  property Width : Integer read FWidth write FWidth;
end;

TOvcDialogPosition = (mpCenter, mpCenterTop, mpCustom);
```

Determines where the dialog is positioned.

Placement defines a class that contains placement properties. The Position property can have one of three values: mpCenter causes the form to be centered vertically and horizontally within the screen; mpCenterTop is similar to mpCenter, except that the vertical position is centered in the top third of the screen; mpCustom causes the dialog to use the Top, and Left properties. The Height, and Width properties operate independently of Position.

The TOvcDialogPlacement class contains the actual properties that control the size and position of the displayed form and is used internally to group these properties into one area of the object inspector. The Top, Left, Height, and Width properties are the same as the liked-named TForm properties.

TOvcCalculatorDialog Component

TOvcCalculatorDialog (as shown in Figure 18.1) provides an easy method for displaying an Orpheus calculator component (see page 499). In addition to the calculator component, the form displays OK, Cancel, and, optionally, Help buttons.

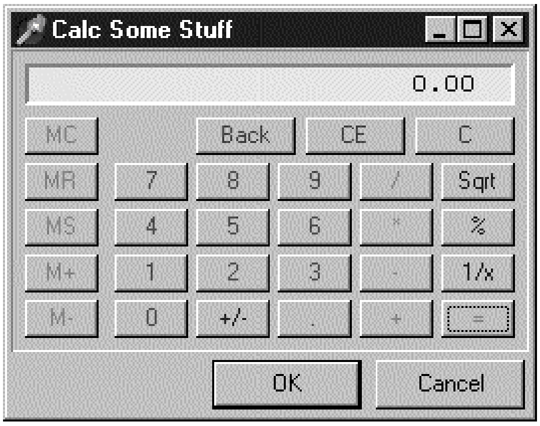


Figure 18.1: Calculator dialog box

The following example displays the calculator dialog and shows the results in a label:

```
{show calculator dialog}
if OvcCalculatorDialog1.Execute then
  Label1.Caption :=
    FloatToStr(OvcCalculatorDialog1.Calculator.DisplayValue);
```

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

❷ TOvcBaseDialog (OvcDlg) 588

TOvcCalculatorDialog (OvcClcDg)

Properties

❶ About	❷ Caption	❷ Options
Calculator	❷ Font	❷ Placement

Methods

❷ Execute

Events

❷ OnHelpClick

Reference Section

Calculator property

```
property Calculator : TOvcCalculator
```

Provides access to the dialog’s TOvcCalculator component.

All of the properties and methods of the TOvcCalculator are available through the Calculator property.

See the TOvcCalculator component on page 499.

TOvcCalendarDialog Component

TOvcCalendarDialog (as shown in Figure 18.2)provides an easy method for displaying an Orpheus calendar component (see page 481). In addition to the calendar component, the form displays OK, Cancel, and, optionally, Help buttons.

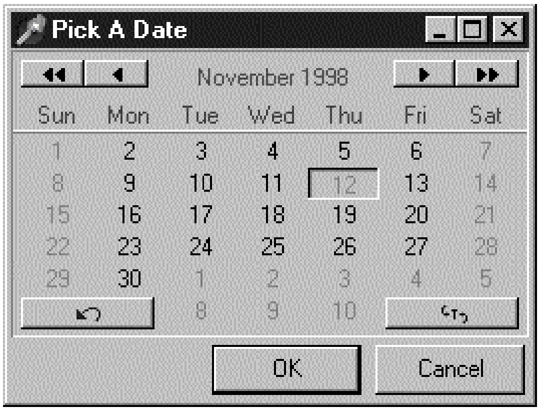


Figure 18.2: Calendar dialog box

The following example invokes the calendar dialog with the calendar initialized to today's date and then retrieves the date selected in the dialog:

```
{initialize calendar date}
OvcCalendarDialog1.Calendar.Date := Now;
{show dialog and retrieve date if OK pressed}
if OvcCalendarDialog1.Execute then
  SelectedDate := OvcCalendarDialog1.Calendar.Date;
```

Hierarchy

TComponent (VCL)

- ❶ TOvcComponent (OvcBase) 21
  - ❷ TOvcBaseDialog (OvcDlg) 588
    - TOvcCalendarDialog (OvcCalDg)

Properties

❶ About	❷ Caption	❷ Options
Calendar	❷ Font	❷ Placement

Methods

❷ Execute

Events

❷ OnHelpClick

Reference Section

Calendar property

```
property Calendar : TOvcCalendar
```

Provides access to the dialog's TOvcCalendar component.

All of the properties and methods of the TOvcCalendar (see page 481) are available through the Calendar property.

TOvcClockDialog Component

TOvcClockDialog (as shown in Figure 18.3)provides an easy method for displaying an Orpheus clock component (see page 515). In addition to the clock component, the form displays OK, and, optionally, Help buttons.

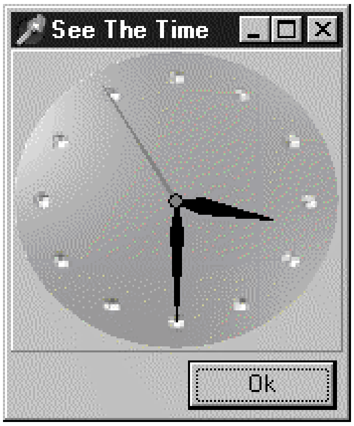


Figure 18.3: Clock dialog box

The following examples of display the clock dialog:

```
{show clock dialog}
OvcClockDialog1.Execute;
...
{show clock dialog for a time zone two hours different}
OvcClockDialog1.Clock.TimeOffset := -2;
```

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

❷ TOvcBaseDialog (OvcDlg) 588

TOvcClockDialog (OvcClkDg)

Properties

❶ About	❷ Caption	❷ Options
Clock	❷ Font	❷ Placement

Methods

❷ Execute

Events

❷ OnHelpClick

Reference Section

Clock property

```
property Clock : TOvcClock
```

Clock provides access to the dialog’s TOvcClock component.

All of the properties and methods of the TOvcClock are available through the Clock property.

See “TOvcClock Component” on page 515.

TOvcMemoDialog Component

TOvcMemoDialog (as shown in Figure 18.4) provides and easy method for displaying a memo component. In addition to the memo component, the form displays OK, Cancel, and, optionally, Help buttons.

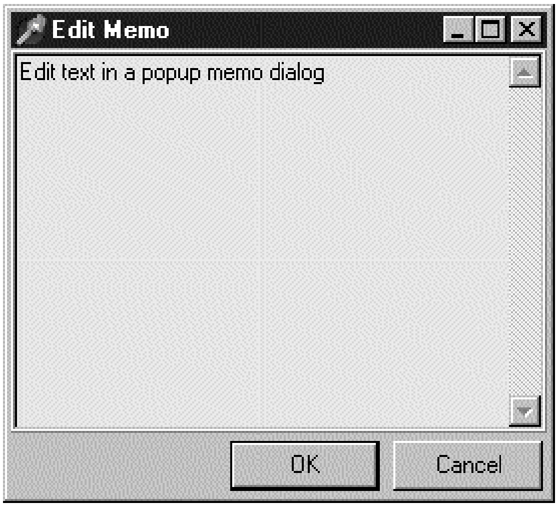


Figure 18.4: Edit memo dialog box

The following example displays the memo dialog and transfers the contents to and from a memo component on the same form:

```
OvcMemoDialog1.Lines.Assign(Memo1.Lines);
if OvcMemoDialog1.Execute then begin
  {transfer memo data}
  Memo1.Lines.Assign(OvcMemoDialog1.Lines);
end;
```

Hierarchy

TComponent (VCL)

- ❶ TOvcComponent (OvcBase) 21
  - ❷ TOvcBaseDialog (OvcDlg) 588
    - TOvcMemoDialog (OvcMoDg)

Properties

❶ About	Lines	❷ Placement
❷ Caption	MemoFont	ReadOnly
❷ Font	❷ Options	WordWrap

Methods

❷ Execute

Events

❷ OnHelpClick

Reference Section

Lines property

```
property Lines : TStrings
```

↳ Represents the lines in the memo that is displayed in the dialog.

Set Lines to initialize the memo and read Lines to retrieve the contents of the edited memo.

MemoFont property

```
property MemoFont : TFont
```

↳ The font used by the memo component.

ReadOnly property

```
property ReadOnly : Boolean
```

Default: False

↳ Determines whether the user can change the contents of the memo.

If ReadOnly is True, the user can't change the contents. If ReadOnly is False, the user can modify the contents.

WordWrap property

```
property WordWrap : Boolean
```

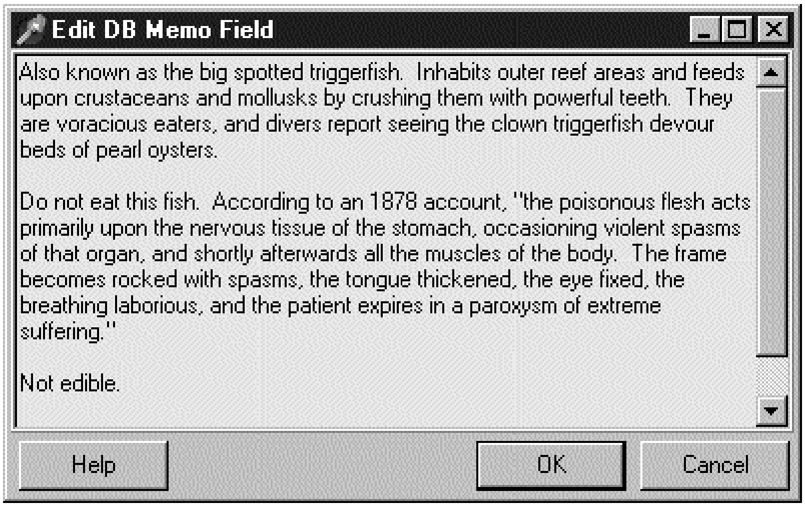
Default: True

↳ Determines if lines wider than the display area of the memo will be wrapped to the next line.

Set WordWrap to True to make the memo control wrap text at the right margin so it fits in the client area. Set WordWrap to False to have the memo control show a separate line only where return characters were explicitly entered into the text.

TOvcDbMemoDialog Component

TOvcDbMemoDialog (as shown in Figure 18.5) provides a means to display and edit the contents of memo fields from a dataset. The displayed dialog contains a memo component for viewing and editing the text of the dataset memo, along with OK, Cancel, and, optionally, Help buttons.



Also known as the big spotted triggerfish. Inhabits outer reef areas and feeds upon crustaceans and mollusks by crushing them with powerful teeth. They are voracious eaters, and divers report seeing the clown triggerfish devour beds of pearl oysters.

Do not eat this fish. According to an 1878 account, "the poisonous flesh acts primarily upon the nervous tissue of the stomach, occasioning violent spasms of that organ, and shortly afterwards all the muscles of the body. The frame becomes rocked with spasms, the tongue thickened, the eye fixed, the breathing laborious, and the patient expires in a paroxysm of extreme suffering."

Not edible.

Figure 18.5: Db memo dialog box

The following example of displays a data-aware memo dialog:

```
if OvcDbMemoDialog1.Execute then begin
  {take some action}
end;
```

Hierarchy

TComponent (VCL)

- ❶ TOvcComponent (OvcBase) 21
  - ❷ TOvcBaseDialog (OvcDlg) 588
    - TOvcDbMemoDialog (OvcDbMDg)

Properties

❶ About	❷ Font	ReadOnly
❷ Caption	MemoFont	WordWrap
DataField	❷ Options	
DataSource	❷ Placement	

Methods

❷ Execute

Events

❷ OnHelpClick

Reference Section

DataField property

```
property DataField : string
```

↳ Links the memo control to a field in the dataset.

The dataset which contains the field must be specified by setting the DataSource property.

See also: DataSource

DataSource property

```
property DataSource : TDataSource
```

↳ Links the memo control with a dataset.

Link the memo control to a dataset by setting DataSource to a data source component that identifies the desired dataset. Specify the field in the dataset that contains the memo data using the DataField property.

See also: DataField

MemoFont property

```
property MemoFont : TFont
```

↳ MemoFont is the font used by the memo component.

ReadOnly property

```
property ReadOnly : Boolean
```

Default: False

↳ Determines whether the user can use the memo to change the value of the field of the current record.

If ReadOnly is False, the user can change the field’s value as long as the dataset is in edit mode. If ReadOnly is True, the user can’t change the contents.

WordWrap property

```
property WordWrap : Boolean
```

Default: True

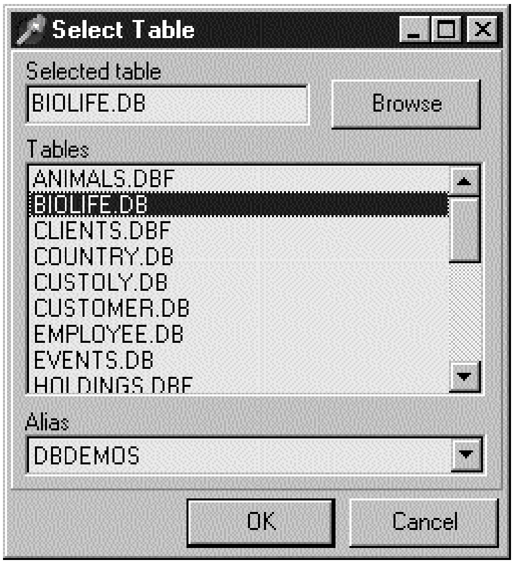
↳ WordWrap determines if lines wider than the display area of the memo will be wrapped to the next line.

Set WordWrap to True to make the memo control wrap text at the right margin so it fits in the client area. Set WordWrap to False to have the memo control show a separate line only where return characters were explicitly entered into the text.

TOvcDbAliasDialog Component

TOvcDbAliasDialog (as shown in Figure 18.6) provides a means to display a list of available aliases and corresponding database tables. The list of aliases is obtained from the current session and its Path property is expanded to locate and display the database tables.

A table name can be selected from the list of available tables, entered directly into the table name edit field, or browsed to and then selected from the file open dialog. The OK button remains disabled until a valid table name is selected.



Selected table: BIOLIFE.DB

Tables:
ANIMALS.DBF
BIOLIFE.DB
CLIENTS.DBF
COUNTRY.DB
CUSTOLY.DB
CUSTOMER.DB
EMPLOYEE.DB
EVENTS.DB
HOLDINGS.DBF

Alias: DBDEMOS

Figure 18.6: Alias dialog box

The following example displays the alias dialog:

```
if OvcDbAliasDialog1.Execute then begin
  {take some action using the alias name and/or the selected
    table}
end;
```

Hierarchy

TComponent (VCL)

- ❶ TOvcComponent (OvcBase)
  - ❷ TOvcBaseDialog (OvcDlg)
    - TOvcDbAliasDialog (OvcDbADg)

Properties

❶	About	❷	Font	❷	Placement
	AliasName	❷	Options		TableName
❷	Caption		Path		

Methods

❷ Execute

Events

❷ OnHelpClick

Reference Section

AliasName — property

```
property AliasName : string
```

Represents the name of an alias that was selected.

AliasName can be assigned a value prior to displaying the dialog to populate the list of tables. Otherwise, no alias or table is selected initially.

Path — read-only property

```
property Path : string
```

Returns the path associated with the selected table.

If an alias was selected, Path returns the path associated with that alias, otherwise Path returns the path of the selected database table.

See also: TableName

TableName — read-only property

```
property TableName : string
```

The name of the selected database table.

The name returned includes the extension, but not the path. Use the Path property to access the path of the selected table.

See also: Path

TOvcSplashDialog Component

TOvcSplashDialog provides a means to display an image while a form is initializing. The splash dialog allows you to specify two images, (one for low resolution systems and one for systems that can handle higher resolutions), a delay value that determines how long the image is visible, and where to display the image on the screen.

The splash screen image can be displayed indefinitely (until you call a routine that hides it) or until a specific number of milliseconds have elapsed. You can leave the image visible longer, by delaying the call to hide it or by increasing the Delay property value.

At run time, the splash screen image is automatically displayed as the form is created. You can bypass the automatic system by setting the Active property to False and using the ShowSplashScreen and HideSplashScreen methods.

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

❷ TOvcBaseDialog (OvcDlg) 588

TOvcSplashDialog (OvcSplDg)

Properties

❶ About	❷ Font	PictureLoRes
Active	Icon	❷ Placement
❷ Caption	❷ Options	StayOnTop
Delay	PictureHiRes	Visible

Methods

Close	❷ Execute	Show

Events

OnClick	OnDblClick
OnClose	❷ OnHelpClick

Reference Section

Active property

```
property Active : Boolean
```

Default: True

Determines if the splash image is automatically displayed when the form is created.

If Active is False, use Show to display the splash image dialog as necessary.

Close method

```
procedure Close;
```

Hides and then removes the splash dialog from memory.

See also: Show

Delay property

```
property Delay : Integer
```

Default: 0 (indefinite)

Determines how long (in milliseconds) the splash image is displayed.

If Delay is zero, the image is displayed until the Close method is called.

See also: Close

OnClick property

```
property OnClick : TNotifyEvent
```

Defines an event handler that is fired when the user clicks on the splash image.

OnClose property

```
property OnClose : TNotifyEvent
```

Defines an event handler that is fired when the splash dialog is closed.

OnDblClick property

```
property OnDblClick : TNotifyEvent
```

Defines an event handler that is fired when the user double-clicks on the splash image.

PictureHiRes property

```
property PictureHiRes : TPicture
```

↳ Defines the picture that is displayed on systems that are using 256 colors or greater.

See also: PictureLoRes

PictureLoRes property

```
property PictureLoRes : TPicture
```

↳ Defines the picture that is displayed on systems that are using less than 256 colors.

If no image is assigned using this property, the image in PictureHiRes is used instead.

See also: PictureHiRes

Show method

```
procedure Show;
```

↳ Creates and displays the splash image dialog.

See also: Close

StayOnTop property

```
property StayOnTop : Boolean
```

Default: True

↳ Determines the state of the window used to display the splash image.

If StayOnTop is True, the window uses the "always on top" style. Otherwise, it is a "normal" window—one that can be covered by other windows.

Visible run-time property

```
property Visible : Boolean
```

↳ Determines if the splash image is currently visible.

Setting Visible to True displays the splash image by internally calling the Show method. Setting Visible to False hides the splash image by internally calling the Close method.

Chapter 19: Slider Components

Orpheus Sliders are visual controls used to select a single value from a valid range.

Orpheus includes a stand-alone Slider control as well as a numeric data entry field with an attached Slider. Both controls are provided in standard and data-aware versions.

TOvcCustomSlider Class

The TOvcCustomSlider class is the immediate ancestor of the TOvcSlider component. It implements all of the methods and properties used by the TOvcSlider component and is identical to the TOvcSlider component except that no properties are published.

TOvcCustomSlider is provided to facilitate creation of descendent components. For property and method descriptions, see “TOvcSlider Component” on page 615.

Hierarchy

TCustomControl (VCL)

    TOvcCustomControl (OvcBase) 16

        TOvcCustomSlider (OvcSlide)

TOvcSlider Component

TOvcSlider is a special data entry control that provides a means to change a value by moving a marker along a horizontal (or vertical) distance. Each segment of motion corresponds to a specific change in the associated value. Alternatively, the marker position can be set programmatically by assigning a value to the Position property.

Optionally, grid marks are drawn along the length of the control to provide a visual indication of the slider position. TOvcSlider automatically determines the correct number and spacing for the displayed grid marks.

The EXSLIDE example project demonstrates many of the slider capabilities.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase). 16

TOvcCustomSlider (OvcSlide) 614

TOvcSlider (OvcSlide)

Properties

❶	About		MarkColor	Position
❶	AttachedLabel		Max	Step
	DrawMarks		Min	
❶	LabelInfo		Orientation	

Events

❶	AfterEnter	❶	AfterExit	❶	OnMouseWheel

Reference Section

DrawMarks property

```
property DrawMarks : Boolean
```

Default: True

↳ Determines if the small indicator marks are drawn.

If DrawMarks is True, the grid marks are drawn using the color specified by the MarkColor property. The number of marks and the distance between marks is automatically determined by the slider component based on the Min, Max, and Step properties.

See also: MarkColor, Max, Min, Step

MarkColor property

```
property MarkColor : TColor
```

Default: clBtnText

↳ Determines the color used to draw the grid marks.

Max property

```
property Max : Double
```

Default: 10

↳ Determines the upper limit to the possible values that are controlled by the slider control.

Max must always be greater than Min or an exception is raised.

See also: Min

Min property

```
property Min : Double
```

Default: 0

↳ Determines the lower limit to the possible values that are controlled by the slider control.

Min must always be smaller than Max or an exception is raised.

See also: Max

Orientation property

```
property Orientation : TOvcSliderOrientation;
```

```
TOvcSliderOrientation = (soHorizontal, soVertical);
```

Default: soHorizontal

Determines if the slider control is displayed horizontally or vertically.

Position property

```
property Position : Double
```

Determines the value of the slider control.

Setting Position causes the slider to display the new value. Changing the location of the slider button causes the value of Position to change.

If values less than Min are assigned, the Min value is used. Likewise, assigning a value greater than Max cause the Max value to be used. No error is generated by these actions.

Step property

```
property Step : Double
```

Default: 1

Determines how much the slider value will change for each change in the slider button's position.

TOvcDbSlider Component

The TOvcDbSlider is identical to the TOvcSlider component with the additional ability of allowing it to be connected to and edit a field in a dataset.

Hierarchy

TCustomControl (VCL)

  ❶ TOvcCustomControl (OvcBase). 16

    TOvcCustomSlider (OvcSlide) 614

      TOvcDbSlider (OvcDbSld)

Properties

❶ About	DataField	Field
❶ AttachedLabel	DataSource	❶ LabelInfo

Events

❶ AfterEnter	❶ AfterExit	❶ OnMouseWheel

Reference Section

DataField property

```
property DataField : string
```

Identifies the field from which the component displays data.

See also: DataSource

DataSource property

```
property DataSource : TDataSource
```

Specifies the data source component where the component obtains the data to display.

See also: DataField

Field read-only property

```
property Field : TField
```

Returns the TField object to which the entry field component is linked.

Use the Field object to change the value of the data in the field programmatically.

TOvcCustomSliderEdit Class

The TOvcCustomSliderEdit class is the immediate ancestor of the TOvcSliderEdit component. It implements all of the methods and properties used by the TOvcSliderEdit component and is identical to TOvcSliderEdit except that no properties are published.

TOvcCustomSliderEdit is provided to facilitate creation of descendent slider edit components. For property and method descriptions, see “TOvcSliderEdit Component” on page 621.

Hierarchy

TCustomEdit (VCL)

  TOvcCustomEdit (OvcEditF) 1272

    TOvcEdpop-up (OvcEdPop). 1275

      TOvcCustomSliderEdit (OvcEdSld)

TOvcSliderEdit Component

TOvcSliderEdit provides an edit control for entry of numeric data. Additionally, a slider control (TOvcSlider) can be displayed by pressing a button positioned at the right edge of the control. The slider control allows an alternate data entry method, which may be more intuitive depending on the meaning of the data the control is editing. Several properties allow you to access the value displayed in the control in different ways—as an integer, as a floating-point value, or as a string.

Hierarchy

TCustomEdit (VCL)

TOvcCustomEdit (OvcEditF) 1272

❶ TOvcEdPopup (OvcEdPop) 1275

TOvcCustomSliderEdit (OvcEdSld) 614

TOvcSliderEdit (OvcEdSld)

Properties

AllowIncDec	❶ ButtonGlyph	PopupOpen
AsFloat	❶ PopupAnchor	PopupWidth
AsInteger	PopupClose	❶ ShowButton
AsString	PopupHeight	Slider

Reference Section

AllowIncDec — property

```
property AllowIncDec : Boolean
```

Default: False

Determines if pressing the <+> and <-> keys will increment and decrement the field value.

If AllowIncDec is True, pressing the <+> key will increase the displayed value by one and pressing the <-> Key will decrease the displayed value by one. If AllowIncDec is False, pressing the <+> or <-> keys is ignored.

AsInteger — run-time property

```
property AsInteger : LongInt
```

Determines the value of the field as a integer value.

AsFloat — run-time property

```
property AsFloat : Double
```

Determines the value of the field as a floating-point value.

AsString — run-time property

```
property AsString : string
```

Determines the value of the field as a text value.

The value assigned to AsString must be a valid numeric value. If not, an exception is raised when the text is converted into a number.

PopupClose — virtual method

```
procedure PopupClose(Sender : TObject); virtual;
```

Closes the pop-up slider control.

Calling PopupClose hides the pop-up slider control. If the slider is not visible, no action is taken.

See also: PopupOpen

PopupHeight property

```
property PopupHeight : Integer
```

Determines the height of the pop-up slider control.

PopupOpen virtual method

```
procedure PopupOpen; virtual;
```

Displays the pop-up slider control.

See also: Popup Close

PopupWidth property

```
property PopupWidth : Integer
```

Determines the width of the pop-up slider control.

Slider run-time, read-only property

```
property Slider : TOvcSlider
```

Provides access to the TOvcSlider component used internally.

Slider is provided mainly so that you can have access to properties of the slider component. Please take care that the settings you change do not affect the behavior expected by the edit portion of the control.

TOvcCustomDbSliderEdit Class

The TOvcCustomDbSliderEdit class is the immediate ancestor of the TOvcDbSliderEdit component. It implements all of the methods and properties used by the TOvcDbSliderEdit component and is identical to TOvcDbSliderEdit except that no properties are published.

TOvcCustomDbSliderEdit is provided to facilitate creation of descendent components. For property and method descriptions, see “TOvcDbSliderEdit Component” on page 625.

Hierarchy

```
TCustomEdit (VCL)
    TOvcCustomEdit (OvcEditF) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1272
        TOvcEdPopup (OvcEdPop) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .1275
            TOvcCustomSliderEdit (OvcEdSld) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .614
                TOvcCustomDbSliderEdit (OvcDbSld)
```

TOvcDbSliderEdit Component

The TOvcDbSliderEdit is identical to the TOvcSliderEdit component with the additional ability of allowing it to be connected to and edit a field in a dataset.

Hierarchy

TCustomEdit (VCL)

  TOvcCustomEdit (OvcEditF) 1272

    ❶ TOvcEdPopup (OvcEdPop) 1275

      TOvcCustomSliderEdit (OvcEdSld) 614

        TOvcCustomDbSliderEdit (OvcDbSld). 624

          TOvcDbSliderEdit (OvcDbSld)

Properties

❶ ButtonGlyph
DataField
DataSource
Field
❶ PopupAnchor
❶ PopupClose
❶ PopupOpen
❶ ShowButton

Reference Section

DataField property

```
property DataField : string
```

Identifies the field from which the component displays data.

See also: DataSource

DataSource property

```
property DataSource : TDataSource
```

Specifies the data source component where the component obtains the data to display.

See also: DataField

Field read-only property

```
property Field : TField
```

Returns the TField object to which the entry field component is linked.

Use the Field object to change the value of the data in the field programmatically.

Identifier Index

Subject Index

Subject Index

Subject Index

Orpheus 4™

User's Manual
Volume 2

TurboPower Software Company
Colorado Springs, CO

www.turbopower.com

First Edition September 1997
Second Edition May 1998
Third Edition December 1998
Fourth Edition June 2001

License Agreement

Table Of Contents

Volume 1

Volume 2

Chapter 20: Buttons

This section introduces several components that behave in a similar way to the standard button components.

- TOvcButtonHeader allows you to select sections of a header. This could be used, for example, above a list of related items to specify the sort parameters for the list.
- TOvcAttachedButton is a TBitButton that you can attach to other components.
- TOvcSpeedButton is an alternative to the standard TSpeedButton control that adds the capability to auto-repeat while the button is pressed.
- TO32FlexButton allows you to provide any number of alternate functionalities for a single button. The user may select from a pop-up list of available functionalities by clicking on a small area in the corner of the button.

TOvcButtonHeader Component

TOvcButtonHeader provides a component with a mixture of THeader and TSpeedButton properties. Each section of the header can be selected using the mouse or the hot key associated with the name of the section. Events are generated when the header is clicked, when sizing starts, and when sizing ends. This component could be used above a list of related items to determine which column the list should be sorted on.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase). 16

❷ TOvcCustomControlEx (OvcBase) 19

TOvcButtonHeader (OvcBtnHdr)

Properties

❶ About	DrawingStyle	SectionCount
AllowDragRearrange	Images	Sections
AllowResize	ItemIndex	Style
❶ AttachedLabel	❶ LabelInfo	TextMargin
BorderStyle	LeftImages	WordWrap
❷ Controller	Section	

Events

❶ AfterEnter	❶ OnMouseWheel	OnSized
❶ AfterExit	OnRearranging	OnSizing
OnClick	OnRearranged	

Reference Section

AllowDragRearrange — property

```
property AllowDragRearrange : Boolean
```

Default: False

↳ Determines whether the sections can be rearranged at run time by dragging them with the mouse.

If AllowRearrange is False, as it is by default, no run-time rearranging is allowed.

AllowResize — property

```
property AllowResize : Boolean
```

Default: True

↳ Determines whether the sections can be resized at run time using the mouse.

If AllowResize is False, no run-time sizing is allowed.

BorderStyle — property

```
property BorderStyle : TBorderStyle
```

Default: bsNone

↳ Determines the style used when drawing the border.

The possible values are bsSingle (a single line border is drawn around the component) or bsNone (no border line).

The TBorderStyle type is defined in the VCL's Controls unit.

DrawingStyle — property

```
DrawingStyle : TOvcBHDrawingStyle

TOvcBHDrawingStyle = (
  bhsDefault, bhsThin, bhsFlat, bhsEtched);
```

Default: bhsDefault

↳ Controls the visual appearance of the button header.

Four different looks are supported. The DrawingStyle property does not affect the behavior of the button header.

Images — property

```
property Images : TImageList
```

↳ Refers to an image list, which can contain images to be shown to the right of the caption in each button header section.

See also: LeftImages, TOvcButtonHeaderSection.ImageIndex, Caption

ItemIndex — property

```
property ItemIndex : Integer
```

↳ Determines which section is currently selected.

Valid values for ItemIndex are 0 through Sections.Count -1.

LeftImages — property

```
property LeftImages : TImageList
```

↳ Refers to an image list, which can contain images to be shown to the left of the caption in each button header section.

See also: Images, TOvcButtonHeaderSection.Caption,
TOvcButtonHeaderSection.LeftImageIndex

OnClick — event

```
property OnClick : TNotifyEvent
```

↳ Defines an event handler that is called when the mouse is clicked on a button header section.

TNotifyEvent is defined in the VCL's Classes unit.

OnRearranged — event

```
property OnRearranged : TOvcButtonHeaderRearrangedEvent
```

```
TOvcButtonHeaderRearrangedEvent = procedure(
  Sender: TObject; OldIndex, NewIndex : Integer) of object;
```

↳ Defines an event handler that is called when the user finishes moving a section.

The OnRearranged event is generated when the mouse button is released.

OnRearranging event

```
property OnRearranging : TOvcButtonHeaderRearrangingEvent
```

```
TOvcButtonHeaderRearrangingEvent = procedure(
  Sender: TObject; OldIndex, NewIndex : Integer;
  var Allow : Boolean) of object;
```

Defines an event handler that is called when the user is moving a section with the mouse.

OnSized event

```
property OnSized : TSectionEvent
```

Defines an event handler that is called when the user finishes sizing a section.

The OnSized event is generated when the mouse button is released.

OnSizing event

```
property OnSizing : TSectionEvent
```

Defines an event handler that is called when the user is sizing a section.

Section property

```
property Section[Index : Integer] : TOvcButtonHeaderSection
```

Returns a reference to the section at position Index.

SectionCount run-time, read-only property

```
property SectionCount : Integer
```

Returns the number of sections in the button header.

Sections property

```
property Sections : TOvcCollection
```

Represents the Orpheus collection holding the sections of the button header.

Each section in the collection is of the type TOvcButtonHeaderSection. See “TOvcButtonHeaderSection Control” on page 633.

Style property

```
property Style : TOvcButtonHeaderStyle
```

```
TOvcButtonHeaderStyle = (bhsNone, bhsRadio, bhsButton);
```

Default: bhsRadio

Determines whether the buttons on the button header should work as individual speed buttons, as a radio group (where the current section stays down), or not as buttons at all.

TextMargin property

```
property TextMargin : Integer
```

Default: 0

Determines the amount of space around the text painted in each section.

WordWrap property

```
property WordWrap : Boolean
```

Default: False

Turns word wrap on or off.

If WordWrap is True, text is word wrapped if it does not fit in the width of the section.

TOvcButtonHeaderSection Control

The TOvcButtonHeaderSection is used internally by the TOvcButtonHeader component to hold information about each individual section in the button header.

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

❷ TOvcCollectible (OvcBase) 41

TOvcButtonHeaderSection (OvcBtnHd)

Properties

❶	About	❷	Collection		LeftImageIndex
	Alignment	❷	DisplayText	❷	Name
	AllowResize		ImageIndex		Visible
	Caption	❷	Index		Width

Reference Section

Alignment property

```
property Alignment : TAlignment
```

Default: taLeftJustify

Determines the text alignment for the current section.

TAlignment is defined in the VCL's Classes unit.

AllowResize property

```
property AllowResize : Boolean
```

Default: True

Determines whether the section may be resized using the mouse at run time.

See also: TOvcButtonHeader.AllowResize

Caption property

```
property Caption : string
```

Defines the text which should be drawn in the current section.

Hint property

```
property Hint : string
```

Sent to the Application object's OnHint event when the button header receives a CM_HINTSHOW message.

Each section of the button header is capable of displaying its own hint in response to the CM_HINTSHOW message.

ImageIndex property

```
property ImageIndex : Integer
```

Default: -1

Specifies an image from the Images property of the button header to be drawn to the right of the caption (if any) in the current section.

A value of –1 denotes no image. If the value for ImageIndex is out of range, or if the Images property of the button header is nil, this property is ignored.

See also: LeftImageIndex, TOvcButtonHeader.Images

LeftImageIndex property

```
property LeftImageIndex : Integer
```

Default: -1

Specifies an image from the LeftImages property of the button header to be drawn to the left of the caption (if any) in the current section.

A value of –1 denotes no image. If the value for LeftImageIndex is out of range, or if the LeftImages property of the button header is nil, this property is ignored.

See also: ImageIndex, LeftImageIndex, TOvcButtonHeader.LeftImages

Visible property

```
property Visible : Boolean
```

Default: True

Determines whether the section is currently drawn.

Setting Visible to False is conceptually equivalent to temporarily setting value of the width property of the section to zero and the value of the AllowResize property to False.

See also: TOvcButtonHeader.AllowResize

Width property

```
property Width : Integer
```

Default: 75

Determines the width of the current section in pixels.

TOvcAttachedButton Component

TOvcAttachedButton is a descendant of and provides all the functionality of the TBitButton component plus the ability to “attach” to another component.

The TOvcAttachedButton component allows you to associate a button with another control. This association allows changes in the control’s position and size to be reflected in the position and size of the TOvcAttachedButton. The association can be two-way: in addition to the button responding to changes in the control’s position and size, the control can also respond to changes in the position and size of the button.

Hierarchy

TBitBtn (Buttons)

  TOvcAttachedButtons (OvcABtn)

Properties

About	AttachTwoWay	Separation
AttachedControl	Position	

Reference Section

About read-only property

```
property About : string
```

Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

AttachedControl property

```
property AttachedControl : TWinControl
```

Attaches the button to another control.

Setting this property to any TWinControl component (or descendant) permits the TOvcAttachedButton to establish a link into the control's window procedure. This allows the TOvcAttachedButton to detect changes in the control's size and position and adjust its own size and position accordingly. Once this property is set, all movement and size actions applied to the control are reflected to the TOvcAttachedButton component, both at design time and at run time.

If this property is not assigned a value, the AttachTwoWay, Position, and Separation properties are ignored. All the inherited methods and properties are still available, however.

Because of the technique used to track the control's movements and size changes, only one TOvcAttachedButton is allowed to be connected to a component. If you attempt to attach a second TOvcAttachedButton to a component, an exception indicates which control is already attached.

Caution: If both the control and the TOvcAttachedButton are selected (at design time) and moved/sized, one or both will move/size twice the desired amount. Each is informing the other to adjust itself.

AttachTwoWay property

```
property AttachTwoWay : Boolean
```

Default: False

Determines whether changes made in the TOvcAttachedButton component are reflected in the attached control.

Setting AttachTwoWay to True causes changes to the TOvcAttachedButton's size and position to be reflected in the attached control. This behavior is in addition to the normal behavior where changes in the attached control are reflected in the TOvcAttachedButton.

Position property

```
property Position : TButtonPosition
```

```
TButtonPosition = (bpRight, bpLeft, bpTop, bpBottom);
```

Default: bpRight

Determines the position of the attached button.

Position determines which edge of the control to attach the button to. It also determines how changes in the control's size are applied to the button. The following are the valid TButtonPosition values:

Value	**Meaning**
bpRight	The button is to the right of the control. Changes to the size of the control are applied to the height of the button.
bpLeft	The button is to the left of the control. Changes to the size of the control are applied to the height of the button.
bpTop	The button is above the control. Changes to the size of the control are applied to the width of the button.
bpBottom	The button is below the control. Changes to the size of the control are applied to the width of the button.

Separation property

```
property Separation : Integer
```

Default: 1

Represents the number of pixels between the control and the attached button.

Separation must be greater than or equal to 0.

TOvcCustomSpeedButton Class

The TOvcCustomSpeedButton class is the immediate ancestor of the TOvcSpeedButton component. It implements all the methods and properties used by the TOvcSpeedButton component and is identical to the TOvcSpeedButton component except that no properties are published.

TOvcCustomSpeedButton is provided to facilitate creation of descendent SpeedButton components. For property and method descriptions, see “TOvcSpeedButton Component” on page 641.

Hierarchy

TGraphic (VCL)

    TOvcGraphicControl (OvcBase) 82

        TOvcCustomSpeedButton (OvcSpeed)

TOvcSpeedButton Component

TOvcSpeedbutton originated because of the need to have a button-like component that possessed the capability to auto-repeat, like the Orpheus spinner component. In addition to being able to auto-repeat while the button is pressed, TOvcSpeedButton has the ability to be drawn flat and to be made transparent.

Hierarchy

TGraphic (VCL)

❶ TOvcGraphicControl (OvcBase) 82

TOvcCustomSpeedButton (OvcSpeed). 640

TOvcSpeedButton (OvcSpeed)

Properties

❶	About	Layout	Spacing
	AutoRepeat	Margin	Style
	Flat	NumGlyphs	Transparent
	Glyph	RepeatDelay	WordWrap
	GrayedInactive	RepeatInterval	

Methods

ButtonClick

Reference Section

AutoRepeat — property

```
property AutoRepeat : Boolean
```

↳ Determines if holding the button down will result in multiple click events.

If AutoRepeat is True, holding the button down for more than RepeatDelay milliseconds causes additional click events. Exactly how many events are generated depends on the setting of the RepeatInterval property.

See also: RepeatDelay, RepeatInterval

ButtonClick — method

```
procedure ButtonClick;
```

↳ Mimics the action of pressing and holding the button down.

Flat — property

```
property Flat : Boolean
```

↳ Determines whether the button has a a 3D border that provides a raised or lowered look.

Set Flat to True to remove the raised border when the button is unselected and the lowered border when the button is clicked or selected. When Flat is True, use separate bitmaps for the different button states to provide visual feedback to the user about the button state.

See also: Glyph

Glyph — property

```
property Glyph : TBitmap
```

↳ Specifies the bitmap that appears on the speed button.

Set Glyph to a bitmap object that contains the image that should appear on the face of the button. Bring up the Open dialog box from the Object Inspector to choose a bitmap file (with a .BMP extension), or specify a bitmap file at run time.

Glyph can provide up to four images within a single bitmap. All images must be the same size and next to each other in a horizontal row. TOvcSpeedButton displays one of these images depending on the state of the button:

Image Position	**Button State**	**Description**
First	Up	This image appears when the button is unselected. If no other images exist in the bitmap, this image is used for all states.
Second	Disabled	This image usually appears dimmed to indicate that the button can't be selected.
Third	Clicked	This image appears when the button is clicked. If GroupIndex is 0, the Up image reappears when the user releases the mouse button.
Fourth	Down	This image appears when the button stays down indicating that it remains selected.

If only one image is present, TOvcSpeedButton attempts to represent the other states by altering the image slightly for each state, although the Down state is always the same as the Up state.

If the bitmap contains multiple images, specify the number of images in the bitmap with the NumGlyphs property.

Note: The lower left hand pixel of the bitmap is reserved for the transparent color. Any pixel in the bitmap which matches that lower left hand pixel will be transparent.

See also: NumGlyphs

GrayedInactive property

```
property GrayedInactive : Boolean
```

Default: True

Determines if the disabled state is indicated by displaying the second glyph.

If GrayedInactive is True, setting Enabled to False will have no visible effect. If GrayedInactive is False, setting Enabled to False causes the second glyph to be displayed.

Layout property

```
property Layout : TButtonLayout
```

```
TButtonLayout = (
  blGlyphLeft, blGlyphRight, blGlyphTop, blGlyphBottom);
```

Determines where the image or text appears on the speed button.

Set Layout to determine the position of the bitmap specified by the Glyph property or the text specified by the Caption property. These are the possible values:

Value	**Meaning**
blGlyphLeft	The image or caption appears near the left side of the button.
blGlyphRight	The image or caption appears near the right side of the button.
blGlyphTop	The image or caption appears near the top of the button.
blGlyphBottom	The image or caption appears near the bottom of the button.

Margin property

```
property Margin : Integer
```

Specifies the number of pixels between the edge of the button and the image or caption drawn on its surface.

Use Margin to specify the indentation of the image specified by the Glyph property or the text specified by the Caption property. The edges that Margin separates depends on the Layout property. If Layout is blGlyphLeft, the margin appears between the left edge of the image or caption and the left edge of the button. If Layout is blGlyphRight, the margin separates the right edges. If Layout is blGlyphTop, the margin separates the top edges, and if Layout is blGlyphBottom, the margin separates the bottom edges.

If Margin is -1 then the image or text are centered on the button.

NumGlyphs property

```
property NumGlyphs : Integer
```

Specifies the number of images specified in the Glyph property.

Set NumGlyphs to the number of images provided by the bitmap assigned to the Glyph property. All images must be the same size and next to each other in a row. The Glyph property can provide up to four images.

See also: Glyph

RepeatDelay property

```
property RepeatDelay : Integer
```

RepeatDelay determines how many milliseconds are between the initial click and the first click event.

This property is ignored unless the AutoRepeat property is True.

RepeatInterval property

```
property RepeatInterval : Integer
```

Determines how many milliseconds are between individual click events.

This property is ignored unless the Autorepeat property is True.

Spacing property

```
property Spacing : Integer
```

Determines where the image and text appear on a speed button.

Set Spacing to the number of pixels that should appear between the image specified in the Glyph property and the text specified in the Caption property.

If Spacing is a positive number, its value is the number of pixels between the image and text. If Spacing is 0, the image and text appear flush with each other. If Spacing is -1, the text appears centered between the image and the button edge.

Style property

```
property Style : TButtonStyle
```

```
TButtonStyle = (bsAutoDetect, bsWin31, bsNew);
```

↳ Determines the appearance of the button.

The TButtonStyle type contains the values the Style property of bitmap buttons can assume. These are the possible values:

Value	**Meaning**
bsAutoDetect	Legacy style left over from the 16-bit days of Orpheus. Originally it was for determining the button style according to whether the application was running on 16-bit or 32-bit Windows. Since Orpheus no longer supports 16-bit Windows, Setting the style to bsAutoDetect is the same as setting it to bsNew.
bsWin31	Uses the old Windows 3.1 bitmap button look.
bsNew	Uses the 32-bit bitmap button look.

Transparent property

```
property Transparent : Boolean
```

↳ Specifies whether the background of the button is transparent.

Use Transparent to specify whether the background of the button is transparent.

WordWrap property

```
property WordWrap : Boolean
```

↳ Determines whether the control inserts soft carriage returns so text wraps at the right margin of the button.

Set WordWrap to True to make the control wrap text at the right margin so it fits in the client area. Set WordWrap to False to have the control show a separate line only where return characters were explicitly entered into the Caption.

TO32FlexButton Component

TO32FlexButton is a descendant of and provides all the functionality of the TBitButton component plus the ability to provide alternate functionality, selectable by the end-users at run time.

The TO32FlexButton component allows you to associate a menu of alternate functionality, along with an separate glyph for each new function. At run time users may select the functionality that they prefer from a pop-up menu.

Hierarchy

TBitBtn (VCL)

 TO32CustomFlexButton(O32FlxBtn)

  TO32FlexButton (O32FlxBtn)

Properties

About	Items	PopPosition
ActiveItem	PopGlyph	

Methods

PopMenu

Events

OnClick	OnMenuClick
OnMenuPop	OnItemChange

Reference Section

About — read-only property

```
property About : string
```

Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

ActiveItem — property

```
property ActiveItem : Integer
```

Contains the index of the currently active item.

Setting ActiveItem directly causes the FlexButton to load the new functionality.

Note: Setting ActiveItem directly does not result in the OnMenuClick event being fired.

Items — property

```
property Items : TO32Collection
```

Contains the selections that appear in the TO32FlexButton's pop-up menu.

Because TO32Collection is a TCollection descendant, you can add, delete, and modify the O32FLexButtons selections using the Items property.

See also: PopGlyph

PopGlyph — property

```
property PopGlyph : TBitmap
```

Default: nil

Contains the image that indicates the spot on the button's face that will trigger pop-up menu when it is clicked.

The PopGlyph will be drawn in the area of the button's face specified by the PopPosition property.

If PopGlyph is nil then standard + and – characters in a small white box will be drawn in the PopPosition area of the button's face. PopGlyph defines a 3-part, nx3n, 16 color bitmap which holds the three different states of the PopGlyph. Each state is nxn pixels. The first and second sections represent the enabled and disabled states of the '+' image, while the third section represents the "-" image. There is no disabled state for the "-" image as the popmenu cannot be displayed if the control is disabled. The default PopGlyph that ships with Orpheus is a 10x30 bitmap with each state being represented by a 10x10 image.

PopPosition property

```
property PopPosition: TO32FlexButtonPopPosition
```

```
TO32FlexButtonPopPosition = (
  ppBottomLeft, ppBottomRight, ppTopLeft, ppTopRight);
```

Default: ppBottomRight;

↳ Defines the spot on the button's face that will trigger the PopMenu when clicked.

PopPosition is also the spot where the PopGlyph will be drawn. The PopMenu will align itself as follows:

PopPosition Setting	PopMenu Alignment
ppBottomLeft	PopMenu will appear attached to the bottom of the control, with the left side of the menu aligned with the left side of the control.
ppBottomRight	PopMenu will appear attached to the bottom of the control, with the right side of the menu aligned with the right side of the control.
ppTopLeft	PopMenu will appear attached to the top of the control, with the left side of the menu aligned with the left side of the control.
ppTopRight	PopMenu will appear attached to the top of the control, with the right side of the menu aligned with the right side of the control.

OnClick event

```
property OnClick: TO32FlexButtonItemEvent
```

```
TO32FlexButtonItemEvent = procedure(
  Sender : TObject; Item: Integer) of object;
```

Defines an event handler that is generated when the FlexButton is clicked.

Use the value of Item to determine what to do based on which functionality is currently selected.

Below is a sample OnClick event handler for the O32FlexButton:

```
  procedure TForm1.O32FlexButton1Click(Sender: TObject;
    Item: Integer);
  begin
    case Item of
      1 : {Do the item 1 stuff...};
      2 : {Do the item 2 stuff...};
      3 : {Do the item 3 stuff...};
          {etcetra ...}
    end;
  end;
```

OnMenuPop event

```
property OnMenuPop : TNotifyEvent
```

Defines an event handler that is generated when the menu is PopMenu is executed.

OnMenuClick event

```
property OnMenuClick: TO32FlexButtonItemChangeEvent
```

```
TO32FlexButtonItemChangeEvent = procedure(
  Sender : TObject; var OldItem, NewItem: Integer) of object;
```

OnMenuClick defines an event that is generated when the user makes a selection from the PopMenu, unless the ActiveItem isn’t changed as a result of the selection. In other words, if Item 3 is currently the ActiveItem and the user pops the menu down and selects item 3 again, the PopMenu closes, no change is made and the event is not fired.

OldItem contains the index of the previously active selection and NewItem contains the index of the new selection.

Below is a sample event handler for the OnMenuClick event:

```
procedure TForm1.O32FlexButton1MenuClick(
  Sender: TObject; var OldItem, NewItem: Integer);
begin
  case NewItem of
    1 : {Do the item 1 stuff...};
    2 : {Do the item 2 stuff...};
    3 : {Do the item 3 stuff...};
        {etcetra ...}
  end;
end;
```

Note: The FlexButton's ActiveItem is updated after execution returns from the OnMenuClick event handler. Modifying the value of NewItem will result in ActiveItem being set to the new value.

OnItemChange — event

```
property OnItemChange: TO32FlexButtonItemChangeEvent;
```

```
TO32FlexButtonItemChangeEvent = procedure(
  Sender : TObject; var OldItem, NewItem: Integer) of object;
```

↳ Defines an event handler that is generated when the FlexButton's ActiveItem is changed.

Note: The FlexButton's ActiveItem is changed before the OnItemChange event is fired. Modifying the value of NewItem at this point will be ignored by the control.

PopMenu — method

```
procedure PopMenu;
```

↳ Allows programmatic execution of the PopMenu.

Calling PopMenu will create the same result as if the user clicked on the PopPosition area of the control.

Chapter 21: Transfer Component

The TOvcTransfer component provides a mechanism to transfer data to and from the fields of a form using a single data structure instead of setting and retrieving each field's value individually. It's pretty cool.

The data structure used by TOvcTransfer corresponds to the data required by the components on the form. For example, if the form contains two Orpheus simple entry fields that are configured to edit Byte data types, the required data structure is a record containing two variables of type Byte.

Admittedly this is a simple case and setting or retrieving the value for two components is hardly justification for use of the transfer component. Consider a form containing 20 edit fields, a couple of list boxes, a combo box, a memo field, and an assortment of checkboxes and radio buttons. Setting and retrieving the data for this form is much more error prone and tedious.

This is where TOvcTransfer is most useful. One data structure and two method calls are all that is needed. The transfer component's property editor even generates the necessary data structure, transfer methods, and a shell for an initialization routine. All you have to do is paste them into your application.

TOvcTransfer supports data transfer for the following component types:

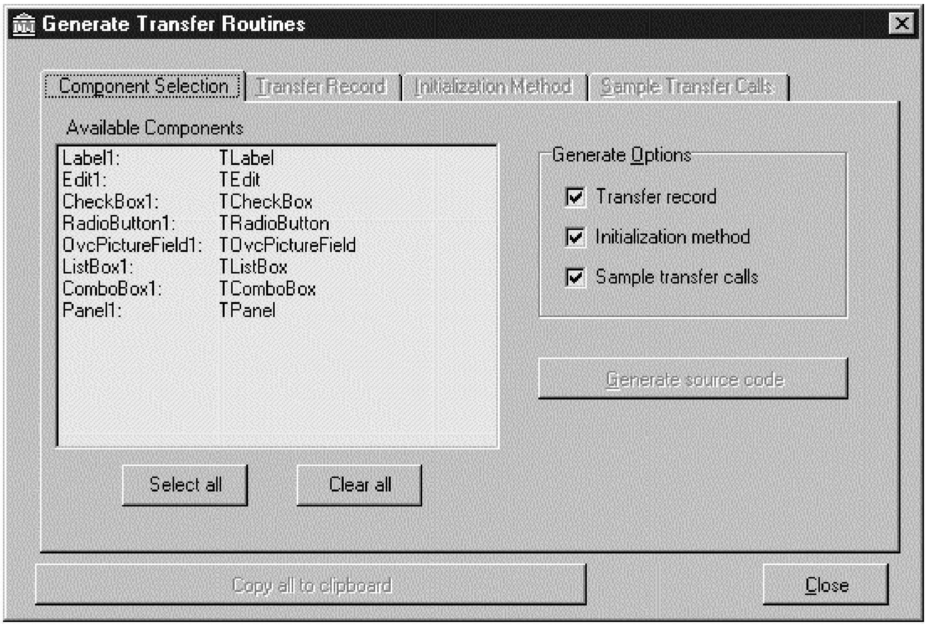
Component	**Data Type**
TLabel	String
TPanel	String
TEdit	String
TMemo	TStrings
TCheckBox	Boolean
TRadioButton	Boolean
TListBox	TStrings, Integer
TComboBox	TStrings, string, Integer
TOvcRotatedLabel	String
TOvcSimpleEntryField	Variable (size obtained from field)
TOvcPictureEntryField	Variable (size obtained from field)
TOvcNumericEntryField	Variable (size obtained from field)

Component editor

The Orpheus transfer component provides a sophisticated component editor that is accessible from the TOvcTransfer component's local menu (the local menu is accessed by right-clicking the transfer component on the form at design time). It displays a list of the components on the form that the transfer component supports, allows you to select the components that will participate in the transfer process, and generates the transfer record, initialization method, and transfer calls.

Component Selection

When the component editor is first displayed, it lists all the supported components on the form in the Available Components list (see Figure 21.1).



Label1:	TLabel
Edit1:	TEdit
CheckBox1:	TCheckBox
RadioButton1:	TRadioButton
OvcPictureField1:	TOvcPictureField
ListBox1:	TListBox
ComboBox1:	TComboBox
Panel1:	TPanel

☑ Transfer record
☑ Initialization method
☑ Sample transfer calls

Figure 21.1: Component Selection tab

Available components

This list box displays the components available on the form when the component editor was invoked. Select the items that you want to participate in the transfer process. After components are selected, the Generate source code button is enabled.

Select all

Use this button to select all items in the list box.

Clear all

Use this button to unselect all items in the list box.

Generate options

This group of checkboxes allows you to select the source code generation options. The items you select depend on whether you are prepared to paste the generated source code into your application. Selecting an item that you are not ready to use does no harm—just don't copy it to the clipboard or, if you do paste it into your application, delete the portion of the code that you don't need.

Generate source code

This button generates source code for the components selected in the Available Components list box, places the generated code in the memo component for the selected components, and makes the appropriate tabs active.

Copy all to clipboard

After source code is generated, selecting this button copies the generated code to the Windows clipboard. For example, if the Transfer record and Sample transfer calls checkboxes are checked, selecting this button copies the contents of the edit fields from those two notebook pages.

Close

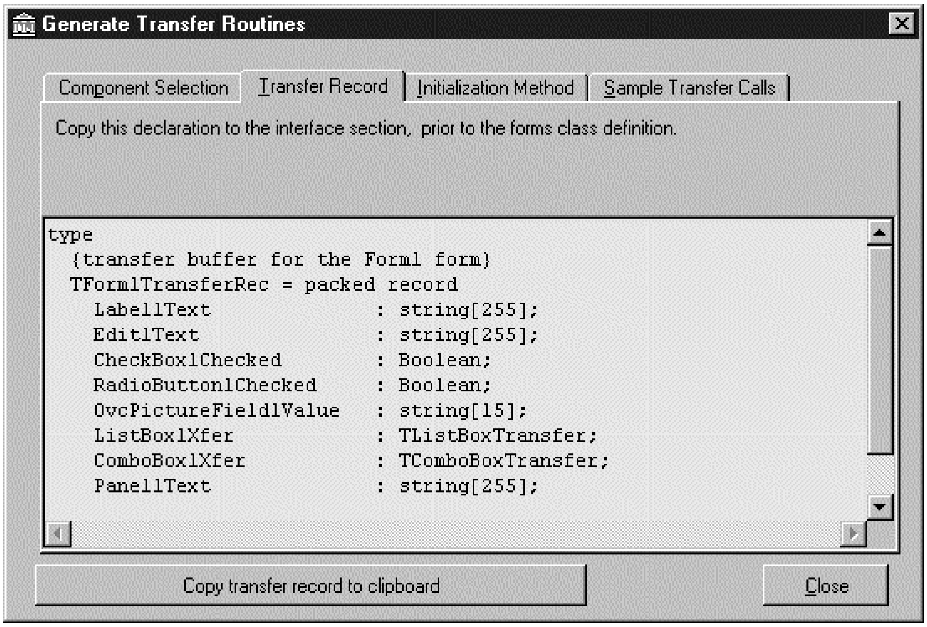
Select this button to close the Generate Transfer Routines dialog box.

Help

Select this button to obtain help specific to this form.

Transfer Record

If the Transfer Record checkbox (shown in Figure 21.2) is checked and Generate source code is selected, the Transfer Record tab is made active. When you select the tab, it displays the source code for the definition of the transfer record. The name of the record is created using the name of your form. The names of the selected components are used to create the names of the variables in the transfer record.



Copy this declaration to the interface section, prior to the forms class definition.

```
type
  {transfer buffer for the Form1 form}
  TForm1TransferRec = packed record
    Label1Text              : string[255];
    Edit1Text               : string[255];
    CheckBox1Checked        : Boolean;
    RadioButton1Checked     : Boolean;
    OvcPictureField1Value   : string[15];
    ListBox1Xfer            : TListBoxTransfer;
    ComboBox1Xfer           : TComboBoxTransfer;
    Panel1Text              : string[255];
```

Figure 21.2: Transfer Record checkbox

Press the Copy transfer record to clipboard button to place a copy of the record definition in the Windows clipboard. The text placed in the clipboard looks like this:

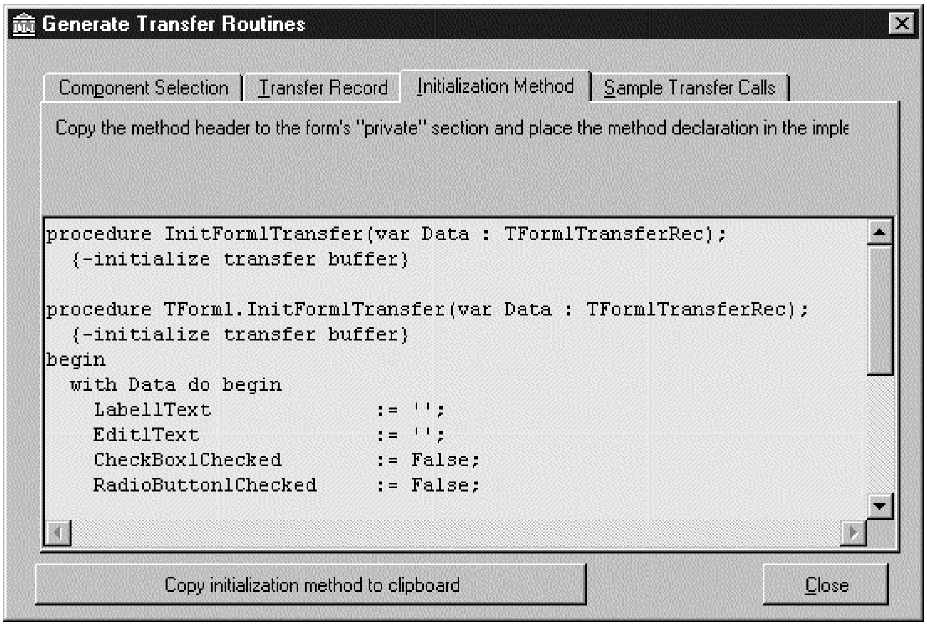
```
>>> Transfer record <<<
type
  {transfer buffer for the Form1 form}
  TForm1TransferRec = packed record
    Label1Text              : string[255];
    Edit1Text               : string[255];
    CheckBox1Checked        : Boolean;
    RadioButton1Checked     : Boolean;
    OvcPictureField1Value   : string[15];
    ListBox1Xfer            : TListBoxTransfer;
    ComboBox1Xfer           : TComboBoxTransfer;
    Panel1Text              : string[255];
  end;
```

TListBoxTransfer and TComboBoxTransfer are defined in the OvcXfer unit as:

```
{structure used to transfer data for a TListBox component}
PListBoxTransfer = ^TListBoxTransfer;
TListBoxTransfer = record
  ItemIndex : Integer;
  Items     : TStrings;
end;

{structure used to transfer data for a TComboBox component}
PComboBoxTransfer = ^TComboBoxTransfer;
TComboBoxTransfer = record
  ItemIndex : Integer;
  Text      : string;
  Items     : TStrings;
end;
```

Initialization Method

If the Initialization Method checkbox (shown in Figure 21.3) is checked and Generate source code is selected, the Initialization Method tab is made active. When you select the tab, it displays the source code for the definition of the initialization method.



Copy the method header to the form's "private" section and place the method declaration in the imple

```
procedure InitForm1Transfer(var Data : TForm1TransferRec);
  {-initialize transfer buffer}

procedure TForm1.InitForm1Transfer(var Data : TForm1TransferRec);
  {-initialize transfer buffer}
begin
  with Data do begin
    Label1Text          := '';
    Edit1Text           := '';
    CheckBox1Checked    := False;
    RadioButton1Checked := False;
```

Figure 21.3: Initialization Method checkbox

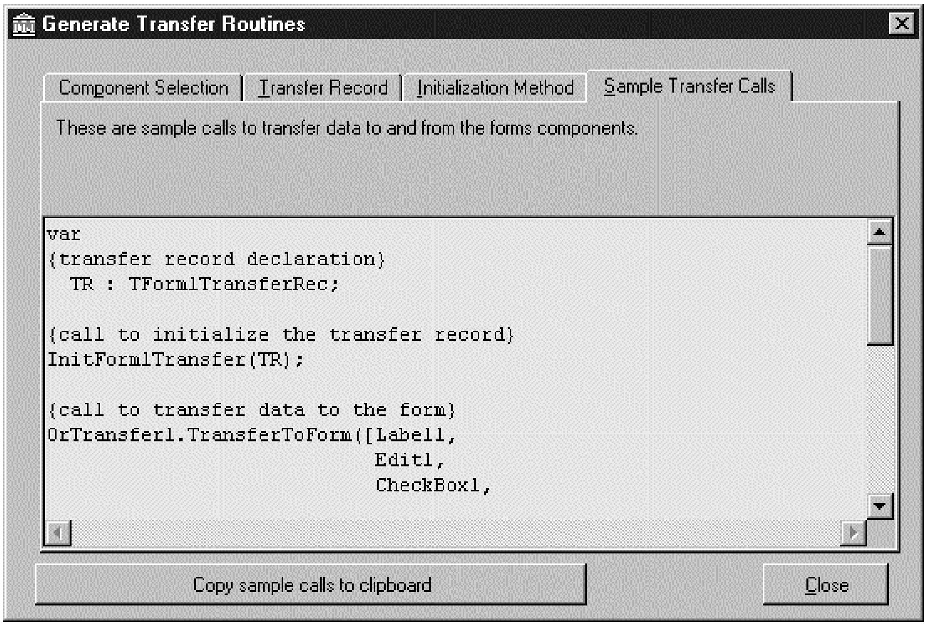
The names of the selected components are used to create the names of the variables in the transfer record.

Select “Copy initialization method to clipboard” to place a copy of the method in the Windows clipboard. The text placed in the clipboard looks like this:

```
>>> Initialization header and method <<<

procedure InitForm1Transfer(var Data : TForm1TransferRec);
  {-initialize transfer buffer}

procedure TForm1.InitForm1Transfer(var Data : TForm1TransferRec);
  {-initialize transfer buffer}

begin
  with Data do begin
    Label1Text              := '';
    Edit1Text               := '';
    CheckBox1Checked        := False;
    RadioButton1Checked     := False;
    OvcPictureField1Value   := '';
    ListBox1Xfer.Items      := TStringList.Create;
    ListBox1Xfer.ItemIndex  := 0;
    ComboBox1Xfer.Items     := TStringList.Create;
    ComboBox1Xfer.ItemIndex := 0;
    ComboBox1Xfer.Text      := '';
    Panel1Text              := '';
  end; {with}
end;
```

Sample Transfer Calls

If the Sample Transfer Calls checkbox (shown in Figure 21.4) is checked and Generate source code is selected, the Sample Transfer Calls tab is made active. When you select the tab, it displays the source code for the definition of the transfer methods.



These are sample calls to transfer data to and from the forms components.

```
var
{transfer record declaration}
  TR : TForm1TransferRec;

{call to initialize the transfer record}
InitForm1Transfer(TR);

{call to transfer data to the form}
OrTransfer1.TransferToForm([Label1,
                            Edit1,
                            CheckBox1,
```

Figure 21.4: Sample Transfer Calls checkbox

The names of the selected components are specified in the array parameter of the TransferToForm and TransferFromForm calls.

Warning: The order of these names must match the declaration order in the transfer record.

Press the Copy sample calls to clipboard button to place a copy of these calls in the Windows clipboard. The text placed in the clipboard looks like this:

```
>>> Sample transfer calls <<<

var
  {transfer record declaration}
  TR : TForm1TransferRec;
...
{call to initialize the transfer record}
InitForm1Transfer(TR);
...
{call to transfer data to the form}
OvcTransfer1.TransferToForm([Label1, Edit1, CheckBox1,
                             RadioButton1, OvcPictureField1,
                             ListBox1, ComboBox1, Panel1], TR);
...
{call to transfer data from the form}
OvcTransfer1.TransferFromForm([Label1, Edit1, CheckBox1,
                               RadioButton1, OvcPictureField1,
                               ListBox1, ComboBox1, Panel1],
                               TR);
```

Example

See the EXTRAN or ADDRBOOK projects for an example that uses the TOvcTransfer component.

TOvcTransfer Component

TOvcTransfer provides a method to transfer data to and from a form by passing one structure for all components on the form.

Hierarchy

TComponent (VCL)

❶ TOvcComponent (OvcBase) 21

TOvcTransfer (OvcXfer)

Properties

❶ About

Methods

GetTransferBufferSize	TransferFromForm	TransferToForm

Reference Section

GetTransferBufferSize method

```
function GetTransferBufferSize(CNA : array of TComponent) : Word;
```

The size required for the transfer buffer.

Use of this routine is not required by the transfer component, but is recommended during design and testing to ensure that the transfer buffer is the correct size.

CNA is the component name array. It is a list of the components for which data will be transferred.

The following example gets the required transfer buffer size for a label and an edit component and then tests to see if it is the correct size before transferring the data to the form:

```
type
  TransferBufferRec = record
    LabelText : string;
    EditText  : string;
  end;
...
var
  MyTransferBuffer : TransferBufferRec;
...
Size := OvcTransfer1.GetTransferBufferSize([Label1, Edit1]);
if Size <> SizeOf(MyTransferBuffer) then
  raise Exception.Create('Invalid transfer buffer size')
else
  OvcTransfer1.TransferToForm([Label1, Edit1], MyTransferBuffer);
```

See also: TransferToForm

TransferFromForm method

```
procedure TransferFromForm(CNA : array of TComponent; var Data);
```

Transfers the data from the components to the data structure.

TransferFromForm transfers the data from each of the components listed in the component name array specified by CNA to the data structure specified by Data. Data must be the correct size or a memory overwrite can occur. Use GetTransferBufferSize to verify the size of the transfer buffer.

Delphi users must use the packed keyword when defining the transfer record in your application. C++Builder users should see the EXTRAN example for information on byte-aligned structures.

The following example transfers data from the form to MyTransferBuffer record:

```
  OvcTransfer1.TransferFromForm([Label1, Edit1],
    MyTransferBuffer);
```

See also: GetTransferBufferSize, TransferToForm

TransferToForm method

```
procedure TransferToForm(CNA : array of TComponent; const Data);
```

Transfers data from the data structure to the components.

TransferToForm transfers the data from the data structure specified by Data to each of the components listed in the component name array specified by CNA. Data must be the correct size or a memory overread can result. Use GetTransferBufferSize to verify the size of the transfer buffer.

Delphi users must use the packed keyword when defining the transfer record in your application. C++Builder users should see the EXTRAN example for information on byte-aligned structures.

The following example transfers data from the MyTransferBuffer record to the specified components on the form:

```
  OvcTransfer1.TransferToForm([Label1, Edit1], MyTransferBuffer);
```

See also: GetTransferBufferSize, TransferFromForm

Chapter 22: List Boxes

This section introduces several list box or list box-like components. Each of the list box components is described in brief here, and in detail in the following sections.

The TOvcListBox component is a direct descendant of the standard VCL TListBox component that adds easier tabset handling and the ability to attach a label. It also serves as a common ancestor for several of the components described in this section.

The TOvcSearchList is an incremental-search list box that is a combination of a standard VCL TListBox component and the TOvcBaseISE class (see page 1136).

The TOvcCheckList provides all of the standard TListBox features plus the ability to display a check mark or an 'X' beside selected items. You can also display bitmaps that are associated with the list items. Tri-state check boxes are supported.

The TOvcDbColumnList component displays the values of a specified database field in a column. Each row represents a different record in the database.

The TOvcVirtualListBox is similar to and behaves much like the standard VCL TListBox component, but can display over 2 billion list box items. The virtual list box requires very little memory. Unlike the standard TListBox component, TOvcVirtualListBox does not maintain a copy of the items it displays—it calls an event handler that you supply to obtain the display strings.

The TOvcVirtualListBox provides support for single and multiple selection, protected items, a list header line, horizontal scrolling, individual colors for the list box items, and tab stops in the list items and in the header. The virtual list box is configurable at design time and at run time. It is most valuable in situations where you need to display a very large number of items and it is difficult to know or obtain the display strings until just prior to displaying them. It is also useful when you need more control over the look of the displayed list.

TOvcCustomListBox Class

The TOvcCustomListBox class is the immediate ancestor of the TOvcListBox component. It implements all of the methods and properties used by the TOvcListBox component and is identical to TOvcListBox except that no properties are published.

TOvcCustomListBox is provided to facilitate creation of descendent list box components. For property and method descriptions, see “TOvcListBox Component” on page 667.

Hierarchy

TCustomListbox (VCL)

    TOvcCustomListBox (OvcLB)

TOvcListBox Component

TOvcListBox component adds to the standard TListBox the capability to display an attached label (see "TOvcAttachedLabel Component" on page 762), display a horizontal scroll bar, and an enhanced method for dealing with embedded tabs and tab stop definitions.

The TabStops property allows you to define tab stop positons using character counts rather than dialog units as the standard TListBox components requires.

Hierarchy

TCustomListbox (VCL)

  TOvcCustomListBox (OvcLB). 666

    TOvcListBox (OvcLB)

Properties

About	Controller	LabelInfo
AttachedLabel	HorizontalScroll	TabStops

Methods

ClearTabStops	ResetHorizontalScrollbar	SetTabStops

Events

OnTabStopsChange

Reference Section

About — read-only property

```
property About : string
```

↳ Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

AttachedLabel — property

```
property AttachedLabel : TOvcAttachedLabel
```

↳ Provides access to the TOvcAttachedLabel component.

The TOvcAttachedLabel is normally accessed only at design time, but this property makes it possible to alter the attached label at run time if necessary.

See “TOvcAttachedLabel Component” on page 762 for more information.

ClearTabStops — method

```
procedure ClearTabStops;
```

↳ Clears the current tab stop values from the TabStops string list.

See also: TabStops

Controller — property

```
property Controller : TOvcController
```

Default: The first TOvcController object on the form

↳ The TOvcController object that is attached to this component.

If this property is not assigned, some or all the features provided by the component will not be available. The Controller property is not published, but descendants can publish it as needed (as most of the Orpheus components do). See “TOvcController Component” on page 28 for additional information.

HorizontalScroll — property

```
property HorizontalScroll : Boolean
```

↳ Determines if a horizontal scroll bar is displayed.

LabelInfo — property

```
property LabelInfo : TOvcLabelInfo
```

↳ Provides access to the status of the attached label.

TOvcLabelInfo (see page 764) groups the Visible, OffsetX, and OffsetY properties that determine whether the attached label is visible and where it is positioned.

OnTabStopsChange — event

```
property OnTabStopsChange : TNotifyEvent
```

↳ Defines an event handler that is fired whenever the values of the tab stops are altered.

ResetHorizontalScrollbar — method

```
procedure ResetHorizontalScrollbar;
```

↳ Causes the setting for the horizontal scroll bar to be recalculated.

Calls to ResetHorizontalScrollbar are ignored if HorizontalScroll is False.

See also: HorizontalScroll

SetTabStops — method

```
procedure SetTabStops(Value : array of Integer);
```

↳ Assigns tab stop values based on the contents of the Value array.

Values in the Value array represent character positions.

TabStops — property

```
property TabStops : TStrings
```

↳ Defines a list of values that determine the position for tab stops as character counts.

For example, entering values of 5, 10, and 20 into the string list as separate items will cause the list box to use those values as the number character positions between the first, second, and third tab stop positions.

TOvcSearchList Component

The TOvcSearchList component is a combination of a TListbox component and the TOvcBaseISE class providing an incremental-search list box. The searching capabilities of the search list component are the same as those provided by the TOvcBaseISE Class (see page 1136). The contained list box is a standard TListBox used to provide the items to search and to indicate the found item. Both of these contained components can be accessed by properties of the TOvcSearchList component.

The contained list box component's Sorted property is set to True by default and must remain so. Using the ListBox property to set Sorted to False is possible, but it causes searching to be unreliable or not to work at all.

Hierarchy

TCustomControl (VCL)

- ❶ TOvcCustomControl (OvcBase) 16
  - ❷ TOvcCustomControlEx (OvcBase) 19
    - TOvcSearchList (OvcISLB)

Properties

❶ About	Columns	Items
❶ AttachedLabel	❷ Controller	KeyDelay
AutoSearch	Edit	❶ LabelInfo
AutoSelect	HideSelection	ListBox
BorderStyle	ItemHeight	PasswordChar
CaseSensitive	ItemIndex	ShowResults

Events

❶ AfterEnter	❶ AfterExit	❶ OnMouseWheel

Reference Section

AutoSearch property

```
property AutoSearch : Boolean
```

Default: True

↳ Determines whether the search is performed automatically.

If AutoSearch is True, the search for the entered text is performed as soon as KeyDelay milliseconds have passed since the last character was entered. If AutoSearch is False, you must call PerformSearch to do the search.

See also: KeyDelay, PerformSearch

AutoSelect property

```
property AutoSelect : Boolean
```

Default: True

↳ Determines whether the edit field contents are selected when it receives the focus.

If AutoSelect is True, the text in the edit box is automatically selected when the user tabs to the control.

BorderStyle property

```
property BorderStyle : TBorderStyle
```

Default: bsSingle

↳ Determines the style used when drawing the border .

The possible values are bsSingle (a single line border is drawn around the component) or bsNone (no border line).

The TBorderStyle type is defined in the VCL's Controls unit.

CaseSensitive protected property

```
property CaseSensitive : Boolean
```

Default: False

↳ Determines whether the search is case sensitive.

Columns property

```
property Columns : LongInt
```

Default: 0

↳ The number of columns in the list box.

Edit run-time property

```
property Edit : TOvcBaseISE
```

↳ Provides access to the contained TEdit control and its properties and methods.

HideSelection property

```
property HideSelection : Boolean
```

Default: True

↳ Determines whether selected text remains selected when the focus shifts to another control.

If HideSelection is False, highlighted text remains highlighted even if the focus leaves the search list component. If HideSelection is True, highlighting is removed when the search list loses the focus and is restored when the search list regains the focus.

ItemHeight property

```
property ItemHeight : Integer
```

↳ Determines the height (in pixels) of an item in the list box.

ItemIndex run-time property

```
property ItemIndex : Integer
```

↳ Determines the currently selected item in the list box.

Items — property

```
property Items : TStrings
```

↳ Contains the strings that appear in the list box.

Because Items is an object of type TStrings, you can add, delete, insert, and move items using the methods of the TStrings object.

The ItemIndex property determines which item is selected, if any.

KeyDelay — property

```
property KeyDelay : Integer
```

Default: 500

↳ The number of milliseconds to wait after keyboard entry before performing the database search.

This property is used only if AutoSearch is True.

See also: AutoSearch

ListBox — run-time property

```
property ListBox : TListBox
```

↳ Provides access to the contained TListBox component.

See TListBox in the VCL online help for more information.

PasswordChar — property

```
property PasswordChar : Char
```

Default: Empty string

↳ Lets you create an edit box that displays special characters in place of the entered text.

By default, PasswordChar is the null character (ANSI character zero), meaning that the control displays its text normally. If you set PasswordChar to any other character, the control displays that character in place of each character in the control's text.

```
property ShowResults : Boolean
```

Default: True

Determines whether the edit display is updated with the result of the search.

If ShowResults is True and the search was successful, the edit display is updated with the result of the search.

The portion of the text that is appended to the end of the search criteria (the entered characters) is selected. This makes it easy to continue searching (the next entered character replaces the selected text).

See also: KeyDelay

TOvcCheckList Component

The TOvcCheckList component is a descendant of the TCustomListBox component. In addition to the standard TListBox features, it provides the capability to display a check mark or an ‘X’ to the left of the currently selected items.

This type of list box is often used in Setup programs for custom installations where there are several optional components. The primary use of TOvcCheckList is to provide a multi-selection list of items where selected items are checked. The check mark or ‘X’ at the left of selected items provides a visual alternative to the highlighting provided by the standard list box.

In addition to providing a checked selection indicator, the TOvcCheckList makes it easy to display bitmaps that are associated with the list items through the use of the Glyphs property.

Hierarchy

TCustomListBox (VCL)

  TOvcCustomListBox (OvcLB). 666

    ❶ TOvcListBox (OvcLB) 667

      TOvcCheckList (OvcCkLB)

Properties

- ❶ About
- ❶ AttachedLabel
- BoxClickOnly
- BoxColor
- BoxMargin
- CheckColor
- CheckStyle
- ❶ Controller
- GlyphMaskColor
- Glyphs
- ❶ HorizontalScroll
- ❶ LabelInfo
- States
- ❶ TabStops
- ThreeState
- ShowGlyphs
- WantDblClicks

Methods

- ❶ ClearTabStops
- ❶ ResetHorizontalScrollbar
- ❶ SetTabStops

Events

- OnStateChange
- ❶ OnTabStopsChange

Reference Section

BoxClickOnly property

```
property BoxClickOnly : Boolean
```

Default: True

↳ Determines where the user must click to check or uncheck an item.

If BoxClickOnly is True (the default), the user must click on the check box to check or uncheck the list item. If BoxClickOnly if False, the user can click on the text of the list item or the check box.

BoxColor property

```
property BoxColor : TColor
```

Default: clWhite

↳ Determines the background color of the box that is drawn around the check mark.

BoxMargin property

```
property BoxMargin : Integer
```

Default: 2

↳ Determines the size (in pixels) of the margin around the check box.

If you increase BoxMargin and do not change ItemHeight, the spacing on all sides of the check box is increased and the size of the check box is decreased.

CheckColor property

```
property CheckColor : TColor
```

Default: clWindowText

↳ Determines the color used to draw the check mark.

CheckStyle property

```
property CheckStyle : TOvcCheckStyle
```

```
TOvcCheckStyle = (csCheck, csX);
```

Default: csCheck

Determines the style of check mark used.

Possible values for CheckStyle are csCheck (a check mark) or csX (an X).

GlyphMaskColor property

```
property GlyphMaskColor : TColor
```

Default: clWhite

The color used as a mask when displaying the bitmaps associated with the list items.

Set GlyphMaskColor to the color used as the background for the bitmaps so that the background can be removed when the bitmap is drawn.

Glyphs run-time indexed property

```
property Glyphs[Index : Integer] : TBitmap
```

Provides access to the bitmap object associated with the list item.

This indexed property provides access to the Items.Objects property of the standard list box so that TBitmap objects can be associated with the list items. If a valid TBitmap object is assigned to a list item and ShowGlyphs is True, the bitmap is displayed between the checkbox and the text of the list item.

See also: ShowGlyphs

OnStateChange event

```
property OnStateChange  : TOvcStateChangeEvent
```

```
TOvcStateChangeEvent = procedure (
  Sender : TObject; Index : Integer;
  OldState, NewState : TCheckBoxState) of object;
```

↳ Defines an event handler that is fired each time the state of a list item is changed.

ShowGlyphs property

```
property ShowGlyphs : Boolean
```

Default: False

↳ Determines if the bitmaps assigned to the Glyphs property are displayed.

If ShowGlyphs is True, glyphs associated with the list items are displayed between the check box and the item text.

See also: Glyphs

States property

```
property States[Index : Integer] : TCheckBoxState
```

↳ An indexed property that allows reading and setting the state of individual list box items.

Possible values that may be assigned to the individual items are (cbUnchecked, cbChecked, or cbGrayed).

ThreeState property

```
property ThreeState : Boolean
```

Default: False

ThreeState determines if the state of the items in the list box can can be set to one of three states (cbUnchecked, cbChecked, or cbGrayed), rather than two (cbUnchecked and cbChecked).

WantDblClicks property

```
property WantDblClicks : Boolean
```

Default: True

Determines whether the check-list accepts mouse double clicks.

If WantDblClicks is True, the method assigned to the OnDblClick event is called when the mouse is double-clicked. If WantDblClicks is False, a double mouse click is treated as two single clicks.

TOvcDbColumnList Component

This component, although similar to a data-aware array editor, is actually more like the virtual list box because the displayed values cannot be edited. The TOvcDbColumnList component displays the values of a specified database field in a column, with each row representing a different record in the database. The list can be scrolled both vertically and horizontally.

You can display the associated TField's DisplayLabel at the top of the list of items or define custom text to be used as the header. Row indicators (showing the active record and its edit/insert state) can be displayed or hidden. The optional scroll bars can be vertical, horizontal, or both.

Column list commands

The commands listed in Table 22.1 are available in the Orpheus column list. The commands and key assignments described here are available when you use the "Default" or "WordStar" command tables (several command tables can be used at the same time). See "TOvcCommandProcessor Class" on page 50 for information on how to customize the command tables.

Table 22.1: *Column list commands*

Command	Default	WordStar	Description
ccDown	<Down>	<CtrlX>	Move the focus down to the next record.
ccLeft	<Left>	<CtrlS>	Scroll the list to the left.
ccRight	<Right>	<CtrlD>	Scroll the list to the right.
ccUp	<Up>	<CtrlE>	Move the focus up to the previous record.
ccHome	<Home>	<CtrlQ><S>	Scroll left to the beginning of the field.
ccEnd	<End>	<CtrlQ><D>	Scroll right to the end of the field.
ccFirstPage	<CtrlHome>	Not assigned.	Move the focus to the first record.

Table 22.1: *Column list commands (continued)*

Command	Default	WordStar	Description
ccLastPage	<CtrlEnd>	Not assigned.	Move the focus to the last record.
ccPrevPage	<PgUp>	<CtrlR>	Move the focus up one page.
ccNextPage	<PgDn>	<CtrlC>	Move the focus down one page.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase)..16

❷ TOvcCustomControlEx (OvcBase).....................................19

TOvcDbColumnList (OvcDbCL)

Properties

❶ About	Header	RowHeight
❶ AttachedLabel	HeaderColor	RowIndicatorWidth
AutoRowHeight	HideSelection	ScrollBars
BorderStyle	HighlightColors	ShowHeader
❷ Controller	IntegralHeight	ShowIndicator
DataField	❶ LabelInfo	TextMargin
DataSource	LineColor	
Field	PageScroll	

Methods

InvalidateItem

Events

❶ AfterEnter	OnClickHeader	❶ OnMouseWheel
❶ AfterExit	OnIndicatorClick	OnUserCommand

Reference Section

AutoRowHeight — property

```
property AutoRowHeight : Boolean
```

Default: True

Determines whether the row height should be automatically adjusted based on the font size.

If AutoRowHeight is True (the default), the column list automatically calculates the row height based on the size of the currently selected font. Changing the font at run time forces the row height to be recalculated.

See also: RowHeight

BorderStyle — property

```
property BorderStyle : TBorderStyle
```

Default: bsSingle

Determines the style used when drawing the border.

The possible values are bsSingle (a single line border is drawn around the component) or bsNone (no border line).

The TBorderStyle type is defined in the VCL's Controls unit.

DataField — property

```
property DataField : string
```

Identifies the field (in the data source) from which the column list component displays data.

DataSource — property

```
property DataSource : TDataSource
```

Specifies the data source component that contains the data to display.

Field — run-time, read-only property

```
property Field : TField
```

Returns the TField object to which the column list component is linked.

Use the Field object to change the value of the data in the field programmatically.

Header property

```
property Header : string
```

↳ The text displayed as the first row in the column list.

If a string is assigned to Header and ShowHeader is True, the text is displayed as the first row in the column list. If this property is set to an empty string and the ShowHeader property is True, the associated DataField's DisplayLabel text is used as the header text.

See also: DataField, ShowHeader

HeaderColor property

```
property HeaderColor : TOvcColors
```

Default: clBtnText and clBtnFace

↳ Determines the colors used to display the header text and background.

The default color choices of clBtnText for the text color and clBtnFace for the background cause the header to be displayed using the standard button look.

See "TOvcColors Class" on page 1152 for more information.

See also: Header, ShowHeader

HideSelection property

```
property HideSelection : Boolean
```

Default: True

↳ Determines whether highlighted text remains highlighted when the column list loses the focus.

If HideSelection is False, highlighted text remains highlighted even if the focus leaves the column list component. If HideSelection is True, highlighting is removed when the column list loses the focus and is restored when the column list regains the focus.

HighlightColors property

```
property HighlightColors : TOvcColors
```

Default: clHighlightText and clHighlight

↳ Are the colors used to display the selected record in the column list.

See "TOvcColors Class" on page 1152 for more information.

IntegralHeight property

```
property IntegralHeight : Boolean
```

Default: True

Indicates whether the column list automatically resizes itself to display only complete list items.

If IntegralHeight is True, the column list is resized so that partial rows are not displayed at the bottom of the list box. If IntegralHeight is False, partial rows might be displayed.

InvalidateItem method

```
procedure InvalidateItem(Row : Integer);
```

Causes the specified row to be repainted.

LineColor property

```
property LineColor : TColor
```

Default: clSilver

Determines the color of the row divider lines.

To hide the row divider lines, set LineColor to the same value as the Color property.

OnClickHeader event

```
property OnClickHeader : THeaderClickEvent
```

```
THeaderClickEvent = procedure(
  Sender : TObject; Point : TPoint) of object;
```

Defines an event handler that is called when a mouse click occurs in the header area.

The method assigned to OnClickHeader is called to provide notification of a mouse click in the header area. Point contains the X and Y (screen-relative) position of the mouse.

OnIndicatorClick event

```
property OnIndicatorClick : TIndicatorClickEvent
```

```
TIndicatorClickEvent = procedure(
  Sender : TObject; Row : LongInt) of object;
```

Defines an event handler that is called to provide notification that the indicator has been clicked.

Row is the visible row clicked. Sender is the class instance of the column list.

OnUserCommand event

```
property OnUserCommand : TUserCommandEvent
```

```
TUserCommandEvent = procedure(
  Sender : TObject; Command : Word) of object;
```

Defines an event handler that is called when a user-defined command is entered.

The method assigned to the OnUserCommand event is called when the key sequence corresponding to one of the user-defined commands (ccUser1, ccUser2, etc.) is entered.

For example, suppose you add the <CtrlD> key sequence to one of the active command tables and assign it to the ccUser1 command. When <CtrlD> is pressed, the method assigned to the OnUserCommand event is called and ccUser1 is passed in Command.

See "TOvcCommandProcessor Class" on page 50 for additional information about user-defined commands and command tables.

PageScroll property

```
property PageScroll : Boolean
```

Default: False

Defines the scroll behavior of the column list.

When PageScroll is False (the default), the active row moves incrementally in the direction of the scroll until the last visible row is reached. All visible cells are then scrolled. When PageScroll is True, the active row jumps by one page in the direction of the scroll before scrolling all visible rows. This behavior is consistent with the scrolling behavior of VCL grids.

RowHeight property

```
property RowHeight : Integer
```

Default: 20

↳ The height of the column list rows in pixels.

The row height is initially determined based on the current font size. Assigning a new value to the RowHeight property causes the column list to resize each row. The overall height of the column list is adjusted if necessary to avoid displaying a partial row. Assigning a value to this property sets the AutoRowHeight property to False.

RowIndicatorWidth property

```
property RowIndicatorWidth : Integer
```

Default: 11

↳ The width of the row indicators.

The RowIndicatorWidth property is used to determine the width (in pixels) of the row indicator that is shown to the left of the column list. The minimum value for RowIndicatorWidth is 0. Setting RowIndicatorWidth to 0 also sets the ShowIndicator property to False.

See also: ShowIndicator

ScrollBars property

```
property ScrollBars : TScrollStyle
```

Default: ssVertical

↳ Determines whether the column list displays scroll bars.

If ScrollBars is set to ssNone, no scroll bars are shown. If it is sbVertical, only a vertical scroll bar is shown. If it is sbHorizontal, only a horizontal scroll bar is shown. If it is sbBoth, both horizontal and vertical scroll bars are shown. The TScrollStyle enumerated type is defined in the VCL's Controls unit.

ShowHeader property

```
property ShowHeader : Boolean
```

Default: False

Determines whether the header is displayed.

If ShowHeader is True, the text assigned to the header property is displayed at the top of the list box. If it is False, no header is displayed.

ShowIndicator property

```
property ShowIndicator : Boolean
```

Default: True

Determines if the active record indicator is displayed.

If ShowIndicator is True, an indicator showing the active record and its editing state is displayed at the left edge of the column list.

See also: RowIndicatorWidth

TextMargin property

```
property TextMargin : Integer
```

Default: 1

Determines the text indent.

The TextMargin property controls the left indent in pixels for the text displayed in each row of the column list component.

TOvcCustomVirtualListbox Class

The TOvcCustomVirtualListbox class is the immediate ancestor of the TOvcVirtualListBox component. It implements all the methods and properties used by the TOvcVirtualListBox component and is identical to the TOvcVirtualListBox component except that no properties are published.

TOvcCustomVirtualListbox is provided to facilitate creation of descendent list box components. For property and method descriptions, see "TOvcVirtualListBox Component" on page 689.

Hierarchy

TCustomControl (VCL)

  TOvcCustomControl (OvcBase) 16

    TOvcCustomControlEx (OvcBase) 19

      TOvcCustomVirtualListBox (OvcVLB)

TOvcVirtualListBox Component

The TOvcVirtualListBox component is similar in appearance and behavior to the VCL's standard list box control, but with TOvcVirtualListBox you can:

- Manage a virtually unlimited number of items, using a very small amount of memory.
- Display rows using any combination of colors.
- Display a header line at the top of the list box.
- Scroll horizontally in the list box.
- Set up to 128 tab positions.
- Select multiple items.
- Protect list items.

The TOvcVirtualListBox can manage a virtually unlimited number of items because it does not store the items it displays. A special method is called when the TOvcVirtualListBox needs to update the display. This routine (an event handler) returns the display string corresponding to the requested row. You can even construct this string on demand in the event handler.

List box commands

The TOvcVirtualListBox component provides configurable key to command translation support through the TOvcController assigned to the Controller property. The supported commands are shown in Table 22.2.

Table 22.2: *List box commands*

Command	**Default**	**WordStar**	**Description**
ccDown	<Down>	<CtrlX>	Move the focus down to the next list item.
ccEnd	<End>	<CtrlQ><D>	Move the focus to the last item in the list.
ccExtendDown	<ShiftDown>	Not assigned.	Extend the selection down to the next list item.
ccExtendEnd	<ShiftEnd>	Not assigned.	Extend the selection to the last list item.

Table 22.2: *List box commands (continued)*

Command	Default	WordStar	Description
ccExtendHome	<ShiftHome>	Not assigned.	Extend the selection to the first list item.
ccExtendPgDn	<ShiftPgDn>	Not assigned.	Extend the selection down one page.
ccExtendPgUp	<ShiftPgUp>	Not assigned.	Extend the selection up one page.
ccExtendUp	<ShiftUp>	Not assigned.	Extend the selection up one list item.
ccHome	<Home>	<CtrlQ><S>	Move the focus to the top list item.
ccLeft	<Left>	<CtrlS>	Scroll the list items left one character position.
ccNextPage	<PgDn>	<CtrlC>	Scroll the list items down one page.
ccPrevPage	<PgUp>	<CtrlR>	Scroll the list items up one page.
ccRight	<Right>	<CtrlD>	Scroll the list items right one character position.
ccUp	<Up>	<CtrlE>	Move the focus up to the previous list item.

Windows messages

In addition to the properties, events, and methods listed later in this section, the virtual list box also responds to the following standard Windows list box messages:

```
LB_GETCARETINDEX
LB_GETCOUNT
LB_GETCURSEL
LB_GETITEMHEIGHT
LB_GETITEMRECT
LB_GETSEL
LB_GETTOPINDEX
LB_RESETCONTENT
LB_SELITEMRANGE
LB_SETCURSEL
LB_SETSEL
LB_SETTABSTOPS
LB_SETTOPINDEX
```

Refer to the Windows API reference or on-line help for descriptions and instructions on how to use these messages.

Example

The following example shows how to use the TOvcVirtualListBox to display a list of pizza toppings.

Create a new project, add components, and set the property values as indicated in Table 22.3.

Table 22.3: *TOvcVirtualListBox example*

Component	**Property**	**Value**
OvcVirtualListbox	Header	Pizza Toppings
	NumItems	11
	Scrollbars	ssBoth

When the TOvcVirtualListBox component is added to the form, a default TOvcController non-visual component is automatically added and the component's Controller property is changed to reflect the newly added TOvcController's name. This happens only if the form does not already contain one or more TOvcController components.

Create an event handler for the OnGetItem event to look like the following:

```
procedure
  TForm1.OvcVirtualListbox1GetItem(
    Sender : TObject; Index : LongInt;
    var ItemString : string);
begin
  case Index of
    0 : ItemString := 'Ham';
    1 : ItemString := 'Pepperoni';
    2 : ItemString := 'Sausage';
    3 : ItemString := 'Mushrooms';
    4 : ItemString := 'Anchovies';
    5 : ItemString := 'Onions';
    6 : ItemString := 'Olives';
    7 : ItemString := 'Eggs';
    8 : ItemString := 'Pineapple';
    9 : ItemString := 'Spices';
   10 : ItemString := 'Artichoke';
  end;
end;
```

Run the project and experiment with the generated program, possibly adding event handlers for colors and protected items. For example, to display the “Onions” list item but avoid having it selected, add the following event handler for the OnGetItemStatus event:

```
procedure
  TForm1.OvcVirtualListbox1GetItemStatus(
    Sender : TObject; Index : LongInt; var Protect : Boolean);
begin
  if Index = 5 then
    Protect := True
  else
    Protect := False;
end;
```

To allow multiple selections, you must add an event handler for the OnIsSelected event. It should return True if the list item is selected, otherwise it returns False. You also need to provide an event handler for the OnSelect event to update the selection state of each list box item.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase). 16

❷ TOvcCustomControlEx (OvcBase) 19

TOvcCustomVirtualListbox (OvcVLB)

TOvcVirtualListBox (OvcVLB)

Properties

❶	About		IntegralHeight		ScrollBars
❶	AttachedLabel		ItemIndex		SelectColor
	AutoRowHeight	❶	LabelInfo		ShowHeader
	BorderStyle		MultiSelect		SmoothScroll
	Columns		NumItems		TopIndex
❷	Controller		OwnerDraw		UseTabStops
	Header		ProtectColor		WheelDelta
	HeaderColor		RowHeight		

Methods

BeginUpdate	EndUpdate	Scroll
DeleteItemsAt	InsertItemsAt	SelectAll
DeselectAll	InvalidateItem	SetBounds
DrawItem	ItemAtPos	SetTabStops

Events

❶	AfterEnter		OnGetItem		OnSelect
❶	AfterExit		OnGetItemColor		OnTopIndexChanged
	OnCharToItem		OnGetItemStatus		OnUserCommand
	OnClickHeader		OnIsSelected		
	OnDrawItem	❶	OnMouseWheel		

Reference Section

AutoRowHeight — property

```
property AutoRowHeight : Boolean
```

Default: True

Determines whether cell heights are adjusted automatically.

If AutoRowHeight is True, changes in the Font are detected and the height of the list box rows is adjusted to be the same height as the selected font. If AutoRowHeight is False, no automatic adjustments are made.

If you change the RowHeight property, AutoRowHeight is set to False.

See also: RowHeight

BeginUpdate — method

```
procedure BeginUpdate;
```

Suspends painting until EndUpdate is called.

This method should be used in situations where the NumItems property is continually being changed.

You should normally use a try-finally block to insure that the corresponding EndUpdate method gets called:

```
    BeginUpdate;
    try
      {increase/decrease elements in related data structure}
      {NumItems = NumItems +/- 1}
    finally
      EndUpdate;
    end;
```

BorderStyle — property

```
property BorderStyle : TBorderStyle
```

Default: bsSingle

The style used when drawing the list box border.

The possible values are bsSingle (a single line border is drawn around the component) or bsNone (no border line).

The TBorderStyle type is defined in the VCL’s Controls unit.

Columns property

```
property Columns : Byte
```

Default: 255

↳ The logical width, in characters, of the list box.

Columns is used to set the horizontal scrolling limits. If you use a value less than the longest string displayed in the list box, the horizontal scrolling distance is limited and part of some list box items will not be visible. A value that is too large will allow horizontal scrolling beyond the end of the displayed list items.

See also: ScrollBars

DeleteItemsAt method

```
procedure DeleteItemsAt(Items : LongInt; Index : LongInt);
```

↳ Removes Items number of items starting at the Index.

See also: BeginUpdate

DeselectAll method

```
procedure DeselectAll;
```

↳ Removes the selection from all items in the list.

DrawItem method

```
procedure DrawItem(Index : LongInt);
```

↳ Called to force the list element specified by Index to be drawn.

DrawItem does this by first invalidating the requested item and then calling Update for the area to be painted.

EndUpdate method

```
procedure EndUpdate;
```

↳ Resumes painting that was paused due to a call to the BeginUpdate method.

Header property

```
property Header : string
```

↳ The text displayed at the top of the list box.

The string assigned to the Header property is displayed at the top of the list box. If the list box contains a horizontal scrollbar, the header is scrolled right and left along with the list box items.

Entering text for this property automatically enables the header display and sets the ShowHeader property to True.

See also: ShowHeader

HeaderColor property

```
property HeaderColor : TOvcColors
```

Default: clBtnText and clBtnFace

↳ Determines the colors used to display the header text and background.

The default color choices of clBtnText for the text color and clBtnFace for the background cause the header to be displayed using the standard button look.

See “TOvcColors Class” on page 1152 for more information.

See also: Header, ShowHeader

InsertItemsAt method

```
procedure InsertItemsAt(Items : LongInt; Index : LongInt);
```

↳ Increases NumItems with Items amount while scrolling the window down from Index.

IntegralHeight property

```
property IntegralHeight : Boolean
```

Default: True

↳ Indicates whether the list box automatically resizes itself to display only complete list items.

If IntegralHeight is True, the list box is resized so that partial rows are not displayed at the bottom of the list box. If IntegralHeight is False, partial rows might be displayed.

InvalidateItem method

```
procedure InvalidateItem(Index : LongInt);
```

↳ Called to invalidate the area occupied by the list element specified by Index.

ItemAtPos method

```
procedure ItemAtPos(Pos : TPoint; Existing : Boolean) : LongInt;
```

↳ Returns the index number of the list box item containing the point Pos.

The ItemAtPos method returns the index of the list box indicated by the coordinates of a point on the control. The Pos parameter is the point in the control in window coordinates. If Pos is beyond the last item or before the first item in the list box, the value of the Existing parameter determines the returned value. If you set Existing to True, ItemAtPos returns -1, indicating that no item exists at that point. If you set Existing to False, ItemAtPos returns the position of the last or first item in the list box. ItemAtPos is useful for detecting if an item exists at a particular point in the list box.

ItemIndex run-time property

```
property ItemIndex : LongInt
```

↳ Determines the currently selected list box item.

ItemIndex contains -1 if no item is selected. ItemIndex contains the index of the list box item that was selected last (i.e., the one that has the focus). Assigning a valid index value to ItemIndex causes that list box item to be selected. The list box is scrolled as necessary to make the newly selected item visible.

The valid range for ItemIndex is 0 through NumItems-1.

See also: MultiSelect, NumItems, TopIndex

MultiSelect property

```
property MultiSelect : Boolean
```

Default: False

↳ Indicates whether multiple items can be selected.

If MultiSelect is True, one or more list box items can be selected using the same techniques as for standard TListBox components. The OnIsSelected event must be assigned to a method that supplies the selected status of all items in the list box. The OnSelect event must also be assigned to maintain the selected status of all items in the list box.

If these events are not assigned and MultiSelect is True, an EOnSelectNotAssigned and/or EOnIsSelectedNotAssigned exception will be generated at run time.

If MultiSelect is False, only one item can be selected at a time. MultiSelect can only be changed at design time.

See also: OnIsSelected, OnSelect

NumItems property

```
property NumItems : LongInt
```

Default: Depends on the font and height of the list box

↳ The actual number of items that the virtual list box can process.

Set this property to the maximum number of items that the list box will display. The virtual list box element indexes are zero based, therefore assigning a value of 10 to NumItems yields a valid index range of 0 to 9.

OnCharToItem event

```
property OnCharToItem : TCharToItemEvent
```

```
TCharToItemEvent = procedure(
  Sender : TObject; Ch : AnsiChar; var Index : LongInt) of object;
```

↳ Defines an event handler that is called to provide character-to-index mapping.

The method assigned to OnCharToItem is called to provide notification of a key press and can optionally return a new item index. Ch contains the character that was entered and Index contains the index of the currently active list item. If you assign Index a value other than the currently selected index, it becomes the new active item.

The following example shows how to allow a user to enter a character, map the character to a list box index, and return the results. If the list box is displaying the letters of the alphabet and you press F, the list box displays the 6th item (list index number 5).

```
procedure
  TForm1.OvcVirtualListbox1CharToItem(
    Sender : TObject; Ch : AnsiChar; var Index : LongInt);
var
  I : LongInt;
begin
  {using the character in Ch, select the corresponding index}
  {that you wish selected in the list box}
  I := Ord('A') - Ord(Ch);
  if I >= 0 then
    Index := I;
  {simple mapping assumes 26 items in the list box}
end;
...
OnCharToItem := Form1.OvcVirtualListBox1CharToItem;
```

OnClickHeader event

```
property OnClickHeader : THeaderClickEvent
```

```
THeaderClickEvent = procedure(
  Sender : TObject; Point : TPoint) of object;
```

Defines an event handler that is called when a mouse click occurs in the header area.

The method assigned to OnClickHeader is called to provide notification of a mouse click in the header area. Point contains the X and Y (screen-relative) position of the mouse.

The following example shows how to detect a mouse click in the header area of the list box. In this example, if the mouse is clicked in the left half of the header one action is taken and if the mouse is clicked in the right half of the header another action is taken.

```
procedure TForm1.OvcVirtualListBox1ClickHeader(
  Sender : TObject; Point : TPoint);
var
  Pt : TPoint;
begin
  Pt := OvcVirtualListBox1.ScreenToClient(Point);
  if Pt.X < OvcVirtualListBox1.Width div 2 then
    {do something}
  else
    {do something else};
end;
```

OnDrawItem event

```
property OnDrawItem : TDrawItemEvent
```

```
TDrawItemEvent = procedure(
  Sender : TObject; Index : LongInt;
  Rect : TRect; const S : string) of object;
```

Defines an event handler that is called when the cell requires painting.

The method assigned to the OnDrawItem event is called to paint the cell defined by Rect. Index is the index number of the cell that requires painting and S is the text of the cell. An event handler assigned to this event will not be called unless the OwnerDraw property is set to True.

See also: OwnerDraw

OnGetItem event

```
property OnGetItem : TGetItemEvent
```

```
TGetItemEvent = procedure(Sender : TObject; Index : LongInt;
  var ItemString : string) of object;
```

Defines an event handler that is called to obtain the display string for the virtual list box item.

ItemString is empty on entry. The event handler must assign the display string corresponding to Index to ItemString. If no method is assigned to this event, the list box displays sample data in all rows. Therefore, you can't do much with the virtual list box until you write an event handler for this event.

The following example causes the list box to display ten items. The text in the rows is "Age = 2", "Age = 5", etc:

```
NumItems := 10

const
  Ages : array[0..9] of Byte = (2,5,7,8,12,44,66,33,17,21);

procedure
  TForm1.OvcVirtualListBox1GetItem(
    Sender : TObject; Index : LongInt;
    var ItemString : OpenString);
begin
  ItemString := Format('Age = %d', [Ages[Index]]);
end;
```

OnGetItemColor event

```
property OnGetItemColor : TGetItemColorEvent
```

```
TGetItemColorEvent = procedure(Sender : TObject;
  Index : LongInt; var FG, BG : TColor) of object;
```

↳ Defines an event handler that is called to return the colors used to draw the list box elements.

The method assigned to OnGetItemColor is called to get the colors to use when drawing the list box elements. On entry, FG and BG contain the default foreground and background colors and Index is the index of the item to be drawn. Set FG and BG to the desired colors for the array element specified by Index.

The following example causes the virtual list box to display the text of every other array element in red:

```
  procedure
    Form1.OvcVirtualListBox1GetItemColor(
      Sender : TObject; Index : LongInt; var FG, BG : TColor);
  begin
    if Odd(Index) then
      FG := clRed;
  end;
```

OnGetItemStatus event

```
property OnGetItemStatus : TGetItemStatusEvent
```

```
TGetItemStatusEvent = procedure(Sender : TObject;
  Index : LongInt; var Protect : Boolean) of object;
```

↳ Defines an event handler that is called to determine the protected status of the list box items.

The method assigned to OnGetItemStatus is called to determine if the list box item specified by Index is protected. A protected list box item cannot be selected and is displayed using the colors specified by the ProtectColor property. If no event handler is assigned, no items can be protected.

Set Protect to True to protect the item specified by Index. Set Protect to False to unprotect the item specified by Index.

See also: ProtectColor

OnIsSelected event

```
property OnIsSelected : TIsSelectedEvent
```

```
TIsSelectedEvent = procedure(Sender : TObject;
  Index : LongInt; var Selected : Boolean) of object;
```

Defines an event handler that is called to determine the item selection status.

If multiple selection is enabled (MultiSelect is True) and a method is assigned to OnIsSelected, the virtual list box calls the assigned method to get the selection status of the element specified by Index. If the method sets Selected to True, the item will then be displayed using the colors specified by the SelectColor property.

The following example causes the virtual list box to display every other list element as selected:

```
  var
    I : Integer;
    SelectStatus : array[0..99] of Boolean;
  ...
  NumItems := 100;
  for I := 0 to 99 do
    SelectStatus[I] := Odd(I);
  ...
  procedure
    TForm1.OvcVirtualListbox1IsSelected(
      Sender : TObject; Index : LongInt;
      var Selected : Boolean);
  begin
    Selected := SelectStatus[Index];
  end;
```

See also: MultiSelect, SelectColor

OnSelect event

```
property OnSelect : TSelectEvent
```

```
TSelectEvent = procedure(Sender : TObject;
  Index : LongInt; Selected : Boolean) of object;
```

Defines an event handler that is called to provide notification of a selection change.

Index contains the list box item index. Selected is True if the item corresponding to Index is selected and False if the item was selected and is being unselected. An Index value of -1 indicates that all items in the list should either be selected or deselected depending on the value of the Selected parameter.

Caution: OnSelect is only fired if the MultiSelect property is set to True. To detect a selection change with MultiSelect set to False, use the OnClick event.

The following example causes the computer to generate a single beep when a new item is selected and two beeps when an item is unselected:

```
procedure TForm1.OvcVirtualListbox1Select(
  Sender : TObject; Index : LongInt; Selected : Boolean);
begin
  if Selected then
    MessageBeep(0)
  else begin
    MessageBeep(0);
    MessageBeep(0)
  end
end;
```

OnTopIndexChanged event

```
property OnTopIndexChanged : TTopIndexChanged
```

```
TTopIndexChanged = procedure(
  Sender : TObject; NewTopIndex : LongInt) of object;
```

Defines an event handler that is called when the top list item changes.

The method assigned to OnTopIndexChanged is called to provide notification that the item displayed at the top of the list box changed. NewTopIndex is the index of the new list item displayed at the top of the list box.

OnUserCommand event

```
property OnUserCommand : TUserCommandEvent
```

```
TUserCommandEvent = procedure(
  Sender : TObject; Command : Word) of object;
```

Defines an event handler that is called when a user-defined command is entered.

The method assigned to the OnUserCommand event is called when the key sequence corresponding to one of the user-defined commands (ccUser1, ccUser2, etc.) is entered.

For example, suppose you add the <CtrlD> key sequence to one of the active command tables and assign it to the ccUser1 command. When <CtrlD> is pressed, the method assigned to the OnUserCommand event is called and ccUser1 is passed in Command.

See “TOvcCommandProcessor Class” on page 50 for additional information about user-defined commands and command tables in general.

OwnerDraw property

```
property OwnerDraw : Boolean
```

↳ Determines if the OnDrawItem event handler is called.

If OwnerDraw is True and an event handler is assigned to the OnDrawItem event, that event handler is called when the cells of the virtual list box require painting.

See also: OnDrawItem

ProtectColor property

```
property ProtectColor : TOvcColors
```

Default: clWhite and clRed

↳ Determines the colors used to display protected list box items.

The default color choices of clWhite and clRed cause protected items to be displayed as white text on a red background.

See “TOvcColors Class” on page 1152 for more information.

See also: OnGetItemStatus

RowHeight property

```
property RowHeight : Integer
```

Default: Depends on the currently selected font

↳ The height of each list box cell.

Assigning a value to this property changes the height of all list box cells and also sets the AutoRowHeight property to False. If the IntegralHeight property is True, the overall height of the list box is adjusted so that a partial row is not displayed.

See also: AutoRowHeight

Scroll method

```
procedure Scroll(HDelta, VDelta : Integer);
```

↳ Performs horizontal and vertical scrolling.

Scroll is provided to allow you to do horizontal and vertical scrolling programmatically. HDelta is the signed horizontal distance to scroll and VDelta is the signed vertical distance to scroll. If either of the parameters is beyond the valid scroll range, the list is scrolled to the appropriate scroll limit.

ScrollBars property

```
property ScrollBars : TScrollStyle
```

Default: ssVertical

↳ Indicates whether vertical and horizontal scrollbars are displayed.

A vertical scroll bar is displayed by default when needed. The TScrollStyle enumerated type is defined in the VCL's Controls unit.

See also: Columns

SelectAll method

```
procedure SelectAll;
```

↳ Selects all items in the list box.

SelectColor property

```
property SelectColor : TOvcColors
```

Default: clHighlightText and clHighlight

↳ Determines the colors used to display selected list box items.

The colors specified are used to display list box items that are selected, but not protected.

See "TOvcColors Class" on page 1152 for more information.

SetTabStops method

```
procedure SetTabStops(Tabs : array of Integer);
```

↳ Assigns tab positions for use in a list box.

Tabs is an array of integers representing the tab positions. The tab positions are specified in dialog units. A dialog unit is one fourth of the average character width for the current font.

Set one tab stop for each tab character embedded in the display string. Additional embedded tab characters and unused tab stop array values are ignored. You can define up to 128 (0-127) tab stop settings.

Don't confuse the tab stop handling of the virtual list box with the columns feature of the standard VCL list box. Embedding tab characters in your display strings does not establish columns in the list. When the virtual list box sees the tab character and tab expansion is enabled, the tab character is removed and the next character is printed at the offset specified in the tab array. If the tab character has scrolled out of the list box's visible region, it is not expanded.

The following example sets tab stop positions at 100, 200, and 300 dialog units. Items in the list box that contain tab characters are displayed using these three tab positions.

```
var
  TabArray : array[0..2] of Integer;
...
TabArray[0] := 100;
TabArray[1] := 200;
TabArray[2] := 300;
OvcVirtualListbox1.SetTabStops(TabArray);
```

See also: UseTabStops

ShowHeader property

```
property ShowHeader : Boolean
```

Default: False

Determines whether the header is displayed.

If ShowHeader is True, the text assigned to the Header property is displayed at the top of the list box. If it is False, no header is displayed.

See also: Header, HeaderColor

SmoothScroll property

```
property SmoothScroll : Boolean
```

Default: True

Determines the granularity used when scrolling the window area of the virtual list box.

Setting SmoothScroll to True causes the scroll logic to use a finer granularity for scrolling operations, resulting in a smoother scroll rate, but taking more time.

TopIndex run-time property

```
property TopIndex : LongInt
```

The index of the item displayed in the first row of the list box.

Assigning a value to TopIndex displays that item at the top of the list box, if possible. Index values that are out of range are ignored.

See also: ItemIndex, NumItems

UseTabStops property

```
property UseTabStops : Boolean
```

Default: False

↳ Indicates whether embedded tab characters are expanded.

If UseTabStops is True, embedded tab characters in the list box display strings (and the header, if it is enabled) are expanded to correspond to the current tab settings as determined by the last call to SetTabStops. If no tab stops are assigned, embedded tab characters are not expanded.

See also: SetTabStops

WheelDelta property

```
property WheelDelta : Integer
```

Default: 3

↳ Determines the number of lines to scroll when the user turns the mouse wheel for mice that have wheels.

Chapter 23: Meters

The Orpheus meter components display progress and status of events in your applications. The look and behavior of each of the meter components is extremely customizable. For example, the PeakMeter can be configured to present a scrolling historical display indicating the status of a particular process (much like the CPU and Memory usage indicator you see in the Windows NT Task Manager).

Each meter can display background images to enhance their visual appeal.

The ExMeter example program demonstrates several possible uses for the meter components.

- The TOvcMeter component displays a simple progress bar that represents the state of a current process of an operation.
- TOvcPeakMeterdisplays a graphical representation of some value of an operation. You can choose from a bar graph or a line graph with variable colors and borders.

TOvcCustomMeter Class

The TOvcCustomMeter class is the immediate ancestor of the TOvcMeter component. It implements all the methods and properties used by the TOvcMeter component and is identical to the TOvcMeter component except that no properties are published.

TOvcCustomMeter is provided to facilitate creation of descendent meter components. For property and method descriptions, see “TOvcMeter Component” on page 711.

Hierarchy

TGraphicControl (VCL)

    TOvcGraphicControl (OvcBase) 82

        TOvcCustomMeter (OvcMeter)

TOvcMeter Component

The TOvcMeter component is a horizontal or vertical progress indicator. Meter components are normally displayed during processes that cannot be interrupted and where some form of progress feedback is appropriate.

For operations that take a very small amount of time to complete, displaying the hourglass cursor is sufficient. Longer delays should display some other type of visual feedback at regular intervals so that the user does not think that the application has stopped.

The TOvcMeter component fulfills this requirement by providing a simple progress bar (shown in Figure 23.1) that can be displayed in different sizes to correspond with the state of current process.

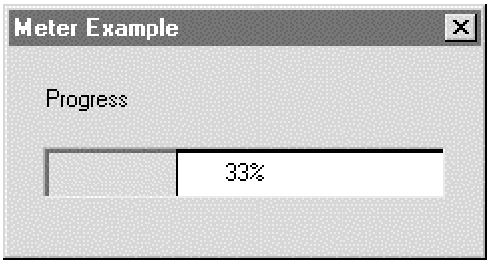


Figure 23.1: Meter example

The TOvcMeter gives you some nice additional features too. You can optionally display the percentage as a number at the center of the component. You can also configure the color for both sections (left and right or top and bottom), the shadow for both sections, and the color of the line dividing the sections.

Example

This example creates a form that displays two TOvcMeter components. The position of each meter component's progress bar is controlled by a VCL timer component.

Create a new project, add components, and set the property values as indicated in Table 23.1.

Table 23.1:

Component	Property	Value
TOvcMeter	Orientation	moHorizontal
	ShowPercent	True
	UsedColor	clRed
	UsedShadow	clYellow
TOvcMeter	BorderStyle	bsSingle
	Orientation	moVertical
	ShowPercent	True
	UsedColor	clTeal
TTimer	Interval	50

Create an OnTimer event handler for the timer component similar to the following:

```
procedure TForm1.Timer1Timer(Sender : TObject);
begin
  {increase meter 1's display value}
  OvcMeter1.Percent := OvcMeter1.Percent + 1;
  if OvcMeter1.Percent = 100 then
    OvcMeter1.Percent := 0;

{decrease meter 2's display value}
  OvcMeter2.Percent := OvcMeter2.Percent - 1;
  if OvcMeter2.Percent = 0 then
    OvcMeter2.Percent := 100;
end;
```

When this project is compiled and run, the first meter’s progress bar will increase in size while the second bar decreases.

Hierarchy

TGraphicControl (VCL)

❶ TOvcGraphicControl (OvcBase) 82

TOvcCustomMeter (OvcMeter) 710

TOvcMeter (OvcMeter)

Properties

❶	About	Percent	UnusedImage
	InvertPercent	ShowPercent	UsedColor
	Orientation	UnusedColor	UsedImage

Reference Section

InvertPercent property

```
property InvertPercent : Boolean
```

Default: True

Determines whether the text with the Percent value should be drawn in colors that are the inverse of the meter color.

Enabling this property makes the percent value text easier to read. If the percentage value is a bitmap, readability may be improved by setting this property to False which will cause the percent value to be displayed using a single color.

See also: Percent

Orientation property

```
property Orientation : TMeterOrientation
```

```
TMeterOrientation = (moHorizontal, moVertical);
```

Default: moHorizontal

Determines whether the meter is horizontal or vertical.

If Orientation is moHorizontal, the meter component is drawn so that the progress bar moves from left to right for increasing values of Percent. If Orientation is moVertical, the meter component is drawn so that the progress bar moves from bottom to top for increasing values of Percent.

See also: Percent

Percent property

```
property Percent : Byte
```

Default: 33

Determines how much of the progress bar is drawn.

Larger values for Percent cause more of the progress bar to display. Valid values for Percent are from 0 to 100. If Percent equals 0, no progress bar is displayed. If Percent equals 100, a complete progress bar is displayed.

The default value (33) has no special meaning. It is used to allow you to view all areas of the meter at design time. You will probably initialize Percent to 0 or 100 at run time.

ShowPercent property

```
property ShowPercent : Boolean
```

Default: False

Determines whether the Percent value is displayed.

This property determines whether the Percent value is displayed at the center of the meter control.

See also: Percent

UnusedColor property

```
property UnusedColor : TColor
```

Default: clWindow

Is the color used to draw the unused portion of the progress bar.

UnusedImage property

```
property UnusedImage : TBitmap
```

Specifies a bitmap to be used as the meter background for the unused portion of the meter.

When an UnusedImage is specified for the meter surface, the UnusedColor property of the meter is ignored. UnusedImage is typically used in combination with UsedImage.

See also: UnusedColor, UsedImage

UsedColor property

```
property UsedColor : TColor
```

Default: clLime

Is the color used to draw the used portion of the progress bar.

UsedImage property

```
property UsedImage : TBitmap
```

Specifies a bitmap to be used as the meter background for the used portion of the meter.

When a UsedImage is specified for the meter surface, the value of the UsedColor property of the meter is ignored. UsedImage is typically used in combination with UnusedImage.

See also: UsedColor, UnusedImage

TOvcPeakMeter Component

The Peak Meter component shows a graphical representation of the current value of some measurement defined by the application, and the highest value during the session. If the history point style is selected, the control displays the most recent values as a graph. The graph scrolls as new sampling values are assigned to the meter.

Hierarchy

TGraphicControl (VCL)

❶ TOvcGraphicControl (OvcBase) 82

TOvcPeakMeter (OvcPeakM)

Properties

❶	About	MarginLeft	ShowValues
	BackgroundColor	MarginRight	Style
	BarColor	MarginTop	Value
	GridColor	Peak	
	MarginBottom	PeakColor	

Reference Section

BackgroundColor — property

```
property BackgroundColor : TColor
```

Default: clBtnFace

↳ Defines the color used when painting the Peak Meter.

BarColor — property

```
property BarColor : TColor
```

Default: clBlue

↳ Specifies the color used to draw the bar or line, which shows the meter values.

GridColor — property

```
property GridColor : TColor
```

↳ Specifies the color used to draw the axis and the grid.

MarginBottom — property

```
property MarginBottom : Integer
```

Default: 10

↳ Specifies the number of pixels from the bottom of the PeakMeter control to where the actual meter is painted.

MarginLeft — property

```
property MarginLeft : Integer
```

Default: 10

↳ Specifies the number of pixels between the axis and the PeakMeter bar.

This property has no effect unless the PeakMeter is using the bar style.

See also: Style

MarginRight property

```
property MarginRight : Integer
```

Default: 10

↳ Specifies the number of pixels between the right side of the meter control and the PeakMeter bar.

This property has no effect unless the PeakMeter is using the bar style.

See also: Style

MarginTop property

```
property MarginTop : Integer
```

Default: 10

↳ Specifies the number of pixels from the top of the PeakMeter control to where the actual meter is painted.

Peak run-time property

```
property Peak : Integer
```

↳ Determines the current peak value for the meter.

You can reset the current peak value by assigning a different value to this property.

See also: Value

PeakColor property

```
property PeakColor : TColor
```

Default: clRed

↳ Specifies the color used to draw the peak indicator.

ShowValues property

```
property ShowValues : Boolean
```

Default: True

↳ Determines whether the PeakMeter draws a value axis.

Style property

```
property Style : TOvcPmStyle
```

```
TOvcPmStyle = (pmBar, pmHistoryPoint)
```

Default : pmBar

Determines the drawing style of the PeakMeter.

With the pmBar style, the PeakMeter is drawn as a single bar showing the current value. When pmHistoryPoint is used, the PeakMeter draws a line graph of recent values up to the current value.

See also: Style, Value

Value property

```
property Value : Integer
```

Assigns the value displayed in the meter.

With the Style set to pmHistory, assigning a new value to Value will cause the meter to scroll one pixel to the left and plot a point for the new value. This happens even if the new value assigned to Value is identical to the previous value.

See also: Style

Chapter 24: Spinner Component

TOvcSpinner implements a spinner control. It allows you to offer an alternative to normal data entry by allowing the user to adjust an associated control's value by clicking on the spin buttons.

To use a spin component, you provide an OnClick event handler that responds to the user clicking on one the spin buttons. However, if a TOvcSpinner has its FocusedControl property set to one of the Orpheus entry field components, you do not need to write an event handler. The spin component is "aware" of Orpheus entry fields and directly calls the necessary methods to increase and decrease field values or position the caret of entry fields editing any of the numeric, date, or time data types.

The spinner component connects or "docks" with another component and informs the component about the state of the spinner buttons.

Spinners have the capability to repeat when the spin button is held down. The auto-repeat capability is optional and the acceleration (how fast the speed of the notification messages increases) is adjustable. The Orpheus spinner components additionally support the ability to switch spin buttons while one is depressed by dragging the mouse cursor over another spin button. They use scaleable button designs that allow them to be used with fields of any reasonable height.

Eight different spinner styles are provided. The Style property determines the type of spinner to display. See the Style property in the reference section for a description and graphical depiction of each of the spinner styles.

TOvcSpinner Component

For a normal spin button (one with an up arrow button and a down arrow button), the up button usually increases the associated component's value, and the down arrow decreases it. For a horizontal spinner, the right button typically moves the caret to the right in an edit control, and the left button moves the caret to the left.

To dock a spinner with another component, right-click on the spinner at design time and choose "Dock With" from the context menu. The Dock Component dialog box is displayed showing a list of components on the form. Select a component to dock with, and the relative position the spinner should occupy (right, left, top, or bottom). When you click the OK button, the spinner will be positioned relative to the associated component according to the selections made. If you move the associated component you will have to re-dock the spinner.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase). 16

TOvcSpinner (OvcSC)

Properties

❶	About		Delta	RepeatCount
	Acceleration		DelayTime	ShowArrows
❶	AttachedLabel		FocusedControl	Style
	AutoRepeat	❶	LabelInfo	WrapMode

Events

❶	AfterEnter		OnClick
❶	AfterExit	❶	OnMouseWheel

Reference Section

Acceleration — property

```
property Acceleration : Integer
```

Default: 5

Determines the speed at which the associated control's value changes when a spinner button is held down.

Acceleration can range from 0 (no acceleration) to 10 (fastest acceleration). Acceleration is only valid if the AutoRepeat property is True.

See also: AutoRepeat, FocusedControl

AutoRepeat — property

```
property AutoRepeat : Boolean
```

Default: True

Controls how the spinner acts when a spinner button is held down.

When AutoRepeat is True, the value of the associated control will continue to change as long as a spinner button is held down. The amount by which the control's value changes is determined by the Acceleration property.

See also: Acceleration, DelayTime

DelayTime — property

```
property DelayTime : LongInt
```

Default: 500

The number of milliseconds of delay before auto repeating occurs.

When a spinner button is held down and AutoRepeat is True, the associated control's value will automatically increment as long as the button is down. DelayTime is the period of time, in milliseconds, before auto repeat starts.

See also: Acceleration, AutoRepeat

Delta property

```
property Delta : Double
```

Default: 1

↳ The value by which the associated control's value will change each time a spinner button is clicked.

When a spinner button is clicked, the associated control's value will change by the value of Delta. Let's say, for example, you have a TOvcNumberEdit as the associated control. If the control's value is initially 0 then when the spinner's up button is clicked the edit's value will change to 1, 2, 3, 4, and so on. If Delta is 5, the value of the number edit will change to 5, 10, 15, 20, etc. on each spinner up button click. Delta is also passed to the OnClick event handler.

See also: FocusedControl, OnClick

FocusedControl property

```
property FocusedControl : TWinControl
```

↳ The control associated with the spinner.

A spinner is typically associated with another control on the form. When you click a spinner button, the value of the associated control changes according to the spinner button clicked and the value of the Delta property. Set FocusedControl to the component that you want to be associated with the spinner. While TOvcSpinner is intended to be used with the Orpheus editor components, it can usually be used with any component derived from TCustomEdit.

See also: Delta

OnClick event

```
property OnClick : TSpinClickEvent
```

```
TSpinClickEvent = procedure(
  Sender : TObject; State : TOvcSpinState;
  Delta : Double; Wrap : Boolean) of object;
```

```
TOvcSpinState = (
  ssNone, ssNormal, ssUpBtn, ssDownBtn,
  ssLeftBtn, ssRightBtn, ssCenterBtn);
```

Defines an event handler that is generated when a spinner button is clicked.

Provide an OnClick handler if you are using the spinner without an associated control, or if you wish to perform special processing when a spinner button is clicked. Delta is the value of the Delta property. Wrap is the value of the spinner's Wrap property. State is the spinner button that was clicked. The possible values for State depend on the spinner's style (set by the Style property). A default up-down spinner, for example, will only generate ssUpBtn and ssDownBtn states. A star spinner with a center button may generate the ssCenterBtn state. The ssNone and ssNormal spin states are for TOvcSpinner's internal use and are not passed on in the OnClick event. The following table lists the possible values of State and a description of each:

Value	**Meaning**
ssUpBtn	The up spin button was pressed.
ssDownBtn	The down spin button was pressed.
ssLeftBtn	The left spin button was pressed.
ssRightBtn	The right spin button was pressed.
ssCenterBtn	The center button was pressed (stStar spinner style only).

The following example illustrates using a TOvcSpinner to modify the value of a TOvcMeter component:

```
procedure
  TForm1.OvcSpinner1Click(
    Sender: TObject; State: TOvcSpinState; Delta: Double;
    Wrap: Boolean);
begin
  if State = ssUpBtn then begin
    OvcMeter1.Percent := OvcMeter1.Percent + Round(Delta);
    if Wrap then
      if OvcMeter1.Percent > 100 then
        OvcMeter1.Percent := 0;
  end;
  if State = ssDownBtn then begin
    OvcMeter1.Percent := OvcMeter1.Percent - Round(Delta);
    if Wrap then
      if OvcMeter1.Percent < 0 then
        OvcMeter1.Percent := 100;
  end;
end;
```

See also: Delta, WrapMode

RepeatCount run-time property

```
property RepeatCount : LongInt
```

↳ The number of changes to the associated control when auto repeating.

Read RepeatCount to determine the number of times the associated control changed when auto repeat is on.

ShowArrows property

```
property ShowArrows : Boolean
```

↳ Determines whether or not arrows are drawn on the spinner buttons.

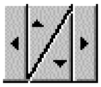
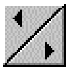
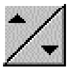
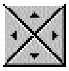
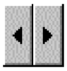
```
property Style : TOvcSpinnerStyle
```

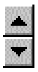
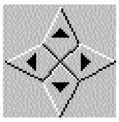
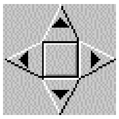
```
TOvcSpinnerStyle = (stNormalVertical, stNormalHorizontal,
  stFourWay, stStar, stDiagonalVertical,
  stDiagonalHorizontal,stDiagonalFourWay, stPlainStar);
```

Default: stNormalVertical

The spinner button style.

The following is a list of the spinner button styles and the appearance of each:

`stDiagonalHorizontal`	
`stDiagonalVertical`	
`stDiagonalFourWay`	
`stFourWay`	
`stNormalHorizontal`	

`stNormalVertical`	
`stPlainStar`	
`stStar`	

WrapMode property

```
property WrapMode : Boolean
```

Determines if the associated control's value wraps when the upper or lower limit is reached.

For example, assume you are using a spinner to increase or decrease the value of a Byte field and Delta is 1. If WrapMode is True and the value is increased past 255, the field value will wrap to 0. Conversely, if the value is decreased below 0, the field value will wrap to 255.

See also: FocusedControl, Delta

Chapter 25: Splitter Component

The Orpheus TOvcSplitter allows you to split a window into two sections, either vertically or horizontally. Each section of the Splitter can contain other controls or even other splitters. If a Splitter section contains a client-aligned splitter, the border and splitter bar are drawn to meld into the parent splitter.

TOvcSplitter Component

The TOvcSplitter component provides two window sections that can be used to parent other controls. A center splitter bar allows the sections to be resized both at design time and run-time. The Orientation property determines if the region is split vertically (the default) or horizontally.

By embedding one splitter component within another, you can create the look of a control with more than two sections.

One special feature of the Orpheus splitter is that when one or more splitters are embedded within another, the edges and the splitter bar are drawn to look as if the combined controls are a single component.

Note: A component pasted into the splitter from the Windows clipboard will generally be placed in the first section of the splitter. To paste into the second section, you first must focus some other component in that section. If there are no components in the second section, drop a TLabel or some other component there and then perform the paste operation.

Hierarchy

TCustomControl (VCL)

  ❶ TOvcCustomControl (OvcBase). 16

    ❷ TOvcCustomControlEx (OvcBase) 19

      TOvcSplitter (OvcSplit)

Properties

- ❶ About
- AllowResize
- ❶ AttachedLabel
- AutoScale
- AutoUpdate
- ColorLeft
- ColorRight
- ❷ Controller
- ❶ LabelInfo
- Orientation
- Position
- Section
- Section1Info
- Section2Info
- SplitterColor
- SplitterSize

Methods

- Center

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❶ OnMouseWheel
- OnOwnerDraw
- OnResize

Reference Section

AllowResize property

```
property AllowResize : Boolean
```

Default: True

↳ Determines whether the center splitter bar can be used to resize the section at run time.

AutoScale property

```
property AutoScale : Boolean
```

↳ Determines if the splitter is automatically repositioned when the splitter's size changes.

If False, the position value will not change.

AutoUpdate property

```
property AutoUpdate : Boolean
```

Default: False

↳ Determines whether the sections are resized during the sizing action.

If AutoUpdate is True, the sections are resized during the sizing action. If AutoUpdate is False, the sections are resized after the mouse button is released.

Center method

```
procedure Center;
```

↳ Resets the position of the splitter bar so that it lies at the center of the splitter component's window.

ColorLeft property

```
property ColorLeft: TColor
```

↳ Provides access to the color used to paint section 1 (the left or top section).

ColorRight property

```
property ColorRight : TColor
```

↳ Provides access to the color used to paint section 2 (the right or bottom section).

Margin property

```
property Margin : Integer
```

Default: 0

Specifies a blank area around splitter sections.

The Margin property can be used when the splitter component is aligned to the client and you don't want it to occupy the complete area of the parent control.

OnOwnerDraw property

```
property OnOwnerDraw : TNotifyEvent
```

Defines an event handler that is called when the splitter requires painting.

If this event is assigned, all painting must be done by your application.

TNotifyEvent is defined in the VCL's Classes unit.

OnResize property

```
property OnResize : TNotifyEvent
```

Defines an event handler that is called when the user finishes sizing a section.

The OnResize event is generated when the mouse button is released.

TNotifyEvent is defined in the VCL's Classes unit.

Orientation property

```
property Orientation : TSplitterOrientation
TSplitterOrientation = (soVertical, soHorizontal);
```

Default: soVertical

Determines whether the splitter bar is drawn horizontally or vertically.

Position property

```
property Position : Integer
```

Default: 100

Determines the distance in pixels from the left/top of the control to the left/top edge of the splitter bar.

Section — read-only property

```
property Section[Index : Integer] : TOvcSection
```

↳ Provides read-only access to the TOvcSection object associated with either of the splitter sections.

Valid values for Index are 0 or 1.

SplitterColor — property

```
property SplitterColor : TColor
```

Default: clWindowText

↳ Determines the color used to draw the center bar.

If Ctl3D is False, SplitterColor is the color used to draw the center bar. If Ctl3D is True, SplitterColor is ignored and clBtnFace is used.

SplitterSize — property

```
property SplitterSize : Integer
```

Default: 3

↳ Determines the width or height (in pixels) of the splitter bar.

Chapter 26: Labels

The label components provide capabilities similar to the standard VCL TLabel component but add special enhanced capabilities to make them well suited for use in applications that use other Orpheus components. For example, the TOvcPictureLabel implements the PictureMask capabilities of the Orpheus TOvcPictureField; the attached label is a special type of label that attaches to most of the Orpheus components.

This section describes the following label components:

- TOvcLabel lets you create eye-catching captions for your applications. It expands on the VCL TLabel with special shading, color, and highlight capabilities.
- TOvcRotatedLabel allows the text to be rotated to any viewing angle.
- TOvcPictureLabel is a descendant of TOvcRotatedLabel that supports displaying data formatted by a picture mask.
- TOvcDBPictureLabel is a data-aware version of the TOvcPictureLabel that connects to a data source and allows viewing of a database field.
- TOvcDBDisplayLabel is a data-aware component that connects to a data source and displays the DisplayLabel property of a given database field.
- TOvcURLLabel is a label component that emulates a hyperlink on a Web page.
- TOvcAttachedLabel is a descendant of the standard TLabel with a special twist—it can be associated with another control and then moves when the other control is moved.
- TOvcLabelInfo implements a persistent storage class used to store properties that pertain to the attached label.

TOvcLabel Component

The TOvcLabel lets you create eye-catching captions for your applications. It expands on the VCL TLabel with special shading, color, and highlight capabilities.

The Orpheus label provides several different appearances and color schemes that you can use for labels. You can just choose one of the pre-defined values for the Appearance and ColorScheme properties. Or, if none of the pre-defined appearances or color schemes is just right for your application, you can change them or even create your own unique style. Styles are created, named and deleted by the built-in Style Manager.

There are three parts of the label text that you can control: the characters, the highlight, and the shadow. This is illustrated in Figure 26.1.

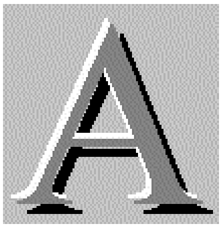


Figure 26.1: Character, highlight, and shadow

This character is dark gray, with a white highlight and a black shadow. You can set the color for each of the three parts. The highlight and shadow can be placed in any of eight positions around the text and be placed at virtually any distance (in pixels) from the text.

You can use a gradient method (a gradual change from one color to another) to paint the text, shadow, or highlight. The character shown in Figure 26.2 has a gradient highlight (the highlight color is blended into the text color).

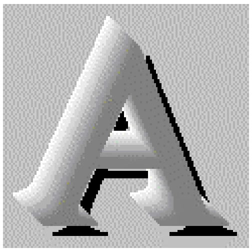


Figure 26.2: Gradient highlight

The highlight and shadow can be drawn using a color extrusion method (the character is drawn repeatedly, each time in a slightly different position). The character shown in Figure 26.3 has a color extrusion highlight.

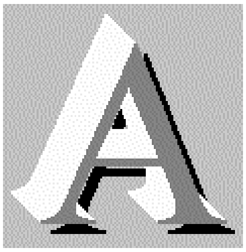
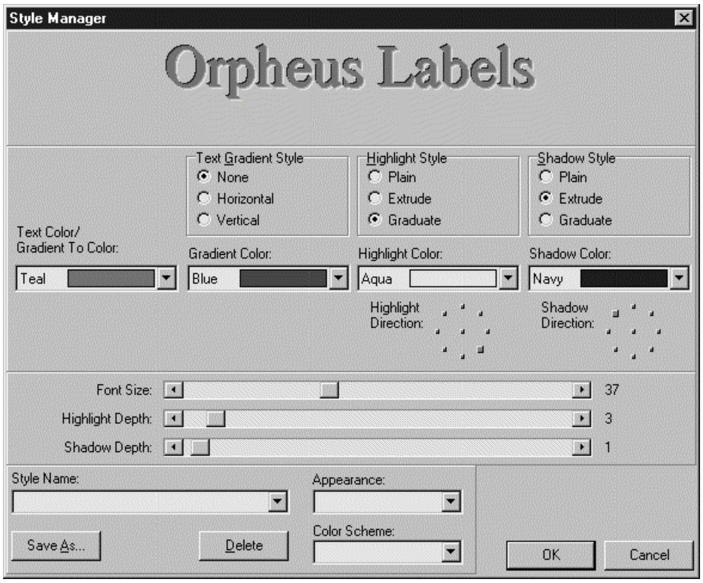


Figure 26.3: Color extrusion highlight

The possible combinations of colors and appearances give you an almost endless number of looks for your labels.

It is easy to experiment with the size, color, and look of the font, highlight, and shadow using the TOvcLabel property editor. Right click on the TOvcLabel and choose Style Manager (shown in Figure 26.4) or double-click on the CustomSettings property.


Style Manager

Orpheus Labels

Text Gradient Style: [✓ None] Horizontal Vertical

Highlight Style: Plain Extrude [✓ Graduate]

Shadow Style: Plain [✓ Extrude] Graduate

Text Color/ Gradient To Color: Teal

Gradient Color: Blue

Highlight Color: Aqua

Shadow Color: Navy

Highlight Direction:

Shadow Direction:

Font Size: 37

Highlight Depth: 3

Shadow Depth: 1

Style Name:

Appearance:

Color Scheme:

Save As... Delete OK Cancel

Figure 26.4: Style Manager

The Text Gradient Style radio buttons allow you to choose a horizontal or vertical gradient for the text (or no gradient). If you do not use a gradient, the text color is set in the Text Color/ Gradient To Color combo box. If you choose a gradient, the “from” color is set in the Gradient Color combo box and the “to” color is set in the Text Color/Gradient To Color combo box.

The Highlight Style radio buttons allow you to choose the drawing method for the highlight. You can choose a plain highlight or use the color extrusion method or the gradient method to draw the highlight. If you use a plain highlight or the extrusion method, the highlight color is set in the Highlight Color combo box. If you use a gradient method, the “from” color is set in the Highlight Color combo box and the “to” color is set in the Text Color/Gradient To Color combo box.

The Shadow Style radio buttons allow you to choose the drawing method for the shadow. You can choose a plain shadow or use the color extrusion method or the gradient method to draw the shadow. If you use a plain shadow or the extrusion method, the shadow color is set in the Shadow Color combo box. If you use a gradient method, the “from” color is set in the Shadow Color combo box and the “to” color is set in the Text Color/Gradient To Color combo box.

If you use a gradient method for the text, highlight or shadow, for the best results your graphics system should be set to more than 256 colors.

The Highlight Direction color wheel allows you to choose the direction (with respect to the text) in which the highlight is painted. If you choose the button in the middle of the wheel, no highlight is displayed.

The Shadow Direction color wheel allows you to choose the direction (with respect to the text) in which the shadow is painted. If you choose the button in the middle of the wheel, no shadow is displayed.

The font size, highlight depth, and shadow depth are set using the scroll bars at the bottom of the dialog. The range for the font size is zero to 100. The range for highlight depth and shadow depth is zero to 50. A highlight depth of zero means no highlight. A shadow depth of 0 means no shadow.

As you make changes in the Style Manager, the label at the top displays the result. Once you find a set of properties that produce the desired effect, you can save them as a style for use again and again. Just press the Save As… button and enter a name for the style.

The Appearance combo box allows you to select a “look” for the text, highlight, and shadow. This allows you to start from any of the pre-defined “looks” (see the Appearance property) and modify it to fit your needs.

The Color Scheme combo box allows you to select a color scheme for the text, highlight, and shadow. This allows you to start from any of the pre-defined color schemes (see the ColorScheme property) and modify it to fit your needs.

Example

Drop a TOvcLabel component on the form and change the Caption to “Many Shadow and Highlight Styles”. Right click on the TOvcLabel component and choose the Style Manager. Set the values as shown in Table 26.1.

Table 26.1: *TOvcLabel example*

Name	**Value**
Shadow Style	Graduate
Highlight Color	Yellow
Shadow Color	Silver
Text Color/Gradient To Color	Purple
Highlight Direction	Upper/left
Shadow Direction	Upper/left
Font Size	28
Highlight Depth	1
Shadow Depth	20

By setting the shadow color to the same color as the background (this example assumes the form color is Silver, which is the default), the shadow appears to emerge from the background.

If this is a label style you would like to use again, you probably won’t want to go through all those steps again. That’s why Orpheus provides a way to save the current settings as a “scheme.” Select the “Save As” button from the Style Manager dialog and enter a name for the scheme. Later, you can select this name from the Scheme combo box and all of the settings you just made are applied automatically.

The TOvcLabel component is derived from the VCL’s TCustomLabel and inherits its properties, events, and methods. For descriptions of the inherited properties, events, and methods, see the compiler’s documentation.

Hierarchy

TCustomLabel (VCL)

  TOvcCustomLabel (OvcLabel)

    TOvcLabel (OvcLabel)

Properties

About	ColorScheme	WordWrap
Appearance	CustomSettings	

Reference Section

About — read-only property

```
property About : string
```

Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

Appearance — property

```
property Appearance : TOvcAppearance
```

```
TOvcAppearance = (
  apNone, apCustom, apFlying, apRaised, apSunken, apShadow);
```

Default: apRaised

Determines the look of the text in the label.

The look of the text includes the position and size of the highlight and shadow as well as the method used to draw them. The possible values for Appearance include some common styles of text:

Value	**Meaning**
apNone	The text is basically the same as in a standard TLabel component.
apFlying	The text has a shadow below it to simulate the shadow under a hovering object.
apRaised	The text has a raised 3-D look.
apSunken	The text appears to be beneath the drawing surface.
apShadow	The text appears to be slightly above the drawing surface and thus produces a small shadow.
apCustom	The text has a user-defined appearance.

Appearance is automatically set to apCustom when any of the CustomSettings values are changed. You can use the CustomSettings property to create many other styles.

See also: CustomSettings

ColorScheme property

```
property ColorScheme : TOvcColorScheme
```

```
TOvcColorScheme = (
  csCustom, csText, csWindows, csEmbossed, csGold, csSteel);
```

Default: csWindows

↳ The color scheme for the label.

You can use ColorScheme to set the label colors to a pre-defined set of coordinated colors. If none of the pre-defined color schemes fits your needs, you can use the CustomSettings property to create you own color scheme.

See also: CustomSettings

CustomSettings property

```
property CustomSettings : TOvcCustomSettings
```

↳ Determines size, color, and look of the font, highlight, and shadows.

The property editor makes it easy to experiment with all of the settings that determine how the text in the label looks. As you make changes in the property editor, a sample text shows you how it looks. The property editor also allows you to save the settings as a named style for later use.

WordWrap property

```
property WordWrap : Boolean
```

Default: True

↳ Determines whether words that extend past the right edge of the label are wrapped to the next line.

If WordWrap is False and the text exceeds the width of the component, the text that extends past the edge of the component is not shown.

TOvcCustomRotatedLabel Class

The TOvcCustomRotatedLabel class is the immediate ancestor of the TOvcRotatedLabel component. It implements all the methods and properties used by the TOvcRotatedLabel component and is identical to the TOvcRotatedLabel component except that no properties are published.

TOvcCustomRotatedLabel is provided to facilitate creation of descendent rotated label components. For property and method descriptions, see “TOvcRotatedLabel Component” on page 744.

Hierarchy

TGraphicControl (VCL)

    TOvcCustomRotatedLabel (OvcRLbl)

TOvcRotatedLabel Component

The rotated label component is similar to the standard VCL TLabel component, but allows you to display text on a form at any angle and size using a TrueType font.

Example

This example creates a form that displays text at a 45° angle.

Create a new project, add components, and set the property values as indicated in Table 26.2.

Table 26.2: *TOvcRotatedLabel example*

Component	**Property**	**Value**
`TOvcRotatedLabel`	`Caption`	`Sample rotated text`
	`Width`	`200`
	`Height`	`200`
	`OriginX`	`20`
	`OriginY`	`150`
	`FontAngle`	`45`

Only TrueType fonts can be rotated, so if you select a non-TrueType font, the FontAngle is forced to 0.

Hierarchy

TGraphicControl (VCL)

  TOvcCustomRotatedLabel (OvcRLbl) 743

    TOvcRotatedLabel (OvcRLbl)

Properties

About	FontAngle	ShadowedText
Alignment	OriginX	Transparent
AutoSizeHeight	OriginY	
AutoSizeWidth	ShadowColor	

Reference Section

About — read-only property

```
property About : string
```

↳ Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

Alignment — property

```
property Alignment : TAlignment
```

```
TAlignment = (taLeftJustify, taRightJustify, taCenter);
```

Default: taLeftJustify

↳ Determines the text alignment.

AutoSizeHeight — property

```
property AutoSizeHeight : Boolean
```

Default: False

↳ Determines if the label's height is automatically determined based on the current font.

See also: AutoSizeWidth

AutoSizeWidth — property

```
property AutoSizeWidth : Boolean
```

Default: False

↳ Determines if the label's width is set automatically based on the current font and text.

See also: AutoSizeHeight

FontAngle property

```
property FontAngle : Integer
```

Default: 0

↳ The number of degrees that the font is rotated.

Positive values (0 through 360) rotate the font counter-clockwise. Negative values (0 through - 360) rotate the font clockwise.

Values outside the range of -360 through 360 are automatically converted to values in the valid range.

OriginX property

```
property OriginX : Integer
```

Default: 0

↳ Determines the position of the first character from the left side of the rotated label.

OriginX is the pixel offset of the first character and is relative to the left side of the client area.

The value assigned to OriginX is only used if FontAngle is non-zero.

See also: OriginY

OriginY property

```
property OriginY : Integer
```

Default: 0

↳ Determines the position of the first character from the top edge of the rotated label.

OriginY is the pixel offset of the first character and is relative to the top edge of the client area.

The value assigned to OriginY is only used if FontAngle is non-zero.

See also: OriginX

ShadowColor property

```
property ShadowColor  : TColor
```

Default: clButtonShadow

↳ Determines the color used when drawing the label text using text shadowing.

See also: ShadowedText

ShadowedText property

```
property ShadowedText  : Boolean
```

Default: False

Determines is the label text is drawn with shadowing.

See also: ShadowColor

Transparent property

```
property Transparent : Boolean
```

Determines whether the label hides items beneath it.

If the Transparent property is True and the rotated label overlaps another component on the form, the component covered by the rotated label is still visible. If the Transparent property is False, other components overlapped by the rotated label are hidden.

TOvcCustomPictureLabel Class

The TOvcCustomPictureLabel class is the immediate ancestor of the TOvcPictureLabel component. It implements all the methods and properties used by the TOvcPictureLabel component and is identical to the TOvcPictureLabel component except that no properties are published.

TOvcCustomPictureLabel is provided to facilitate creation of descendent picture label components. For property and method descriptions, see “TOvcPictureLabel Component” on page 749.

Hierarchy

TGraphicControl (VCL)

    TOvcCustomRotatedLabel (OvcRLbl) 743

        TOvcCustomPictureLabel (OvcPLbl)

TOvcPictureLabel Component

The picture label component is similar to the Orpheus TOvcRotatedLabel component, but allows you to display text formatted by a picture mask.

Hierarchy

TGraphicControl (VCL)

TOvcCustomRotatedLabel (OvcRLbl). 743

TOvcCustomPictureLabel (OvcPLbl) 748

TOvcPictureLabel (OvcPLbl)

Properties

AsBoolean	AsString	IntlSupport
AsDate	AsStTime	PictureMask
AsFloat	AsTime	UseIntlMask
AsInteger	AsVariant	UserData
AsStDate	AsYesNo	

26

Methods

Clear	GetDisplayString	PaintTo

Reference Section

AsBoolean — run-time, write-only property

```
property AsBoolean : Boolean
```

Causes the label to display a Boolean value.

The PictureMask must provide a mask suitable for displaying this data type.

See also: PictureMask

AsDate — run-time, write-only property

```
property AsDate : TDateTime
```

Causes the label to display the date portion of a TDateTime value.

The PictureMask must provide a mask suitable for displaying this data type.

See also: PictureMask

AsFloat — run-time, write-only property

```
property AsFloat : Extended
```

Causes the label to display a floating-point value.

The PictureMask must provide a mask suitable for displaying this data type.

See also: PictureMask

AsInteger — run-time, write-only property

```
property AsInteger : LongInt
```

Causes the label to display an integer value.

The PictureMask must provide a mask suitable for displaying this data type.

See also: PictureMask

AsStDate — run-time, write-only property

```
property AsStDate : TStDate
```

Causes the label to display an Orpheus date value.

The PictureMask must provide a mask suitable for displaying this data type.

See also: PictureMask

AsString — run-time, write-only property

```
property AsString : string
```

Causes the label to display a string value.

The PictureMask must provide a mask suitable for displaying this data type.

See also: PictureMask

AsStTime — run-time, write-only property

```
property AsStTime : TStTime
```

Causes the label to display an Orpheus time value.

The PictureMask must provide a mask suitable for displaying this data type.

See also: PictureMask

AsTime — run-time, write-only property

```
property AsTime : TDateTime
```

Causes the label to display the time portion of a TDateTime value.

The PictureMask must provide a mask suitable for displaying this data type.

See also: PictureMask

AsVariant — run-time, write-only property

```
property AsVariant : Variant
```

Causes the label to display a value that corresponds to the data type that is assigned

The PictureMask must provide a mask suitable for displaying this data type. (See the VCL's Variant data type for additional information.)

See also: PictureMask

AsYesNo — run-time, write-only property

```
property AsYesNo : Boolean
```

Causes the label to display a Boolean value using the "Yes" or "No" characters instead of the "True" and "False" characters.

The PictureMask must provide a mask suitable for displaying this data type.

See also: PictureMask

Clear method

```
procedure Clear;
```

Clears the label's value and PictureMask contents.

See also: PictureMask

GetDisplayString method

```
procedure GetDisplayString;
```

Returns the string the label is displaying.

See also: AsString, PictureMask

IntlSupport run-time property

```
property IntlSupport : TOvcIntlSup
```

Default: OvcIntlSup

Provides access to the international support object that is attached to the picture label component.

A global international support object is created during initialization and all picture label components use it by default. You can create an additional TOvcIntlSup object (see page 344), tailor the international settings, and attach it to the picture label component by assigning it to this property.

PaintTo method

```
procedure PaintTo(DC : hDC; X, Y : Integer);
```

Provides a means to draw the label contents on another canvas.

The PaintTo method can be use to draw the label contents on another canvas and location. DC is the device context (Canvas.Handle). X and Y represent the position within the device context where the label should draw itself.

```
property PictureMask : string
```

Default: “XXXXXXXXXX”

Defines the display format.

This property defines the display mask used for the picture label. The possible values are:

Value		**Meaning**
pmAnyChar	= ‘X’;	Allows any character.
pmForceUpper	= ‘!’;	Allows any character, forces upper case.
pmForceLower	= ‘L’;	Allows any character, forces lower case.
pmForceMixed	= ‘x’;	Allows any character, forces mixed case.
pmAlpha	= ‘a’;	Allows alphas only.
pmUpperAlpha	= ‘A’;	Allows alphas only, forces upper case.
pmLowerAlpha	= ‘l’;	Allows alphas only, forces lower case.
pmPositive	= ‘9’;	Allows numbers and spaces only.
pmWhole	= ‘i’;	Allows numbers, spaces, minus.
pmDecimal	= ‘#’;	Allows numbers, spaces, minus, period.
pmScientific	= ‘E’;	Allows numbers, spaces, minus, period, ‘e’.
pmHexadecimal	= ‘K’;	Allows 0-9 and A-F, forces upper case.
pmOctal	= ‘O’;	Allows 0-7, space.
pmBinary	= ‘b’;	Allows 0-1, space.
pmTrueFalse	= ‘B’;	Allows T, t, F, f.
pmYesNo	= ‘Y’;	Allows Y, y, N, n.

The value assigned to PictureMask is not tested for validity. It is possible to assign a mask that is invalid. You can directly enter a mask string in the PictureMask edit field in the Object Inspector or you can invoke the picture mask property editor to view and select from several example picture masks. Double click the PictureMask property value in the Object Inspector or select the dialog button to display the “Picture Mask” property editor.

See “Picture Masks” on page 316 for more information about picture masks and their usage.

UseIntlMask property

```
property UseIntlMask : Boolean
```

Default: False

↳ Determines if the label will use the international date mask for dates and times.

If UseIntlMask is True the label will use the mask returned from Windows (as defined in the WIN.INI file) to format date and time values. If False, the mask contained in the PictureMask property is used.

See also: PictureMask

UserData run-time property

```
property UserData : TOvcUserData
```

Default: OvcUserData

↳ Determines which TOvcUserData object the label uses for user-defined mask and substitution characters.

See “User-defined mask characters” on page 322 and “Substitution characters” on page 323for additional information.

TOvcDbPictureLabel Component

TOvcDbPictureLabel is a direct descendant of the TOvcCustomPictureLabel and inherits all its properties and methods, which are documented in the TOvcPictureLabel section.

There are very few differences in the behavior of the TOvcDbPictureLabel and TOvcPictureLabel except that TOvcDbPictureLabel is capable of connecting to a data source and displaying the value of a field within the attached database.

TOvcDbPictureField provides three additional properties. The DataField, DataSource, and Field properties are exactly the same as the like-named properties in the VCL's standard data- aware components.

To use a TOvcDbPictureLabel component, just assign a TDataSource to the DataSource property and assign a field name to the DataField property. The label automatically gets the data from the data field and displays it according to the picture mask in the PictureMask property.

Hierarchy

TGraphicControl (VCL)

  TOvcCustomRotatedLabel (OvcRLbl). 743

    TOvcCustomPictureLabel (OvcPLbl) 748

      TOvcDbPictureLabel (OvcDbPLb)

Properties

DataField	Field
DataSource	ShowDateOrTime

Reference Section

DataField — property

```
property DataField : string
```

Identifies the field (in the data source component) from which the picture label component displays data.

DataSource — property

```
property DataSource : TDataSource
```

Specifies the data source component where the picture label obtains the data to display.

Field — run-time, read-only property

```
property Field : TField
```

Returns the TField object to which the label component is linked.

Use the Field object when you want to change the value of the data in the field programmatically.

ShowDateOrTime — property

```
property ShowDateOrTime : TShowDateOrTime
```

```
TShowDateOrTime = (ftShowDate, ftShowTime);
```

Default: ftShowDate

Determines which portion of a TDateTime value the picture label displays.

If ShowDateOrTime is set to ftShowDate, the date portion of the TDateTime value is displayed. Otherwise, the time portion is displayed.

This property is ignored unless the data type of the attached database field is TDateTime.

TOvcDbDisplayLabel Component

TOvcDbDisplayLabel is a direct descendant of the TOvcCustomRotatedLabel and inherits all its properties and methods, which are documented in "TOvcRotatedLabel Component" on page 744.

There are very few differences in the behavior of the TOvcDbDisplayLabel and the TOvcRotatedLabel except that the TOvcDbDisplayLabel is capable of connecting to a data source and displaying the value of a field's DisplayLabel property.

TOvcDbDisplayField provides three additional properties. The DataField, DataSource, and Field properties are exactly the same as the like-named properties in VCL's standard data-aware components.

To use a TOvcDbDisplayLabel component, just assign a TDataSource to the DataSource property and assign a field name to the DataField property. The label automatically gets the data from the data field's DisplayLabel property and displays it accordingly.

Hierarchy

TGraphicControl (VCL)

TOvcCustomRotatedLabel (OvcRLbl) 743

TOvcDbDisplayLabel (OvcDbDLb)

Properties

DataField	DataSource	Field

Reference Section

DataField property

```
property DataField : string
```

Identifies the field (in the data source component) from which the picture label component displays data.

DataSource property

```
property DataSource : TDataSource
```

Specifies the data source component where the picture label obtains the data to display.

Field run-time, read-only property

```
property Field : TField
```

Returns the TField object to which the editor component is linked.

Use the Field object when you want to change the value of the data in the field programmatically.

TOvcURL Component

The TOvcURL component is a label component that emulates a hyperlink on a Web page. When the mouse cursor passes over the component, the text of the caption changes color. When the component is clicked, the user's default Web browser is invoked and a Web page is displayed. The Web page displayed is determined by the value of the URL property.

Although TOvcURL was designed to display Web pages, it can invoke a wide variety of applications. You can, for example, set the URL property to the path and file name of a text file. When the component is clicked, Windows will execute the application that is associated with the TXT file name extension (usually Notepad) and the text file will be loaded. Similarly, if you set the URL property to an e-mail address (such as 'mailto: info@turbopower.com', for example), Windows will start the user's default e-mail client and populate the To: line with the specified e-mail address when the Label is clicked. Setting the OvcUrl's FontColor to clBlue and the FontStyle to fsUnderline will cause it to have a familiar hyperlink look.

Hierarchy

TCustomLabel (VCL)

    TOvcURL (OvcURL)

Properties

About	HighlightColor
Caption	URL

Reference Section

About — read-only property

```
property About : string
```

↳ Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

Caption — property

```
property Caption : string
```

↳ The text the TOvcURL component will display.

If no caption is specified, the text of the URL property is used as the caption.

See also: URL

HighlightColor — property

```
property HighlightColor : TColor
```

Default: clRed

↳ The color of the component's caption when the mouse moves over the component.

The OvcURL component emulates a hypertext link. The caption is initially displayed with the color as determined by the Font's Color property. When the cursor passes over the component, the caption is redrawn in the color specified by HighlightColor. If HighlightColor is set to clNone, the caption's text color does not change when the mouse passes over.

See also: Caption

```
property URL : string
```

↳ The Internet address of the page that will appear in the Web browser when the component is clicked.

Set URL to the Internet address of the page you want displayed when the user's Web browser starts. URL will typically be a Web site address. You can, however, use the OvcURL component to display any document type for which Windows has a file association. Setting URL to the path and filename of a text file, for example, will cause Windows to execute Notepad and display the file when the component is clicked (assuming the user has not changed the association for the TXT file extension).

TOvcAttachedLabel Component

TOvcAttachedLabel is a special descendant of the standard TLabel component, which adds the capability to associate itself with another control.

Nearly all Orpheus components have the ability to display an attached label and to communicate to it so that the label remains positioned in the same place relative to the component. Each of the Orpheus controls that have this ability will publish the LabelInfo property. LabelInfo allows you to decide if the attached label is visible and to fine-tune the label position (see TOvcLabelInfo on page 764).

Note: The link maintained between the TOvcAttachedLabel and other controls persists through normal save and load operations but not when components are cut to and copied from the Windows clipboard. If you cut both the attached label and control to the clipboard, you will need to use the attached label's Control property to manually reattach the label to the control after they have been pasted.

Hierarchy

TLabel (VCL)

TOvcAttachedLabel (OvcBase)

Properties

Control

Reference Section

Control property

```
property Control : TWinControl
```

↳ Determines which Orpheus component the label is associated with.

Control is set automatically when the attached label is made visible and is also used to determine which Orpheus control the label is attached to when the form and your application are loaded.

TOvcLabelInfo Class

TOvcLabelInfo implements a persistent storage class used to store properties that pertain to the attached label (see TOvcAttachedLabel on page 762). It allows you to determine if the attached label is visible and to adjust the position of the attached label relative to the component that publishes the LabelInfo property. (LabelInfo is the name of the property published by Orpheus components that provides access to the instance of TOvcLabelInfo.)

Hierarchy

TPersistent (VCL)

    TOvcLabelInfo (OvcBase)

Properties

OffsetX	OffsetY	Visible

OffsetX property

```
property OffsetX : Integer
```

Default: 0

↳ Determines the distance (in pixels) between the left edge of the control and the left edge of the attached label.

OffsetY property

```
property OffsetY : Integer
```

Default: 0

↳ Determines the distance (in pixels) between the top of the control and the bottom of the attached label.

Visible property

```
property Visible : Boolean
```

↳ Determines if the attached label is displayed.

Setting Visible to True creates and displays the attached label above and aligned to the left edge of the control. Setting Visible to False hides and destroys the attached label.

26

Chapter 27: Text Editors

Orpheus provides two text editor components. The text editor, TOvcEditor, is a general-purpose editor that can edit ASCII text (since you supply the text, it can come from any source). The text file editor, TOvcTextFileEditor, is a special-purpose editor for editing ASCII text files. Throughout this chapter references to “editor” or “editor component” apply to both the text editor and the text file editor.

The editor component is a multi-line text editor capable of editing ASCII text (up to 16MB). It provides a powerful undo facility that allows multiple edit commands to be undone. In addition, the redo command allows you to undo an undo.

Multiple edit components can work with the same text stream. This allows you to work with different views of a file at the same time.

True real-time word wrap is available as an option. It can be turned on or off at any time and the column at which word wrap occurs is adjustable. Paragraphs can be up to 32K characters long.

Search and replace functions are provided by the edit control. The application must provide for initiating the search or replace and specifying the options. The available search options include backwards search, case-sensitive search, whole word search, and search from beginning/end of file. The replace function allows you to replace either a particular occurrence or all occurrences.

The editor component works with fixed pitch fonts and does not support multi-font viewing.

The editor component provides complete support for the Windows clipboard. While editing, you can copy, cut, and paste text using the standard <CtrlC>, <CtrlX>, and <CtrlV> commands or special keyboard commands that you add to the command processor.

A data-aware version of the editor is also provided. The TOvcDbEditor can be connected to a data source to allow editing the text of a memo field.

Text Handling

The Orpheus text editors give your application sophisticated text handling capabilities. You get all the standard features like searching for and replacing text, tab expansion, and word wrap. In addition there are nice extras like bookmarks, the ability to limit user input, and a powerful undo facility.

Searching for and replacing text

The editor component offers no built-in commands to allow searching for or replacing text, since doing so would require it to display a form that might conflict with the overall look-and- feel of the parent application. It does, however, provide a Search method that you can call to search for text and a Replace method that you can call to make replacements. It is your responsibility to provide the means for initiating the search, and for displaying a form in which the user can enter search criteria and a replacement string (see the TEXTEDIT demo program for an example).

The Search method takes two parameters, a search string and a set of search options. The available search and replace options are listed in Table 27.1.

Table 27.1: *Search and replace options*

Option	Description
soFind	A single find operation is requested.
soReplace	A single replace operation is requested.
soReplaceAll	All matching text in a given range should be replaced.
soBackward	Search backward.
soMatchCase	Perform a case-sensitive search.
soWholeWord	A match should not occur if the search string is found embedded in another word.
soSelText	Search only in the currently selected text.
soGlobal	The search should start at the beginning or end of the file, depending on the state of the soBackward option, instead of at the current caret position.

These options are defined in OvcData, so you must add OvcData to your unit's uses clause (Delphi) or OVCDATA.HPP file (C++Builder), if appropriate.

A return value of True indicates that the search string was found and is currently highlighted in the editor.

The Replace method is similar to Search, but takes a third parameter, a replacement string. The options passed to Replace must specify either soReplace or soReplaceAll. You can also specify any or all of the other option flags listed above. Replace returns a long integer that indicates how many replacements were made. A return value of -1 means that the search string wasn't found. A return value of 0 means that the search string was found, but no replacements were made. Any other return value indicates the number of replacements that were made.

Bookmarks

Bookmarks provide a means of moving quickly from place to place within a file. Assume that you are primarily interested in two sections of a text file that are widely separated, and want to be able to move between the two of them quickly. This can be accomplished by setting a bookmark at the beginning of each section, and jumping between them with the GoToMarker command.

To set a bookmark using the default key assignments, press <ShiftCtrlN>, where N is a number in the range 0 to 9. The current position of the caret is marked as bookmark N. To return to that spot later, press <CtrlN>, where N is the number used to set the bookmark.

You might also want to have a menu item that sets a bookmark at the current position. You can call either the SetMarker method, which sets a bookmark at the current position of the caret, or the SetMarkerAt method, which sets a bookmark at a specified line/column position.

The Orpheus editor and viewer components allow either hidden or visible bookmarks.

Tabs

Tab characters in the text stream are expanded to spaces. The number of spaces used is a function of the contents of the line and the current tab size, which can be changed by assigning a new value to the TabSize property. The default tab size is 8. The <Tab> key is processed by the editor only if the WantTab property is set to True.

You can change the behavior of the editor component by changing the TabType property. If the tab type is set to ttReal, pressing <Tab> inserts a real tab character into the text stream. If the tab type is set to ttFixed, pressing <Tab> inserts enough spaces to move the caret to the next tab stop. (This is called *fixed tabs* because the position of existing text will not change if the tab size is later changed, as it does if real tabs are used.) If the tab type is set to ttSmart, spaces are also inserted into the text stream, but the positions of the tab stops are a function of the positions of the words in the previous line.

The following illustrates the positions of the tab stops when the tab type is set to ttSmart:

```
previous line:    This is a sample line demonstrating smart tab
placement.
current line: x   x  x x     x    x            x    x   x
```

In most applications, the default setting of ttReal should be used. The other two tab types should be available, if at all, only as optional settings that can be selected or deselected by the user at run time.

Word wrap

The editor component offers true word wrap as an option. Word wrap can be enabled initially by setting the WordWrap property to True, and it can be turned on and off at any time by changing the property value at run time. By default, word wrap occurs at column 80. This setting can be changed by assigning a new value to the WrapColumn property.

When the word wrap option is enabled, editing is done on a paragraph-by-paragraph basis, rather than a line-by-line basis. That is, when the user presses <Enter> while in insert mode, a new paragraph is created. As text is entered into the paragraph, or existing text is deleted, the editor component automatically recalculates where line breaks should occur, and redraws the affected portions of the screen. The line breaks exist only in the editor component's internal data structures; the text of the paragraph contains no special characters to mark the line breaks.

Undo and redo

The editor component provides a powerful undo facility that allows multiple edit commands to be undone. You can even "undo an undo" by issuing a redo command. Using the default command processor, only the newer Windows commands for undo and redo are supported (<CtrlZ> and <ShiftCtrlZ>). You can easily add support for the standard Windows 3.0 commands (<AltBkSp> and <ShiftAltBkSp>) by adding these commands to the command table. Keep in mind, however, that duplicate command key sequences are not allowed within the same command table.

The editor component stores a record of each edit command in a single contiguous buffer, which is 8KB by default. The size of this buffer can be adjusted by assigning a new value to the UndoBufferSize property. You can request a buffer of up to 64KB, or you can disable the undo facility completely by specifying a size of 0. It is not possible to configure the editor component to undo only the last command, since the undo record could occupy anywhere from 16 bytes to several thousand bytes in the undo buffer.

The editor component is designed to treat a series of related edit commands as a single command. For example, if you move the caret to the beginning of a paragraph, delete 10 characters by pressing the <Del> key 10 times, then issue the undo command, all 10 deletions are undone. Similarly, if you insert 10 characters of text at the beginning of the paragraph, issuing the undo command deletes all 10 characters. There is no provision in the editor for allowing you to undo these 10 insertions individually, one character at a time.

Line/column vs. paragraph/offset coordinates

To facilitate the implementation of the word wrap option, the editor component stores all text in a linked list of paragraphs, rather than lines. This approach has many benefits, but one minor drawback: sometimes locations within the text stream need to be expressed in terms of line/column coordinates, and sometimes they need to be expressed in terms of paragraph/ offset coordinates. Both of these coordinate systems are 1-based.

For the most part, the dual coordinate systems affect only the editor component itself. With only a couple of exceptions, all of the routines provided by TOvcEditor accept and return line/ column coordinates. The only time that you really need to concern yourself with paragraph/ offset coordinates is when you are working directly with the text being edited and examining the text of entire paragraphs rather than individual lines.

The following is an outline, in pseudo-code form, of what you would need to do to implement your new-and-improved search facility:

```
;get current position in paragraph, offset coordinates
CurLine = GetCaretPosition(CurCol)
CurPara = CurLine
CurOfs = CurCol
LineToPara (CurLine, CurOfs)

{get total number of paragraphs}
TotalParas = GetParaCount()

;search each paragraph in the text stream
Found = False
while (CurPara <= TotalParas) and (not Found)
  {get pointer to this paragraph}
  P = GetPara (CurPara)
  {point to current position in this paragraph}
  P = P+(CurOfs-1);
  {search this paragraph (RegularExpSearch is an imaginary
   function)}
  Ofs = RegularExpSearch(P, SearchMask, MatchLen)
  if Ofs = -1
    {not found, point to start of next paragraph}
    CurPara = CurPara+1
    CurOfs = 1
    else
      {Ofs is a 0-based offset into the paragraph for the match}
      Found = True
      CurOfs = CurOfs+Ofs

  {was text found?}
  if Found
    {start highlight at CurPara/CurOfs --
      convert to line/column coordinates}
    SelStartLine = CurPara
    SelStartCol = CurOfs
    ParaToLine(SelStartLine, SelStartCol)
    {end highlight at CurPara/CurOfs+MatchLen --
      convert to line/column}
    SelEndLine = CurPara
    SelEndCol = CurOfs+MatchLen
    ParaToLine(SelEndLine, SelEndCol)
    {highlight the match}
    SetSelection(SelStartLine, SelStartCol, SelEndLine,
                 SelEndCol, True)
```

The idea here is to: 1) obtain the current position in paragraph/offset coordinates; 2) search forward from the caret position a paragraph at a time; and 3) if a match is found, calculate the position of the match in line/column coordinates and call SetSelection to highlight it. The key to making this algorithm work are the LineToPara and ParaToLine methods, which allow you to convert coordinates from one system to another.

You might be wondering if it wouldn't be easier to just use line/column coordinates throughout and call GetLine instead of GetPara, and of course it would. But the results would be less than satisfactory if word wrap were on, because you'd never find matching strings that spanned multiple lines in the same paragraph. For example, if you were searching for "John Doe", you wouldn't find a match in a case where word wrap occurred after "John."

Limiting user input

In some cases, you may want or need to limit the amount of text that can be entered. The editor component allows you to set three kinds of limits, which can be used either separately or together.

The ByteLimit property imposes a limit on the total number of bytes that can be entered into the text stream. The default is 7FFFFFFF (MaxLongInt) bytes.

The ParaLimit property imposes a limit on the total number of paragraphs that can be entered. Typically, you would set ParaLimit only if word wrap is disabled, so that there is a one-to-one correspondence between lines and paragraphs. The default limit is 7FFFFFFF paragraphs.

The ParaLengthLimit property imposes a limit on the length of a paragraph. Like ParaLimit, ParaLengthLimit is typically set only if word wrap is disabled, and what you really limit is the length of individual lines. The default limit is 32,767 (MaxInt) bytes, which is also the maximum value.

Printing the text in the editor

Neither the TOvcEditor nor the TOvcTextFileEditor component offers any direct means of printing the text displayed by the editor. However, you can get the text of a line in a form suitable for printing (i.e., with tabs expanded to spaces). GetPrintableLine loads a printable version of the specified line into a buffer that you provide, and returns the length of the string in the buffer. Lines (an indexed property) returns a string containing the printable version of the line.

TOvcCustomEditor Class

The TOvcCustomEditor class is the immediate ancestor of the TOvcEditor component. It implements all of the methods and properties used by the TOvcEditor component and is identical to the TOvcEditor except that no properties are published.

TOvcCustomEditor is provided to facilitate creation of descendent editor components. For property and method descriptions, see “TOvcEditor Component” on page 775.

Hierarchy

TCustomControl (VCL)

  TOvcCustomControl (OvcBase) 16

    TOvcCustomControlEx (OvcBase) 19

      TOvcCustomEditor (OvcEdit)

TOvcEditor Component

When you use the TOvcEditor component, you are responsible for obtaining the text stream from wherever it is stored and loading it into the editor. If you use the TOvcTextFileEditor component (see page 821), this step is automated for you, but this reduces its flexibility and prevents you from editing text that is embedded in a database record, for example. Similarly, when using the TOvcEditor you must take steps to store the editor text stream back to the desired file.

There are three steps involved in loading data into the TOvcEditor component:

1. Delete any existing text by calling the DeleteAll method. This can be skipped if the editor component is newly created. It is necessary only when loading new text into an existing editor.

2. Make repeated calls to the AppendPara function to add each paragraph of text to the editor's linked list of paragraphs. This can be skipped if there is no existing text to be edited.

3. Call the ResetScrollBars method, which effectively signals that you are finished adding paragraphs to the editor's text stream.

If the text to be edited is in a single contiguous buffer and the end of the text is marked by a terminating null, you can combine these three steps into one by calling the SetText method, passing a pointer to the buffer as the parameter.

Extracting data from the editor is a simpler, two-step process:

1. Use the ParaCount run-time property to find out how many paragraphs are in the editor's text stream.

2. Use the ParaPointer indexed property once for each paragraph in the text stream, and either write the contents of the paragraph to disk or make a copy of it.

Caution: Do not store a copy of the pointer returned by the ParaPointer property; it is a pointer to memory owned by the editor and will be released when the editor component is destroyed.

Although the editor component always assumes that a paragraph is terminated by a carriage return-line feed (CRLF) sequence, it does not actually store these characters. So, when you use AppendPara to add paragraphs, you must remove any trailing CRLF pair before making the call. When extracting paragraphs with ParaPointer, you must account for the assumed CRLF at the end of the paragraph when necessary (e.g., when writing a paragraph out to disk).

For good examples of how to load data into an editor component and get it back, see the source code for the LoadFromFile and SaveToFile methods in the TOvcTextFileEditor component.

Editor commands

The commands listed in Table 27.2 are available in any Orpheus editor. The commands and key assignments described here are available when you use the “Default” or “WordStar” command tables (several command tables can be used at the same time). See “TOvcCommandProcessor Class” on page 50 for information on how to customize the command tables.

Table 27.2: *Editor commands*

Command	**Default**	**WordStar**	**Description**
ccBack	<BkSp>	<CtrlH>	Delete the character to the left of the caret.
ccBotOfPage	<CtrlPgDn>	Not assigned.	Move the caret to the bottom of the window.
ccCopy	<CtrlIns>, <CtrlC>	Not assigned.	Copy the selected text to the clipboard.
ccCut	<ShiftDel>, <CtrlX>	Not assigned.	Delete the selected text after copying it to the clip board.
ccDel	<Del>	<CtrlG>	Delete the current character. Or, if text is selected, delete the selection.
ccDelEol	Not assigned.	<CtrlQ><Y>	Delete text from the caret position to the end of the line.
ccDelLine	Not assigned.	<CtrlY>	Delete the current line.
ccDelWord	Not assigned.	<CtrlT>	Delete the word to the right of the caret.
ccDown	<Down>	<CtrlX>	Move the caret down to the next line.

Table 27.2: *Editor commands (continued)*

Command	Default	WordStar	Description
ccEnd	<End>	<CtrlQ><D>	Move the caret to the end of the current line.
ccExtBotOfPage	<CtrlShiftPgDn>	Not assigned.	Extend the selection to the bottom of the page.
ccExtendDown	<ShiftDown>	Not assigned.	Extend the selection down to the next line.
ccExtendEnd	<ShiftEnd>	Not assigned.	Extend the selection to the end of the current line.
ccExtendHome	<ShiftHome>	Not assigned.	Extend the selection to the start of the current line.
ccExtendLeft	<ShiftLeft>	Not assigned.	Extend the selection to the left by one character.
ccExtendPgDn	<ShiftPgDn>	Not assigned.	Extend the selection down one page.
ccExtendPgUp	<ShiftPgUp>	Not assigned.	Extend the selection up one page.
ccExtendRight	<ShiftRight>	Not assigned.	Extend the selection to the right by one character.
ccExtendUp	<ShiftUp>	Not assigned.	Extend the selection up one line.
ccExtFirstPage	<CtrlShiftHome>	Not assigned.	Extend the selection to the first row.
ccExtLastPage	<CtrlShiftEnd>	Not assigned.	Extend the selection to the last row.
ccExtTopOfPage	<CtrlShiftPgUp>	Not assigned.	Extend the selection to the top of the page.
ccExtWordLeft	<CtrlShiftLeft>	Not assigned.	Extend the selection to the left by one word.

Table 27.2: *Editor commands (continued)*

Command	Default	WordStar	Description
ccExtWordRight	<CtrlShiftRight>	Not assigned.	Extend the selection to the right by one word.
ccFirstPage	<CtrlHome>	Not assigned.	Move the caret to the first character in the editor's text stream.
ccGotoMarker0-9	<Ctrl0-9>	<CtrlQ><0-9>	Move to the previously set marker (0-9).
ccHome	<Home>	<CtrlQ><S>	Move the caret to the beginning of the line.
ccIns	<Ins>	<CtrlV>	Toggle insert mode. The caret reflects the current mode. By default, a solid line indicates insert mode; a block indicates overwrite mode.
ccLastPage	<CtrlEnd>	Not assigned.	Move the caret to the last character in the editor's text stream.
ccLeft	<Left>	<CtrlS>	Move the caret left one character.
ccNewLine	<CtrlEnter>	Not assigned.	Create a new line.
ccNextPage	<PgDn>	<CtrlC>	Move the caret to the next page.
ccPaste	<ShiftIns>, <CtrlV>	Not assigned.	Paste the text from the clipboard.
ccPrevPage	<PgUp>	<CtrlR>	Move the caret to the previous page.
ccRedo	<CtrlShiftZ>	Not assigned.	Redo the last undone operation.
ccRight	<Right>	<CtrlD>	Move the caret right one character.
ccScrollDown	not assigned	<CtrlZ>	Scroll down one line.

Table 27.2: *Editor commands (continued)*

Command	Default	WordStar	Description
ccScrollUp	Not assigned.	<CtrlW>	Scroll up one line.
ccSetMarker0-9	<CtrlShift0-9>	<CtrlK><0-9>	Set the position of marker (0-9) to the current caret position.
ccTab	Not assigned.	Not assigned.	Insert a tab character into the text stream.
ccTopOfPage	<CtrlPgUp>	Not assigned.	Move the caret to the top of the window.
ccUndo	<CtrlZ>	not assigned	Undo the last operation.
ccUp	<Up>	<CtrlE>	Move the caret up to the previous line.
ccWordLeft	<CtrlLeft>	<CtrlA>	Move the caret left one word.
ccWordRight	<CtrlRight>	<CtrlF>	Move the caret right one word.

Example

This example shows how to construct and use an editor component to allow entry of text. The initial editor contains no text in its text stream.

Create a new project and add the following declaration to the private area of the form’s class definition in the source code window:

```
  EditorData : array[0..2047] of AnsiChar;  {a 2K text buffer}
```

This is the data structure that will hold the information entered within the editor.

Add components and set the property values as indicated in Table 27.3.

Table 27.3: *Editor component example*

Component	**Property**	**Value**
TOvcEditor	ByteLimit	2048
	Margin	2
	UndoBufferSize	1024
	WordWrap	True
	WrapColumn	40
TButton	Caption	Save

Double-click on the Save button and modify the generated method to match this:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I  : LongInt;
  S  : PAnsiChar;
begin
  EditorData[0] := #0;
  with OvcEditor1 do begin
    for I := 1 to ParaCount do begin
      {get pointer to paragraph}
      S := ParaPointer[I];

      {append new paragraph to end of buffer}
      StrLCat(EditorData, S, 2048);
      StrLCat(EditorData, #13#10, 2048);
    end;
  end;
end;
```

This method gets the text entered into the editor one paragraph at a time and places it in the EditorData buffer, ready for writing to a file. It does not actually write to a file.

Run the project and experiment with the generated program. You can use the IDE to step through the Button1Click method to get a feel for what it is doing.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase) 16

❷ TOvcCustomControlEx (OvcBase) 19

TOvcCustomEditor (OvcEdit) 774

TOvcEditor (OvcEdit)

Properties

❶	About	LineLength	ShowLineNumbers
❶	AttachedLabel	Lines	ShowRules
	AutoIndent	MarginColor	TabSize
	Borders	MaxLength	TabType
	BorderStyle	Modified	TextLength
	ByteLimit	NewStyleIndent	TopLine
	CaretIns	NextEditor	TrimWhiteSpace
	CaretOvr	ParaLength	UndoBufferSize
❷	Controller	ParaLengthLimit	VisibleColumns
	FirstEditor	ParaLimit	VisibleRows
	FixedFont	ParaPointer	WantEnter
	HideSelection	PrevEditor	WantTab
	HighlightColors	ReadOnly	WheelDelta
	InsertMode	RightMargin	WordDelimiters
❶	LabelInfo	RightMarginLine	WordWrap
	LeftColumn	ScrollBars	WrapAtLeft
	LeftMargin	ScrollBarsAlways	WrapColumn
	LeftMarginLine	ScrollPastEnd	WrapToWindow
	LineCount	ShowBookmarks	

Methods

- AppendPara
- Attach
- BeginUpdate
- CanRedo
- CanUndo
- Clear
- ClearMarker
- ClearSelection
- CopyToClipboard
- CutToClipboard
- DeleteAll
- Deselect
- EffectiveColumn
- EndUpdate
- FlushUndoBuffer
- GetCaretPosition
- GetCurrentWord
- GetLine
- GetMarkerPosition
- GetMousePos
- GetPara
- GetPrintableLine
- GetSelection
- GetSelTextBuf
- GetSelTextLen
- GetText
- GotoMarker
- HasSelection
- Insert
- InsertString
- LineToPara
- ParaCount
- ParaToLine
- PasteFromClipboard
- ProcessCommand
- Redo
- Replace
- ResetScrollBars
- Search
- SelectAll
- SetCaretPosition
- SetMarker
- SetMarkerAt
- SetSelection
- SetSelTextBuf
- SetText
- Undo

Events

- ❶ AfterEnter
- ❶ AfterExit
- OnDrawLine
- OnError
- ❶ OnMouseWheel
- OnShowStatus
- OnTopLineChanged
- OnUserCommand

Reference Section

AppendPara — method

```
function AppendPara(Para : PAnsiChar) : Word;
```

Appends a string to the end of the internal list of paragraphs.

This method should be used only in conjunction with DeleteAll and ResetScrollBars. When you want to load a new file into the text editor, call DeleteAll to delete all existing paragraphs. Next, call AppendPara in a loop to add each paragraph in the file to the editor. Finally, call ResetScrollBars. See the implementation of the TOvcTextFileEditor for an example of this process.

The function result is 0 if AppendPara is successful. Otherwise, it is an error code (see the OnError event on page 800 for a list of error codes).

See also: DeleteAll, ResetScrollBars

Attach — virtual method

```
procedure Attach(Editor : TOvcCustomEditor); virtual;
```

Attaches this editor to another editor's text stream.

Use Attach when you need to allow multiple editors to display the same file. Once you have obtained an instance pointer to any other editor component that is displaying the file, you can pass the pointer to Attach. Once the two editors are attached to each other, any changes in one editor's window are reflected in the other's (as time permits). There is no limit on the number of editors that can be attached. There is no Detach routine to remove an editor from the linked list of attached editors, because calling DeleteAll (or destroying the editor) effectively does so.

See also: DeleteAll

AutoIndent — property

```
property AutoIndent : Boolean
```

Default: False

Determines whether automatic indentation for new lines is active.

If AutoIndent is True, a new paragraph created by pressing <Enter> or <CtrlEnter> is given the same indentation level as the last line of the previous paragraph (this is generally useful only when word wrap is off).

See also: TabType, NewStyleIndent

BeginUpdate method

```
procedure BeginUpdate;
```

↳ Prevents the screen from being repainted while changes are made to the editor's text stream.

After BeginUpdate is called, the editor's window is not updated. EndUpdate must be called to re-enable screen repainting. It is probably best to use a try-finally block to ensure that EndUpdate is called.

The following example uses a try-finally block to ensure that EndUpdate is called:

```
  BeginUpdate;
  try
    {do something with the editor's text stream}
  finally
    EndUpdate;
  end;
```

See also: EndUpdate

Borders property

```
property Borders
```

↳ An implementation of the TOvcBorders class.

Borders provides an interface to the TOvcBorders class, which provides control over the way the optional borders appear.

See also: TOvcBorders

BorderStyle property

```
property BorderStyle : TBorderStyle
```

Default: bsSingle

↳ Determines the style used when drawing the border.

The possible values are bsSingle (a single line border is drawn around the component) or bsNone (no border line).

The TBorderStyle type is defined in the VCL's Controls unit.

ByteLimit property

```
property ByteLimit : LongInt
```

Default: MaxLongInt

↳ The maximum number of bytes (characters) that can be entered into the editor.

This property sets a limit on the total number of bytes of text (including CRLF characters) that can be entered.

The following example sets a total byte limit of 512KB:

```
  Editor1.ByteLimit := (512*1024);
```

CanRedo method

```
function CanRedo : Boolean;
```

↳ Returns True if the last undo operation can be redone.

CanRedo is typically used in applications that provide a menu item to redo the last operation. If CanRedo returns False, the menu item should be disabled. It is not necessary to call CanRedo before calling Redo.

See also: CanUndo, Redo, Undo

CanUndo method

```
function CanUndo : Boolean;
```

↳ Returns True if the last change can be undone.

CanUndo is typically used in applications that provide a menu item to undo the last operation. If CanUndo returns False, the menu item should be disabled. It is not necessary to call CanUndo before calling Undo.

See also: CanRedo, Redo, Undo

CaretIns property

```
property CaretIns : TOvcCaretType
```

Default: Vertical line caret

↳ Determines the characteristics of the editing caret.

This property allows you to customize the attributes of the editing caret used while the editor is in insert mode. For additional information see “TOvcCaret Class” on page 1147.

See also: CaretOvr

CaretOvr property

```
property CaretOvr : TOvcCaretType
```

Default: Solid block caret

↳ Determines the characteristics of the editing caret.

This property allows you to customize the attributes of the editing caret used while the editor is in overwrite mode. For more information see “TOvcCaret Class” on page 1147.

See also: CaretIns

Clear method

```
procedure Clear;
```

↳ Deletes the entire editor contents.

See also: ClearSelection, Deselect

ClearMarker method

```
procedure ClearMarker(N : Byte);
```

↳ Clears the specified bookmark.

This method removes the bookmark specified by N. If the bookmark was displayed, the editor is updated to erase the bookmark bitmap. Valid values for N are 0 to 9.

If the bookmark specified by N has not been set, no action is taken.

See “Bookmarks” on page 769 for more information.

See also: SetMarker, SetMarkerAt

ClearSelection method

```
procedure ClearSelection;
```

↳ Deletes the current selection.

This method deletes the currently selected region of text. If there is no selection, ClearSelection does nothing.

The following example deletes the currently selected text, if any (the call to GetSelection is not necessary):

```
if GetSelection(Line1, Col1, Line2, Col2) then
  ClearSelection;
```

See also: CopyToClipboard, CutToClipboard, GetSelection, PasteFromClipboard

CopyToClipboard — dynamic method

```
procedure CopyToClipboard; dynamic;
```

↳ Copies the selection to the Windows clipboard.

This method copies the currently selected text to the clipboard. If there is no selection, CopyToClipboard does nothing.

The following example copies the currently selected text, if any, to the clipboard (the call to GetSelection is not necessary):

```
if GetSelection(Line1, Col1, Line2, Col2) then
  CopyToClipboard;
```

See also: Clear, CutToClipboard, GetSelection, PasteFromClipboard

CutToClipboard — dynamic method

```
procedure CutToClipboard; dynamic;
```

↳ Copies the selection to the clipboard and then deletes the selected text.

This method copies the currently selected text to the clipboard, then deletes it. If there is no selection, CutToClipboard does nothing.

The following example copies the currently selected text, if any, to the clipboard and then deletes it (the call to GetSelection is not necessary):

```
if GetSelection(Line1, Col1, Line2, Col2) then
  CutToClipboard;
```

See also: Clear, CopyToClipboard, GetSelection, PasteFromClipboard

DeleteAll method

```
procedure DeleteAll(UpdateScreen : Boolean);
```

Deletes all text in the editor.

This method deletes all text in the editor, detaches it from any list of attached editors to which it belongs, flushes the undo buffer, resets the scroll bars, and clears the editor's modified flag. Passing True for UpdateScreen indicates that the window should be updated.

See also: AppendPara, ResetScrollBars

Deselect method

```
procedure Deselect(CaretAtEnd : Boolean);
```

Removes the highlight for selected text.

If CaretAtEnd is True, the selection is cleared and the caret is placed at the end of the selection. If CaretAtEnd is False, the selection is cleared and the caret is placed at beginning of the selection. If no text is selected, Deselect does nothing.

See also: Clear, ClearSelection

EffectiveColumn method

```
function EffectiveColumn(S : PAnsiChar; Col : Integer) : Integer;
```

Converts an actual column to an effective column.

Col is the actual column number. It indicates the position of the caret within the string, without regard for tab expansion. If the caret is on the fourth character in the current line, the actual column number is 4.

The effective column number is the editor column at which the caret would be placed given the contents of string S, the current tab size, and the actual column number. For example, suppose that the tab size is 8, the Col parameter is 2, and S is a two-character string consisting of a tab character followed by 'A'. EffectiveColumn would return 9 in this case, since the second character in the string is 'A', and 'A' would be displayed in the editor at column 9.

If S contains no tab characters, the actual column and the effective column are the same.

See also: GetCaretPosition

EndUpdate method

```
procedure EndUpdate;
```

↳ Re-enables screen repainting.

If BeginUpdate is called, EndUpdate must be called to re-enable screen repainting. See BeginUpdate for more information and an example.

See also: BeginUpdate

FirstEditor run-time, read-only property

```
property FirstEditor : TOvcCustomEditor
```

↳ Returns a pointer to the first attached editor.

This method returns a pointer to the first editor component in the list of editors attached to the current editor. If there are no other editors attached to the current editor, FirstEditor returns a pointer to the current editor. Note that NextEditor and PrevEditor do likewise because the list of attached editors is a circular linked list.

The following example shows how to cycle through all attached editors:

```
  First := Editor1.FirstEditor;
  AEditor := First;

  repeat
    {do something with AEditor}
    AEditor := AEditor.NextEditor;
  until AEditor = First;
```

See also: Attach, NextEditor, PrevEditor

FixedFont property

```
property FixedFont : TOvcFixedFont
```

Default: System (fpFixed)

↳ Determines which font the editor uses.

FixedFont performs the same function as the standard Font property, except that it limits the displayed fonts (in the Name sub-property) to only fixed-pitch fonts. This is because the editor component supports only fixed-pitch fonts.

FlushUndoBuffer method

```
procedure FlushUndoBuffer;
```

↳ Empties the editor's undo buffer.

This method is typically used only when your application is implementing a user-defined function that modifies the editor's text stream in such a way that the change cannot be undone (e.g., changing the case of all selected text).

See also: UndoBufferSize

GetCaretPosition method

```
function GetCaretPosition(var Col : Integer) : LongInt;
```

↳ Returns the current position of the editing caret.

This method returns the current line number as the function result and the current column number in Col. The value returned in Col is an "actual" column number, not an "effective" column number. See EffectiveColumn for a discussion of column numbers.

See also: EffectiveColumn, SetCaretPosition

GetCurrentWord method

```
procedure GetCurrentWord : string;
```

↳ Returns a string containing the word at the current caret position.

GetCurrentWord uses the characters defined by the WordDelimiters property to determine the beginning and end of the word.

See also: WordDelimiters

GetLine method

```
function GetLine(LineNum : LongInt;
  Dest : PAnsiChar; DestLen : Integer) : PAnsiChar;
```

↳ Returns the text of the specified line.

GetLine returns the text of the line specified by LineNum. Dest is the buffer for the string, and DestLen is its size. The function result equals Dest. If word wrap is off, a single line can contain as many as 32,767 characters unless ParaLengthLimit or ByteLimit were used to limit the amount of text. If the line is longer than DestLen-1 characters, it is truncated.

See also: ByteLimit, GetPara, GetPrintableLine, ParaLengthLimit

GetMarkerPosition method

```
function GetMarkerPosition(N : Byte; var Col : Integer) : LongInt;
```

Obtains the position of the specified bookmark.

The function result is the line number and Col is the position within the line. N is the bookmark number (0 through 9). If the specified bookmark has not been set, GetMarkerPosition returns -1.

See also: SetMarker, SetMarkerAt

GetMousePos method

```
procedure GetMousePos(
  var L : LongInt; var C : Integer; Existing : Boolean);
```

Returns the line and column the mouse is currently over.

The GetMousePos method returns the position of the mouse in line/column coordinates. The line is returned in L and the column is returned in C.

If the mouse is outside of the editor window, the value of the Existing parameter determines the returned values. If you set Existing to True, GetMousePos returns -1 for both the line and column, indicating that no line exists at that point. If you set Existing to False, GetMousePos returns the position of the nearest line and column.

GetPara method

```
function GetPara(ParaNum : LongInt; var Len : Word) : PAnsiChar;
```

Returns a pointer to the specified paragraph.

This method returns a pointer to the text of the paragraph specified by ParaNum. If ParaNum is invalid (i.e., less than 1 or greater than ParaCount), GetPara returns a pointer to an empty string.

The pointer returned by GetPara should be considered a pointer to a read-only string. If you need to insert, delete, or replace text in the paragraph, you must use the appropriate combinations of the SetSelection, Clear, and Insert routines. The only exception is when you are making a modification that does not change the number of bytes in the paragraph or alter the line breaks if word wrap is on. The only obvious case that meets these rules is one in which you are changing the case of one or more characters in the string.

The following example demonstrates how to use this method by calculating the current position of the caret as a byte offset into the text stream. The “+2” accounts for the implicit carriage return and line feed at the end of each paragraph.

```
var
  L : LongInt;
  I, C : Integer;
  Bytes : LongInt;
  Len : Word;
begin
  L := OvcEditor1.GetCaretPosition(C);
  OvcEditor1.LineToPara(L, C);
  Bytes := Pred(C);  {bytes into current row}
  for I := 1 to L-1 do begin
    OvcEditor1,GetPara(I, Len);
    Inc(Bytes, Len + 2);
  end;
  Label1.Caption := Format('Bytes: %d', [Bytes]);
end;
```

See also: GetLine

GetPrintableLine method

```
function GetPrintableLine(LineNum : LongInt;
  Dest : PAnsiChar; DestLen : Integer) : Integer;
```

Returns a line suitable for printing.

This method returns the string at the line specified by LineNum in a format suitable for printing (i.e., with tabs expanded to spaces). Dest is the buffer for the string, and DestLen is its size. The function result indicates the length of the null-terminated string placed in Dest.

The following example shows how to use this method. S contains the text of line 1, with all tabs expanded, and SLen contains the length of the string.

```
var
  S    : array[0..1023] of AnsiChar;
  SLen : Integer;
...
  SLen := GetPrintableLine(1, S, SizeOf(S));
```

See also: GetLine, GetPara

GetSelection method

```
function GetSelection(
  var Line1 : LongInt; var Col1 : Integer;
  var Line2 : LongInt; var Col2 : Integer) : Boolean;
```

Returns True if any text is currently selected.

If the function result is True, the bounds of the selected region are the values returned in Line1, Col1, Line2, and Col2. GetSelection is typically used in applications that provide menu items for selecting the various clipboard commands (e.g., Copy, Cut). If GetSelection returns False, these commands should be disabled.

See also: SetSelection

GetSelTextBuf method

```
function GetSelTextBuf(Buffer : PAnsiChar; BufSize: Word) : Word;
```

Obtains the currently selected text.

The GetSelTextBuf method is used to obtain the currently selected text in the editor. Buffer is filled with the selected text and the length of the copied text is returned as the function result. BufSize is the size of Buffer.

See also: GetSelTextLen

GetSelTextLen method

```
procedure GetSelTextLen : LongInt;
```

Returns the length of the currently selected text.

See also: GetSelTextBuf

GetText method

```
function GetText(P : PAnsiChar; Size : Word) : Word;
```

Returns a copy of the editor’s text stream.

This method returns the contents of the editor’s text stream in P. Size is the size of the buffer pointed to by P, including the null terminator. If Size is smaller than the editor’s text stream, P is truncated. The function result is the number of bytes copied into P. Each paragraph copied to P is terminated by a carriage return and line feed.

For Delphi 1.0, GetText is limited to handling 64KB or less.

See also: SetText

GotoMarker — method

```
procedure GotoMarker(N : Byte);
```

↳ Moves the caret to the specified bookmark.

This method moves the caret to the N'th bookmark, where N is a value in the range 0-9. If there is no N'th bookmark, GotoMarker does nothing.

See "Bookmarks" on page 769 for more information.

See also: SetMarker, SetMarkerAt

HasSelection — method

```
function HasSelection : Boolean;
```

↳ Determines whether any text in the editor's text stream is currently selected.

This method returns True if any text in the editor's text stream is currently selected, False otherwise.

HideSelection — property

```
property HideSelection : Boolean
```

Default: True

↳ Determines if highlighted text remains highlighted when the editor loses the focus.

If HideSelection is False, highlighted text remains highlighted even if the focus leaves the editor component. If HideSelection is True, highlighting is removed when the editor loses the focus and is restored when the editor regains the focus.

HighlightColors — property

```
property HighlightColors : TOvcColors
```

Default: clHighlightText and clHighlight

↳ Are the colors used to display the selected portion of the editor's text stream.

The following example shows how to change the editor's colors at run time so that highlighted text is displayed with blue text on a white background:

```
  HighlightColors.TextColor := clBlue;
  HighlightColors.BackColor := clWhite;
```

See "TOvcColors Class" on page 1152 for more information.

Insert method

```
procedure Insert(S : PAnsiChar);
```

↳ Replaces the current selection with a text string.

This method replaces the currently selected text with the contents of S. If no text is currently selected, the text in S is inserted at the current position of the caret.

The following example demonstrates how to insert “Hello World!” at the current caret position:

```
  if GetSelection(L1, C1, L2, C2) then begin
    Line := GetCaretPosition(Col);
    SetSelection(Line, Col, Line, Col);
  end;
  Insert('Hello World!');
```

See also: GetCaretPosition, InsertString, SetSelection

InsertMode property

```
property InsertMode : Boolean
```

Default: True

↳ Turns insert mode on or off.

This property turns insert mode on or off within the editor. If insert mode is on, text is inserted into the editor, if off, entered text overwrites existing text at the current caret position.

InsertString method

```
procedure InsertString(const S : string);
```

↳ Inserts a string into the editor.

InsertString inserts the string (S) into the editor at the current caret position. If any text is currently highlighted, it is replaced by the string passed to InsertString. If S is an AnsiString, if can be up to ParaLengthLimit characters in length.

See also: Insert

LeftColumn — run-time property

```
property LeftColumn: Integer
```

↳ Determines the column at the left edge of the editor.

If the editor is not scrolled horizontally, LeftColumn returns zero.

LeftMargin — property

```
property LeftMargin : Integer
```

Default: 15

↳ Determines the left margin in the editor window.

This property is used to set the width of the left margin (in pixels). If no margin is desired, assign 0 to the LeftMargin property. Note that bookmark bitmaps are not displayed if the left margin is less than the width of the bitmaps (12 pixels).

See also: RightMargin, ShowBookmarks

LeftMarginLine — property

```
property MarginLeft : TOvcEditorMargin;
```

```
TOvcEditorMargin = class(TPersistent)
protected{private}
  FSide: TOvcMarginSide;
  FLineWeight: Integer;
  FLineStyle: TPenStyle;
  FLineColor: TColor;
  FLinePosition: Integer;
  procedure SetLinePosition(Value: Integer);
  procedure SetLineWeight(Value: Integer);
  procedure SetLineStyle(Value: TPenStyle);
  procedure SetLineColor(Value: TColor);
public
  constructor Create(Side: TOvcMarginSide);
  property LinePosition: Integer
    read FLinePosition write SetLinePosition;
```

```
published
  property LineWeight: Integer
    read FLineWeight write SetLineWeight;
  property LineStyle: TPenStyle
    read FLineStyle write SetLineStyle;
  property LineColor: TColor
    read FLineColor write SetLineColor;
end;
```

↳ An instance of the TOvcEditorMargin class that defines the properties of the left margin line.

See also: LeftMarginLine, TOvcEditorMargin.

LineCount — run-time, read-only property

```
property LineCount : LongInt
```

↳ Returns the total number of lines in the editor.

If word wrap is off, the value returned by LineCount is the same as that returned by ParaCount.

See also: LineLength, Lines, ParaCount

LineLength — run-time, read-only property

```
property LineLength[LineNum : LongInt] : Integer
```

↳ Returns the length of the specified line.

This indexed property returns the length of the line specified by LineNum. The valid range for LineNum is 1 to LineCount. If LineNum is invalid, -1 is returned.

See also: LineCount, Lines

Lines — run-time, read-only property

```
property Lines[LineNum : LongInt] : string
```

↳ Returns the text of the specified line.

This indexed property returns the text of the line specified by LineNum as a string. If AnsiStrings are enabled, the returned string can be up to ParaLengthLimit in length. Otherwise, if the length of the specified line is greater than 255, it is truncated to 255 characters.

The valid range for LineNum is 1 to LineCount. If you attempt to retrieve the text of a line outside the valid range, an empty string is returned.

See also: LineCount, LineLength

LineToPara — method

```
procedure LineToPara(var L : LongInt; var C : Integer);
```

↳ Converts a line/column coordinate to a paragraph/position coordinate.

This method translates a line/column coordinate (L, C) into a paragraph/position coordinate (returned in L, C). If word wrap is off, the L and C parameters remain unchanged. The column/position value in C is a 1-based number.

LineToPara is most often used in cases where a line/column coordinate was obtained by calling GetCaretPosition or GetSelection, and you want to call GetPara to look at the text at that location.

See “Line/column vs. paragraph/offset coordinates” on page 771 for more information.

See also: GetCaretPosition, GetPara, GetSelection, ParaToLine

MarginColor — property

```
property MarginColor: TColor
```

↳ Specifies the color used to paint the margin area of the editor.

The margin is the left side gutter area where the line numbers and bookmarks appear.

MaxLength property

```
property MaxLength : LongInt
```

Default: MaxLongInt

↳ The maximum number of bytes that can be entered into the editor.

This property sets a limit on the total number of bytes (characters) of text (including CRLF characters) that can be entered.

The following example sets a total byte limit of 512KB:

```
  Editor1.MaxLength := (512*1024);
```

Modified run-time property

```
property Modified : Boolean
```

↳ Indicates whether the editor's text stream was modified since the editor last received the focus.

Assigning a value to this property turns the editor's modified flag on or off. Changing the state of the modified flag is primarily handled internally, but you can do so if you modify the text stream directly (see GetPara) or you need to clear the modified flag manually (it is done automatically when DeleteAll or TOvcTextFileEditor.SaveToFile is called).

See also: GetPara

NewStyleIndent property

```
property NewStyleIndent : Boolean
```

↳ Determines how new paragraphs will be indented.

In previous versions of Orpheus, AutoIndent would set the indent level of new paragraphs to the same level as the last line of the previous paragraph. This was inconsistent to new design standards and user expectations so this new property was created which allows the user to specify whether to use NewStyleIndent or keep the old style.

New style indentation will cause the first line of a new paragraph to be indented at the same level as the first line of the previous paragraph.

See also: AutoIndent

NextEditor — run-time, read-only property

```
property NextEditor : TOvcCustomEditor
```

↳ Returns a pointer to the next attached editor.

This method returns a pointer to the next editor component in the list of editors attached to the current editor. If there are no other editors attached to the current editor NextEditor returns a pointer to the current editor. Note that FirstEditor and PrevEditor do likewise because the list of attached editors is a circular linked list.

See also: Attach, FirstEditor, PrevEditor

OnDrawLine — property

```
property OnDrawLine : TEditorDrawLineEvent
```

```
TEditorDrawLineEvent = procedure(
  Sender : TObject; EditorCanvas : TCanvas; Rect : TRect;
  S : PAnsiChar; Len, Line, Pos, Count, HBLine, HBCol, HELine,
  HECol : Integer; var WasDrawn : Boolean) of object;
```

↳ Defines an event handler that is called when the editor is painting its client area.

Sender is the editor component. EditorCanvas is the TCanvas object that you should use to perform all painting in the editor's client area. Rect provides the boundaries of the region that the line should be painted in.

S contains the text for the entire line and Len is the length of S. Line is the number of the line being painted. Pos is the offset within S where painting should begin (Pos is 0 unless the editor window is scrolled horizontally). Count is the number of characters to be drawn (this is the number of characters that fit within the width of the windows as defined by Rect).

HBLine and HBCol define the starting position of the highlighted text. HELine and HECol define the end of the highlighted text. If no text is highlighted, HBLine, HBCol, HELine, and HECol are all 0.

OnError — event

```
property OnError : TEditorErrorEvent
```

```
TEditorErrorEvent = procedure(
  Sender : TObject; var ErrorCode : Word) of object;
```

↳ Defines an event handler that is called when an editor error occurs.

If a method is assigned to this event, an exception is not raised when an error occurs. Instead, the assigned method is called, passing the error code as the ErrorCode parameter.

Sender is the object that generated the error. ErrorCode is one of the following error codes:

Error Code	**Meaning**
oeOutOfMemory	There is not sufficient memory to perform the requested operation.
oeRegionSize	The selected text exceeds 64KB and cannot be cop ied to the clipboard.
oeTooManyParas	The limit on the number of paragraphs was exceeded.
oeCannotJoin	The paragraphs cannot be joined. Typically this occurs when joining the two paragraphs would cause the new paragraph to exceed its size limit (32KB by default).
oeTooManyBytes	The limit on the total number of bytes was exceeded.
oeParaTooLong	The limit on the length of an individual para graph was exceeded.

If no method is assigned to the OnError event, an EEditorError exception is raised when an error occurs. The exception dialog displays an abbreviated form of one of the error descriptions shown above.

OnShowStatus event

```
property OnShowStatus : TShowStatusEvent
```

```
TShowStatusEvent = procedure(
  Sender : TObject; ColNum : Integer; LineNum : LongInt) of object;
```

Defines an event handler that is called when the caret position changes.

Sender is the object that generated the event. ColNum is the new effective caret column position. LineNum is the new line number.

OnTopLineChanged event

```
property OnTopLineChanged : TTopLineChangedEvent
```

```
TTopLineChangedEvent = procedure(
  Sender : TObject; Line : LongInt) of object;
```

Defines an event handler that is called when the line at the top of the editor window changes.

Sender is the object that generated the event. Line is the new line number of the line at the top of the editor window.

OnUserCommand event

```
property OnUserCommand : TUserCommandEvent
```

```
TUserCommandEvent = procedure(
  Sender : TObject; Command : Word) of object;
```

Defines an event handler that is called when a user-defined command is entered.

The method assigned to the OnUserCommand event is called when the key sequence corresponding to one of the user-defined commands (ccUser1, ccUser2, etc.) is entered.

For example, suppose you add the <CtrlF2> key sequence to the one of the active command tables and assign it to the ccUser1 command. When <CtrlF2> is pressed, the method assigned to the OnUserCommand event is called and ccUser1 is passed in Command. Sender is the object that intercepted the key sequence.

See "TOvcCommandProcessor Class" on page 50 for additional information about user-defined commands and command tables.

ParaCount method

```
function ParaCount : LongInt;
```

Returns the current number of paragraphs in the editor's text stream.

See also: LineCount

ParaLength run-time, read-only property

```
property ParaLength[ParaNum : LongInt] : Integer
```

Returns the length of a paragraph.

This indexed property returns the length of the paragraph specified by ParaNum. The valid range for ParaNum is 1 to ParaCount. If ParaNum is outside the valid range, zero is returned for the length.

See also: GetPara, ParaCount

ParaLengthLimit property

```
property ParaLengthLimit : Integer
```

Default: MaxInt

The limit on the length of a paragraph.

Typically ParaLengthLimit is used only in cases where word wrap is always off and you want to limit the length of each line.

The following example turns word wrap off and sets the maximum length for each paragraph (line) to 127 characters:

```
WordWrap := False;
ParaLengthLimit := 127;
```

See also: ParaLimit

ParaLimit property

```
property ParaLimit : LongInt
```

Default: MaxLongInt

↳ The limit on the total number of paragraphs.

Typically ParaLimit is used only in cases where word wrap is always off and you want to limit both the number of lines that can be entered and the length of the lines.

The following example turns word wrap off, limits the total number of paragraphs to 25, and limits each paragraph (line) to 127 characters:

```
WordWrap := False;
ParaLimit := 25;
ParaLengthLimit := 127;
```

See also: ParaLengthLimit

ParaPointer run-time, read-only property

```
property ParaPointer[ParaNum : LongInt] : PAnsiChar
```

↳ Returns a pointer to a paragraph.

This indexed property returns a pointer to the paragraph specified by ParaNum. The valid range for ParaNum is 1 to ParaCount. If ParaNum is outside the valid range, a pointer to a null string is returned.

The following example retrieves a copy of the 5th paragraph:

```
P := ParaPointer[5];
StrCopy(OurBuffer, P);
```

See also: GetPara

ParaToLine method

```
procedure ParaToLine(var L : LongInt; var C : Integer);
```

↳ Converts a paragraph/position coordinate to a line/column coordinate.

This method converts a paragraph/position coordinate (L, C) into a line/column coordinate (returned in L, C). If word wrap is off, the L and C parameters remain unchanged. The column/position value in C is a 1-based number.

ParaToLine might be used, for example, to implement a specialized search routine (e.g., if you needed to allow searches based on regular expressions). This can be done by calling GetPara repeatedly until you find a match. Once the match is found, you can calculate the bounds of the text that you want to highlight in terms of the paragraph/position coordinates, call ParaToLine to translate the endpoints into the line/column coordinates, then call SetSelection to highlight the text (or use Insert to make replacements).

See “Line/column vs. paragraph/offset coordinates” on page 771 for more information.

The following example moves the caret to column 1 of the next paragraph:

```
  L := GetCaretPosition(C);
  LineToPara(L, C);
  Inc(L);
  C := 1;
  ParaToLine(L, C);
  SetCaretPosition(L, C);
```

See also: GetCaretPosition, GetPara, LineToPara, SetCaretPosition

PasteFromClipboard dynamic method

```
procedure PasteFromClipboard; dynamic;
```

↳ Copies the contents of the Windows clipboard into the editor.

This method pastes the text from the clipboard into the editor, overwriting the current selection. PasteFromClipboard does nothing if the clipboard is empty or contains data other than text. To determine whether the clipboard contains any text that can be pasted, call Clipboard.HasFormat(CF_TEXT). If the result is True, the clipboard contains text that can be pasted.

See also: Clear, CopyToClipboard, CutToClipboard

PrevEditor — run-time, read-only property

```
property PrevEditor : TOvcCustomEditor
```

↳ Returns a pointer to the previous attached editor.

This method returns a pointer to the previous editor component in the list of editors attached to the current editor. If there are no other editors attached to the current one, PrevEditor returns a pointer to the current editor, and so do FirstEditor and NextEditor, because the list of attached editors is a circular linked list.

See also: Attach, FirstEditor, NextEditor

ProcessCommand — method

```
function ProcessCommand(Cmd, CharCode : Word) : Boolean;
```

↳ Sends commands and characters to the editor.

This method allows sending commands and characters to the editor, just as if they were entered from the keyboard. Typical uses for this could include a simple macro playback facility.

If the editor processes the command, the function result is True; otherwise, it is False.

See “TOvcCommandProcessor Class” on page 50 for more information.

The following example demonstrates sending a series of commands to the editor while ignoring the function results:

```
  ProcessCommand(ccChar, Ord('A'));
  ProcessCommand(ccUp, 0);
  ProcessCommand(ccChar, Ord('B'));
```

ReadOnly — property

```
property ReadOnly : Boolean
```

Default: False

↳ Determines if the editor’s text stream can be edited.

If ReadOnly is True, the text cannot be modified. All insertion and deletion commands are ignored.

Redo dynamic method

```
procedure Redo; dynamic;
```

↳ The last undone operation.

Redo has no effect if there is no undone operation to redo.

The following example redoes all undone operations:

```
  while CanRedo do
    Redo;
```

See also: CanRedo, Undo

Replace dynamic method

```
function Replace(const S, R : string;
  Options : TSearchOptionSet) : LongInt; dynamic;

TSearchOptionSet = set of TSearchOptions;

TSearchOptions = (
  soFind, soBackward, soMatchCase, soGlobal, soReplace,
  soReplaceAll, soWholeWord, soSelText);
```

↳ Searches for a string and replaces it with another string.

This method replaces search string S with the replace string R. If S is found, the function result is a replacement count. If S is not found, the function result is -1.

The following can be specified in Options:

Value	**Meaning**
soReplace	A single replace operation is requested.
soReplaceAll	All matching text in a given range should be replaced.
soBackward	Search backward.
soMatchCase	Perform a case-sensitive search.

Value	**Meaning**
soWholeWord	A match should not occur if the search string is found embedded in another word. A word is delimited by a space, tab, or any of the following characters: ,./ ?;:`'"<>[]{}-=\+|()%@
soSelText	Search only in the currently selected text.
soGlobal	The search should start at the beginning or end of the file, depending on the state of the soBackward option, instead of at the current caret position.

The soXxx constants are defined in the OvcData unit.

The Options parameter is assumed to have either the soReplace or soReplaceAll option set (these options are mutually exclusive). It can also have any of the other options set.

If the soReplace option is set, a replacement is made only if the search string specified by S is currently selected. If it is not selected, Replace searches for the string. If it finds the string, it highlights the string and returns 0 (since the string was found, but no replacements were made). If it does not find the string, it returns -1. If the string is selected, it replaces the string with R, then searches for the next instance of the string, and selects it if it is found. Whether another instance of the string is found or not, 1 is returned because 1 replacement was made.

A subsequent call to Replace with the same parameters repeats the search and returns a result of -1. If the soReplaceAll option is set, the caret remains at its original position after all the occurrences of the string are replaced.

To understand the search options and the use of the Replace function, study the source code for the TEXTEDIT sample project and experiment with a compiled version of it.

The following example replaces every instance of ‘a’ (as a word) with ‘A’. Count is set to -1 if no instances are found, otherwise to the number of replacements that were made:

```
Count := Replace('a', 'A', [soReplaceAll, soMatchCase,
                  soGlobal, soWholeWord);
```

See also: Search

ResetScrollBars method

```
procedure ResetScrollBars(UpdateScreen : Boolean);
```

Resets the horizontal and vertical scroll bars.

ResetScrollBars adjusts the ranges, updates the positions, and redraws the horizontal and vertical scroll bars for the editor component. If UpdateScreen is True, the editor window is updated immediately, otherwise it is updated when Windows is not busy with other tasks.

ResetScrollBars is intended primarily for internal use. In the rare cases where you might call it directly, it will typically be used in conjunction with DeleteAll and AppendPara.

See also: AppendPara, DeleteAll

RightMargin property

```
property RightMargin : Integer
```

Default: 5

Determines the width of the editor's right margin.

This property is used to set the width of the right margin (in pixels). If no right margin is desired, assign 0 to the RightMargin property.

See also: LeftMargin

RightMarginLine property

```
property MarginRight: TOvcEditorMargin;

TOvcEditorMargin = class(TPersistent)
protected{private}
  FSide: TOvcMarginSide;
  FLineWeight: Integer;
  FLineStyle: TPenStyle;
  FLineColor: TColor;
  FLinePosition: Integer;
  procedure SetLinePosition(Value: Integer);
  procedure SetLineWeight(Value: Integer);
  procedure SetLineStyle(Value: TPenStyle);
  procedure SetLineColor(Value: TColor);
public
  constructor Create(Side: TOvcMarginSide);
  property LinePosition: Integer
    read FLinePosition write SetLinePosition;
published
  property LineWeight: Integer
    read FLineWeight write SetLineWeight;
  property LineStyle: TPenStyle
    read FLineStyle write SetLineStyle;
  property LineColor: TColor
    read FLineColor write SetLineColor;
end;
```

↳ An instance of the TOvcEditorMargin class that defines the properties of the right margin line.

See also: LeftMarginLine, TOvcEditorMargin

ScrollBars property

```
property ScrollBars : TScrollStyle
```

Default: ssBoth

↳ Determines whether vertical and horizontal scroll bars are displayed.

The TScrollStyle enumerated type is defined in the VCL’s Controls unit. Both vertical and horizontal scroll bars are displayed by default.

See also: ScrollBarsAlways

ScrollBarsAlways property

```
property ScrollBarsAlways : Boolean
```

Default: False

Determines if the selected scroll bars are always displayed.

Setting ScrollBarsAlways to True forces the editor to display the selected scroll bars even if they are not currently needed. Otherwise, scroll bars are displayed only if part of the text stream is not visible in the current editor window.

See also: ScrollBars

ScrollPastEnd property

```
property ScrollPastEnd : Boolean
```

Default: False

Determines if the caret is allowed to move beyond the end of a line.

If ScrollPastEnd is True, the caret can move beyond the end of the current line. If it is False, pressing <Right> when the caret is at the end of the line moves the caret to the beginning of the next line. ScrollPastEnd is ignored if word wrap is on, because this behavior is always disabled.

See also: WordWrap

Search method

```
function Search(
  const S : string; Options : TSearchOptionSet) : Boolean;
```

```
TSearchOptionSet = set of TSearchOptions;
```

```
TSearchOptions = (soFind, soBackward, soMatchCase, soGlobal,
  soReplace, soReplaceAll, soWholeWord, soSelText);
```

Searches for a string.

If the search string S is found, Search selects the matched string and positions the caret to the right of the selected text, returning True as the function result.

The following can be specified in Options:

Value	**Meaning**
soFind	A single find operation is requested.
soBackward	Search backward.
soMatchCase	Perform a case-sensitive search.
soWholeWord	A match should not occur if the search string is found embedded in another word. A word is delimited by a space, tab, or any of the following characters: ,./ ?;:`'"<>[]{}-&=\+|()%@^$#!~*
soSelText	Search only in the currently selected text.
soGlobal	The search should start at the beginning or end of the file, depending on the state of the soBackward option, instead of at the current caret position.

The soXxx constants are defined in the OvcData unit.

The Options parameter must have the soFind option set. It can also have any of the other options set.

See also: Replace

SelectAll method

```
procedure SelectAll(CaretAtEnd : Boolean);
```

Highlights all text in the editor's text stream.

If CaretAtEnd is True, the caret is placed at the end of the selected text. If CaretAtEnd is False, the caret is placed at the start of the selected text.

See also: SetSelection

SetCaretPosition method

```
procedure SetCaretPosition(Line : LongInt; Col : Integer);
```

↳ Moves the caret to the specified line and column.

Col is an actual column number, not an effective column number. See EffectiveColumn for a discussion of column numbers.

The following example moves the caret to column 1 of the last line:

```
SetCaretPosition(LineCount, 1);
```

See also: EffectiveColumn, GetCaretPosition

SetMarker method

```
procedure SetMarker(N : Byte);
```

↳ Sets a bookmark at the current caret position.

This method sets bookmark N to point to the current position of the caret.

See “Bookmarks” on page 769 for more information.

The following example demonstrates setting bookmark 1 to point to the current position of the caret:

```
SetMarker(1);
```

See also: GotoMarker, SetMarkerAt

SetMarkerAt method

```
procedure SetMarkerAt(N : Byte; Line : LongInt; Col : Integer);
```

↳ Sets a bookmark at the specified position.

This method sets bookmark N to point to the specified line and column. Col is assumed to be an actual column number, not an effective column number. See the EffectiveColumn method on page 788 for a discussion of column numbers.

See “Bookmarks” on page 769 for more information.

The following example demonstrates setting bookmark 1 to point to the first character of the last line in the editor:

```
SetMarkerAt(1, LineCount, 1);
```

See also: GotoMarker, SetMarker

SetSelection method

```
procedure SetSelection(Line1 : LongInt; Col1 : Integer;
  Line2 : LongInt; Col2 : Integer; CaretAtEnd : Boolean);
```

Selects a region of text.

After the call to SetSelection, the region from (Line1, Col1) to (Line2, Col2) is selected. The caret is positioned at (Line2, Col2) if CaretAtEnd is True, or at (Line1, Col1) if it is False. The line and column numbers are one-based.

The following example selects all text in the editor and places the caret at (1, 1):

```
  SetSelection(1, 1, LineCount, LineLength(LineCount), False);
```

See also: GetSelection, LineLength

SetSelTextBuf method

```
procedure SetSelTextBuf(Buffer : PAnsiChar);
```

Replaces the current selection with the contents of passed Buffer.

Calling SetSelTextBuf will replace any currently selected text in the editor with the contents of Buffer. If no text is highlighted, the text is inserted at the current caret location. Buffer is a null- terminated string of characters.

See also: SetText

SetText method

```
procedure SetText(P : PAnsiChar);
```

Replaces the current editor’s contents.

SetText sets the contents of the editor’s text stream to the contents of the buffer pointed to by P. This is accomplished by calling DeleteAll, Insert, and ResetScrollBars.

If P contains multiple paragraphs, separate each with a carriage return and line feed.

See also: GetText, SetSelTextBuf

ShowBookmarks property

```
property ShowBookmarks : Boolean
```

Default: True

↳ Determines if the bookmark icons are displayed.

If ShowBookmarks is True and there is sufficient room in the editor window margin, the bookmark icons are displayed. All marker functions operate regardless of the ShowBookmarks setting.

See “Bookmarks” on page 769 for more information.

See also: GotoMarker, Margin, SetMarker, SetMarkerAt

ShowLineNumbers property

```
property ShowLineNumbers : Boolean
```

Default: False

↳ Specifies whether to display line numbers in the left margin.

If ShowLineNumbers is True then the left margin’s width will be increased enough to accomodate the line numbers.

ShowRules property

```
property ShowRules : Boolean
```

Default: False

↳ Specifies whether to display a thin, horizontal line at the bottom of each line of text.

If ShowRules is True, the editor looks like a page of notebook paper.

TabSize property

```
property TabSize : Byte
```

Default: 8

↳ Determines the distance between tab stops.

See “Tabs” on page 769 for more information on tabs.

See also: TabType, WantTab

TabType property

```
property TabType : TTabType
```

```
TTabType = (ttReal, ttFixed, ttSmart);
```

Default: ttReal

↳ Determines the type of tab used by the editor.

This property selects the type of behavior that is desired when the <Tab> key is pressed. Possible values are ttReal, ttFixed, and ttSmart.

See “Tabs” on page 769 for additional information on tab types.

See also: TabSize, WantTab

TextLength run-time, read-only property

```
property TextLength : LongInt
```

↳ Returns the length of text in the editor.

This property returns the current length of text in the editor’s text stream. This value is the same as the total number of bytes in the editor’s text stream. The editor does not store CR/LF characters, so this value will be less than the length of a file containing the same text.

See also: ByteLimit

TopLine run-time property

```
property TopLine : LongInt
```

↳ Determines the line at the top of the editor window.

Assigning a value between 1 and the number of lines in the editor causes the editor to scroll the requested line to the top of the editor.

TrimWhiteSpace property

```
property TrimWhiteSpace : Boolean
```

Default: True

↳ Determines if trailing spaces are removed.

If TrimWhiteSpace is True, spaces at the end of an appended paragraph are removed prior to appending the paragraph to the editor’s text stream. If False, trailing spaces are left intact.

See also: AppendPara

Undo dynamic method

```
procedure Undo; dynamic;
```

Undoes the last editing operation.

To determine whether there is anything to be undone, call CanUndo. To undo an undo, call Redo.

The following example undoes all operations that can be undone:

```
  while CanUndo do
    Undo;
```

See also: CanUndo, Redo, UndoBufferSize

UndoBufferSize property

```
property UndoBufferSize : Word
```

Default: 8K

The size of the undo buffer.

The default size of the undo buffer is 8KB, which is more than adequate for most applications. Smaller buffers might be appropriate when the editor component is inside an entry form. Larger buffers are probably desirable in dedicated text editing applications.

Caution: Changing the buffer size destroys the contents of the existing buffer, even if the new buffer is larger than the old buffer.

Except when you want to disable the undo facility altogether (by assigning 0 to UndoBufferSize), it is not advisable to specify a buffer size less than 1KB. It is not really possible to configure the editor component to allow only a single operation to be undone. Any editing operation might create a single undo record containing as few as 16 bytes, or it might create a series of related records occupying hundreds or even thousands of bytes.

The following example disables the undo facility:

```
  UndoBufferSize := 0;
```

VisibleColumns run-time, read-only property

```
property VisibleColumns : Integer
```

Returns the number of visible columns in the editor window.

See also: WrapColumn

VisibleRows — run-time, read-only property

```
property VisibleRows : Integer
```

↳ Returns the number of visible rows in the editor window.

WantEnter — property

```
property WantEnter : Boolean
```

Default: True

↳ Determines whether the editor processes the <Enter> key.

If WantEnter is True, the editor processes the <Enter> key and creates a new paragraph. If WantEnter is False, you can still insert new paragraphs by pressing <CtrlEnter> and the <Enter> key is passed on to the parent form.

See also: WantTab

WantTab — property

```
property WantTab : Boolean
```

Default: False

↳ Determines whether the editor inserts tab characters when <Tab> is pressed.

If WantTab is True and the <Tab> key is pressed, depending on the setting of the TabType, either spaces or a tab character are inserted. If WantTab is False, <CtrlTab> is used instead of <Tab> to insert tabs into the text stream.

See “Tabs” on page 769 for more information on tabs.

See also: TabType, WantEnter

WheelDelta — property

```
property WheelDelta: Integer
```

Default: 1

↳ Defines how many lines of text will scroll on each mouse wheel click.

WheelDelta defaults to 1 but it is recommended that the developer acquire the default system setting from the registry and set this property at run time.

WordDelimiters run-time property

```
property WordDelimiters : string
```

Default: ^I#39#13#10' ,./?;:`”<>[]{}-=\+|()%@&^$#!~*';

Determines the characters that are considered to separate words in the editor.

See also: GetCurrentWord, WordWrap

WordWrap property

```
property WordWrap : Boolean
```

Default: False

Turns word wrap on or off.

Caution: A potentially time-consuming process of recalculation occurs every time you turn word wrap on or off, and every time you change the wrap column while word wrap is already on. Therefore, if word wrap is currently off, and you want to simultaneously turn word wrap on and set the wrap column, you should set WrapColumn first, then set WordWrap to True. If you set WordWrap to True first, the calculations are done twice.

The following example turns on word wrap:

```
  WordWrap := True;
```

See also: WordDelimiters, WrapAtLeft, WrapColumn

WrapAtLeft property

```
property WrapAtLeft : Boolean
```

Default: True

Determines the caret placement when <Left> is pressed while in column 1 of the editor.

If WrapAtLeft is True, the editing caret is moved to the end of the previous line if <Left> is pressed when the caret is in column 1. WrapAtLeft is ignored if word wrap is on, because this behavior is always enabled.

See also: WordWrap

WrapColumn property

```
property WrapColumn : Integer
```

Default: 80

The column for word wrap.

Changing WrapColumn has no immediate effect if WordWrap is False.

If you want to emulate the behavior of the standard Windows edit control (which bases the wrap column on the current dimensions of the editor's window), set WrapColumn to some value less than or equal to the number of visible columns. See the TEXTEDIT demo program for an example of how to implement this.

Caution: A potentially time-consuming process of recalculation occurs every time you turn word wrap on or off, and every time you change the wrap column while word wrap is already on. Therefore, if word wrap is currently off, and you want to simultaneously turn word wrap on and set the wrap column, you should set WrapColumn first, then set WordWrap to True. If you set WordWrap to True first, the calculations are done twice.

The following example sets the editor's word wrap column to one less than the number of visible columns. An ideal place to do this is in the form's OnResize event handler.

```
  WrapColumn := VisibleColumns-1;
```

See also: WrapToWindow

WrapToWindow property

```
property WrapToWindow : Boolean
```

Default: False

Determines if the wrap column is automatically calculated.

If WrapToWindow is True, the editor's WrapColumn is automatically determined based on the current width of the editor's client area.

See also: WrapColumn

TOvcCustomTextFileEditor Class

The TOvcCustomTextFileEditor class is the immediate ancestor of the TOvcTextFileEditor component. It implements all the methods and properties used by the TOvcTextFileEditor component and is identical to the TOvcTextFileEditor except that no properties are published.

TOvcCustomTextFileEditor is provided to facilitate creation of descendent text file editor components. For property and method descriptions, see “TOvcTextFileEditor Component” on page 821.

Hierarchy

TCustomControl (VCL)

  TOvcCustomControl (OvcBase) 16

    TOvcCustomControlEx (OvcBase) 19

      TOvcCustomEditor (OvcEdit) 774

        TOvcCustomTextFileEditor (OvcEdit)

TOvcTextFileEditor Component

Since a common use for the editor component is to edit the contents of an ASCII text file, a ready-made descendant of TOvcEditor, TOvcTextFileEditor, is provided. TOvcTextFileEditor provides a LoadFromFile method, which reads the contents of a text file and loads the text into the associated editor component by calling AppendPara. You can ask for a file to be read either when you create the TOvcTextFileEditor or later. To edit a different file later on, simply call LoadFromFile again.

In addition to LoadFromFile, TOvcTextFileEditor provides several additional methods and properties beyond those it inherits from TOvcEditor.

Example

This example shows how to construct a text file editor component to load and allow editing an existing text file.

Create a new project, add components, and set the property values as indicated in Table 27.4.

Table 27.4: *Text file editor example*

Component	Property	Value
OvcTextFileEditor	FileName	C:\AUTOEXEC.BAT
	IsOpen	True

Run the project and experiment with the generated program. Don't worry about the edits made, since this example does not save the file.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase) 16

❷ TOvcCustomControlEx (OvcBase) 19

TOvcCustomEditor (OvcEdit) 774

TOvcCustomTextFileEditor (OvcEdit). 820

TOvcTextFileEditor (OvcEdit)

Properties

❶	About	❷	Controller	❶	LabelInfo
❶	AttachedLabel		FileName		MakeBackup
	BackupExt		IsOpen		

Methods

LoadFromFile	NewFile	SaveToFile

Events

❶	AfterEnter	❶	AfterExit	❶	OnMouseWheel

Reference Section

BackupExt property

```
property BackupExt : TextStr
```

```
TextStr = string[3];
```

Default: “BAK”

The extension used when creating backup files.

This property determines the file extension used when creating a backup file for the file currently being edited.

See also: MakeBackup

FileName property

```
property FileName : string
```

Default: Empty string

The name of the edit file.

FileName is the file in the editor or the name of a file that will be loaded from disk. Changing FileName does not cause the file to be loaded, except at design time.

The following example loads a file into the text file editor:

```
  OvcTextFilerEditor.FileName := 'C:\AUTOEXEC.BAT';
  OvcTextFilerEditor.IsOpen := True;
```

See also: IsOpen

IsOpen property

```
property IsOpen : Boolean
```

Default: False

Indicates whether the file is open.

The IsOpen property determines if the file specified by the FileName property is open (i.e., being edited). Setting IsOpen to True causes the file specified by the FileName property to be loaded using the LoadFromFile method, deleting any text currently in the editor. Setting IsOpen to False deletes any existing text in the editor's text stream and resets the scroll bars. File errors generate standard VCL exceptions.

Caution: IsOpen does not check to see if the existing text was modified before deleting it. You must test the Modified property and save the file if necessary to prevent editing changes from being lost.

LoadFromFile dynamic method

```
procedure LoadFromFile(const Name : string); dynamic;
```

Loads a file into the text editor.

This method loads the entire contents of the file specified by Name into memory and redraws the editor component. If an error occurs, an appropriate exception is raised (see OnError). The FileName property is changed to Name and IsOpen is set to True.

The following example loads a file into the editor:

```
  OvcTextFilerEditor.LoadFromFile('C:\AUTOEXEC.BAT');
```

See also: FileName, IsOpen, SaveToFile

MakeBackup property

```
property MakeBackup : Boolean
```

Default: False

Determines whether the text editor makes a backup file.

This property determines if the text file editor creates a backup file before saving the current editor's text. If MakeBackup is True, the existing disk file is renamed using BackupExt as the extension before the current contents of the editor are saved to disk.

See also: BackupExt

NewFile method

```
procedure NewFile(const Name : string);
```

Creates a new file.

NewFile does three things:

1. Calls DeleteAll to delete all existing text.
2. Calls ResetScrollBars to reset the scroll bars.
3. Assigns Name to the FileName property.

Although NewFile is typically called with an empty string as a parameter, it can also be called in lieu of LoadFromFile to edit a file that does not yet exist.

Caution: NewFile does not check to see if the existing text was modified before deleting it. You must test the Modified property and save the file if necessary to prevent editing changes from being lost.

The following example creates a new file if it does not already exist:

```
if (Length(SomeFileName) <> 0) and FileExists(SomeFileName) then
  OvcTextFilerEditor.LoadFromFile(SomeFileName)
else
  OvcTextFilerEditor.NewFile(SomeFileName);
```

See also: LoadFromFile, SaveToFile, TOvcEditor.Modified

SaveToFile dynamic method

```
procedure SaveToFile(const Name : string); dynamic;
```

Saves the editor’s text stream to a file.

This method saves the contents of the editor’s text stream in the file specified by Name, then sets the Modified property to False.

If MakeBackup is True, SaveToFile creates a backup file (if appropriate) by renaming the existing file before saving. For example, if Name is SOMEFILE.TXT and a file by that name already exists, the existing file is renamed to SOMEFILE.BAK before the new SOMEFILE.TXT is created. If a SOMEFILE.BAK already exists, it is deleted. This assumes that BackupExt is set to its default of “BAK.”

The following example saves the existing file using its current file name and then prepares the editor for a new file:

```
if OvcTextFilerEditor.Modified then
  OvcTextFilerEditor.SaveToFile(FileName);
OvcTextFilerEditor.NewFile('');
```

See also: BackupExt, LoadFromFile, MakeBackup

TOvcDbEditor Component

TOvcDbEditor is a direct descendant of the TOvcCustomEditor and inherits all of its properties and methods (these are the same properties and methods documented in the TOvcEditor).

There are very few differences in the behavior of the TOvcDbEditor and TOvcEditor except that TOvcDbEditor is capable of connecting to a data source and editing the text of a memo field.

TOvcDbEditor provides four additional properties. AutoUpdate allows you to configure the editor to automatically update the data source whenever the editor loses the focus. The DataField, DataSource, and Field properties are exactly the same as the like-named properties in standard data-aware components.

Using a TOvcDbEditor component is very simple—assign a TDataSource to the DataSource property and assign a memo field name to the DataField property. The editor automatically gets the text to edit from the data field and optionally updates the data source with the changed data.

The TOvcDbEditor is limited to 32K of text, unlike its ancestor, which is limited to 16M of text.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase) 16

❷ TOvcCustomControlEx (OvcBase) 19

TOvcCustomEditor (OvcEdit) 774

TOvcDbEditor (OvcDbEd)

Properties

❶ About	❷ Controller	Field
❶ AttachedLabel	DataField	❶ LabelInfo
AutoUpdate	DataSource	

Events

❶ AfterEnter	❶ AfterExit	❶ OnMouseWheel

Reference Section

AutoUpdate — property

```
property AutoUpdate : Boolean
```

Default: True

Determines whether the data source is updated when the editor loses the focus.

If AutoUpdate is True, the editor component automatically calls UpdateRecord for the attached DataSource when it loses the focus. Set it to False if you update the data source explicitly.

DataField — property

```
property DataField : string
```

Identifies the field (in the data source component) from which the editor component displays data.

DataSource — property

```
property DataSource : TDataSource
```

Specifies the data source component where the editor obtains the data to display.

Field — run-time, read-only property

```
property Field : TField
```

Returns the TField object to which the editor component is linked.

Use the Field object when you want to change the value of the data in the field programmatically.

Chapter 28: File and Text Viewers

Orpheus provides two viewer components designed to give your application data viewing capabilities.

The viewer components were designed primarily for viewing (or browsing) ASCII text stored in a file or list, although they can be used for other purposes as well. The abstract viewer class (TOvcBaseViewer) knows only how many items of text there are to display. Your application is responsible for returning a null-delimited string that represents a particular item. In most cases, however, the viewer class will probably be used by one of its descendent components, TOvcTextFileViewer or TOvcFileViewer, which automatically respond to requests for strings.

The viewer components allow you to browse text or binary files. While viewing a file, you can use most of the common caret movement commands to move around in the file and to select text. Selected text can be copied to the Windows clipboard for use by another application, but of course the standard cut and paste commands are not available.

A search function is provided. The application must provide for initiating the search and specifying search options. The available search options include backwards search, case-sensitive search, and search from the beginning or end of the file.

Text markers (bookmarks) provide a means of moving quickly from place to place in the file. You can set up to ten text markers at different locations in a file.

The viewer components work only with fixed pitch fonts and do not support multi-font viewing.

Three varieties of viewer are supplied. The abstract viewer class (TOvcBaseViewer) provides a base for the common functions for the text file viewer and the general file viewer. The text file viewer (TOvcTextFileViewer) reads a complete text file into memory in a linked list of lines. The general file viewer (TOvcFileViewer) uses a buffering scheme to display files of any size in ASCII or hex mode.

TOvcBaseViewer Class

The TOvcBaseViewer is an abstract class, so you will never instantiate an instance of this class. You must either derive your own class from TOvcBaseViewer, or use one of the other two classes derived from it (TOvcTextFileViewer or TOvcFileViewer). In most cases, you will probably take the latter course.

When do you need to derive a new class from TOvcBaseViewer? When you want to use a viewer component to display textual data that is not stored in a file, or when you want to display only part of a file rather than the entire file, or when the data in the file must be formatted before it can be displayed. For example, suppose that you want to use a viewer component to display the contents of a binary data file, and that the data is arranged into fixed- length records. Neither TOvcTextFileViewer nor TOvcFileViewer fit your needs, so you need to implement a new component derived from TOvcBaseViewer that does exactly what you want.

As far as the viewer component itself is concerned, there are only two things you must do in your TOvcBaseViewer descendant:

1. Override the GetLinePtr method, which returns a pointer to a string representation of a given line.
2. Set the LineCount property to tell the viewer component how many lines to display.

Of course, in order to implement your GetLinePtr routine, you also need to implement a storage, or retrieval mechanism to fetch the data for a given line and do whatever is necessary to convert that data to a displayable form.

The design and implementation of the storage/retrieval mechanism is entirely your responsibility. TOvcBaseViewer will neither help nor hinder you. But whatever you do, you need to do it fast. If you're reading data from disk in your GetLinePtr routine, you should design a buffering scheme to prevent excessive disk reads. If you're storing the data in a linked list, develop a scheme to minimize the distance that you search to locate a given node in the list (see the implementation of TOvcTextFileViewer's GetLinePtr routine). If you're formatting binary data, develop a scheme for buffering copies of pre-formatted strings to avoid formatting the same data repeatedly. The bottom line is that if your GetLinePtr routine can't respond rapidly to requests for text, the viewer component will perform sluggishly.

The remainder of this section describes capabilities provided by the TOvcBaseViewer class and inherited by the TOvcFileViewer, TOvcTextFileViewer, or your class derived from TOvcBaseViewer.

Searching for text

The viewer components offer no built-in command to allow the user to search for text, since doing so would require it to display a form that might conflict with the overall look-and-feel of the parent application. It does, however, provide a Search method that you can call to search for text. It is your responsibility to provide the means for initiating the search, and for displaying a form in which the user can enter search criteria.

The Search method takes two parameters, a search string and a set of search options. The available search options are shown in Table 28.1.

Table 28.1: *Available search options*

Option	**Description**
soFind	A single find operation is requested.
soBackward	Search backward.
soMatchCase	Perform a case-sensitive search.
soGlobal	The search should start at the beginning or end of the file, depending on the state of the soBackward option, instead of at the current caret position.

The soXxx constants are defined in the OvcData unit, so you must add OvcData to your unit's uses clause if appropriate.

When calling the Search method, you must always select the soFind option. You can also select one or more of the other options. If the search string is found, Search highlights the string, positions the caret at the end of the string, and returns True.

The TEXTVIEW and FILEVIEW sample programs contain examples of how to use the Search method in an application. The implementation of the search facility in these two programs is typical of most Windows applications that allow the user to search for text.

Bookmarks

Bookmarks provide a means of moving quickly from place to place within a file. Assume that you are primarily interested in two sections of a text file that are widely separated, and want to be able to move between the two of them quickly. This can be accomplished by setting a bookmark at the beginning of each section, and jumping between them with the GotoMarker command.

To set a bookmark using the default key assignments (see "TOvcCommandProcessor Class" on page 50), press <ShiftCtrlN>, where N is a number in the range 0 to 9. The current position of the caret is marked as bookmark N. To return to that spot later, press <CtrlN>, where N is the number used to set the bookmark.

You might also want to have a menu item that sets a bookmark at the current position. You can call either the SetMarker method, which sets a bookmark at the current position of the caret, or the SetMarkerAt method, which sets a bookmark at a specified line and column position.

Tab expansion

The viewer component is designed to automatically expand tab characters to spaces when text is displayed on screen. This behavior, which is enabled by default, is controlled by the ExpandTabs property. If ExpandTabs is True, tabs are expanded. If it is False, tabs are treated as ordinary characters (their appearance on the screen depends on the font).

The distance between tab stops depends on the current tab size, which is determined by the value of the TabSize property (the default setting is 8). You can change the tab size by assigning a new value to the TabSize property, which changes the tab size and redraws the viewer component's window to reflect the change.

Copying

The Orpheus viewer components provide support for copying to the Windows clipboard. You can copy text using either the standard Windows 3.0 command, <CtrlIns>, or the newer Windows command, <CtrlC>, depending on which command processor tables are active in the Controller's attached TOvcCommandProcessor. See "TOvcCommandProcessor Class" on page 50 for more information.

For examples of using the CopyToClipboard method, see the source code for the TEXTVIEW and FILEVIEW sample programs.

Actual columns vs. effective columns

In various places in the viewer component, a distinction is made between "actual columns" and "effective columns." For example, when you call the GetCaretPosition method, it returns the location of the caret in line/column coordinates and the column coordinate is an actual column number. By contrast, CaretEffectivePos returns a structure containing the current line and column position of the caret, but the column coordinate is an effective column number. So what's the distinction?

The actual column number indicates the position of the caret within the string, without regard for tab expansion. If the cursor is on the fourth character in the current line, the actual column number is 4.

The effective column number is the screen column at which the caret would be placed given the contents of the string, the current tab size, the state of the ExpandTabs property, and the actual column number. For example, suppose that tab expansion is on, the tab size is 8, the actual column is 2, and the string consists of a tab character followed by an 'A'. The effective column would be 9 in this case, since the second character in the string is 'A', and 'A' would be displayed on screen at column 9. If tab expansion were off, the effective column would be 2. Note that if tab expansion is off, or if the current line contains no tab characters, the actual column and the effective column are the same.

Displaying status information

When using a viewer component, it is often necessary to continuously display status information (e.g., caret position, total number of lines). To facilitate this, the viewer component calls the method assigned to the OnShowStatus event every time the caret is moved or the number of lines changes. By assigning an OnShowStatus event handler, you can update the region of the screen where you display status information.

An OnShowStatus event is called with two parameters: the number of the current line, and the number of the current column. The column value is an effective column number. The TEXTVIEW and FILEVIEW sample programs contain examples.

When you are displaying status information for a TOvcFileViewer, remember that the total number of lines in the file is not known initially. If LineCount returns FFFFFFF, then the total number of lines is unknown. You can handle this situation however you want. The FILEVIEW sample program handles it by displaying "Total: ?" until the actual number of lines is known.

Printing the text in a viewer

Neither TOvcBaseViewer nor the viewer components offer any direct means of printing the text displayed by the viewer. TOvcBaseViewer does, however, provide a method you can call to get the text of a line in a form suitable for printing (i.e., with tabs expanded to spaces). GetPrintableLine loads a printable version of the specified line into a buffer that you provide, and returns the length of the string in the buffer. The TEXTVIEW and FILEVIEW sample programs both demonstrate how to print the contents of a viewer component.

Viewer commands

The commands in Table 28.2 are available in any Orpheus viewer. The commands and key assignments described here are available when you use the “Default” or “WordStar” command tables (several command tables can be used at the same time). See “TOvcCommandProcessor Class” on page 50 for information on how to customize the command tables.

Table 28.2: *Viewer commands*

Command	**Default**	**WordStar**	**Description**
ccBotOfPage	<CtrlPgDn>	Not assigned.	Move the caret to the bottom of the window.
ccCopy	<CtrlIns>, <CtrlC>	Not assigned.	Copy the selected text to the clipboard.
ccDown	<Down>	<CtrlX>	Move the caret down to the next line.
ccEnd	<End>	<CtrlQ><D>	Move the caret to the end of the current line.
ccExtBotOfPage	<CtrlShiftPgDn>	Not assigned.	Extend the selection to the bottom of the page (the column remains unchanged).
ccExtendDown	<ShiftDown>	Not assigned.	Extend the selection down one line.
ccExtendEnd	<ShiftEnd>	Not assigned.	Extend the selection to the end of the current line.
ccExtendHome	<ShiftHome>	Not assigned.	Extend the selection to the start of the current line.
ccExtendLeft	<ShiftLeft>	Not assigned.	Extend the selection to the left one character.
ccExtendPgDn	<ShiftPgDn>	Not assigned.	Extend the selection down one page.
ccExtendPgUp	<ShiftPgUp>	Not assigned.	Extend the selection up one page.

Table 28.2: *Viewer commands (continued)*

Command	Default	WordStar	Description
ccExtendRight	<ShiftRight>	Not assigned.	Extend the selection to the right one character.
ccExtendUp	<ShiftUp>	Not assigned.	Extend the selection up one line.
ccExtFirstPage	<CtrlShiftHome>	Not assigned.	Extend the selection to the first line.
ccExtLastPage	<CtrlShiftEnd>	Not assigned.	Extend the selection to the last line.
ccExtTopOfPage	<CtrlShiftPgUp>	Not assigned.	Extend the selection to the top of the page (the column remains unchanged).
ccExtWordLeft	<CtrlShiftLeft>	Not assigned.	Extend the selection to the left one word.
ccExtWordRight	<CtrlShiftRight>	Not. assigned	Extend the selection to the right one word.
ccFirstPage	<CtrlHome>	Not assigned.	Move the caret to the first character in the viewer’s text stream.
ccGotoMarker0-9	<Ctrl0-9>	<CtrlQ> <0-9>	Move to the previously set marker (0-9).
ccHome	<Home>	<CtrlQ><S>	Move the caret to the beginning of the line.
ccLastPage	<CtrlEnd>	Not assigned.	Move the caret to the last character in the viewer’s text stream.
ccLeft	<Left>	<CtrlS>	Move the caret left one character.
ccNextPage	<PgDn>	<CtrlC>	Move the caret to the next page.
ccPrevPage	<PgUp>	<CtrlR>	Move the caret to the previous page.

Table 28.2: *Viewer commands (continued)*

Command	**Default**	**WordStar**	**Description**
`ccRight`	`<Right>`	`<CtrlD>`	`Move the caret right one character.`
`ccScrollDown`	`not assigned`	`<CtrlZ>`	`Scroll down one line.`
`ccScrollUp`	`not assigned`	`<CtrlW>`	`Scroll up one line.`
`ccSetMarker0-9`	`<CtrlShift0-9>`	`<CtrlK><0-9>`	`Set the position of marker (0-9) to the current caret position.`
`ccTopOfPage`	`<CtrlPgUp>`	`Not assigned.`	`Move the caret to the top of the window.`
`ccUp`	`<Up>`	`<CtrlE>`	`Move the caret up to the previous line.`

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase) 16

❷ TOvcCustomControlEx (OvcBase) 19

TOvcBaseViewer (OvcViewr)

Properties

❶ About	❷ Controller	Lines
❶ AttachedLabel	ExpandTabs	MarginColor
Borders	FixedFont	ScrollBars
BorderStyle	HighlightColors	ShowBookmarks
Caret	❶ LabelInfo	ShowCaret
CaretActualPos	LineCount	TabSize
CaretEffectivePos	LineLength	TopLine

Methods

ActualColumn	GetLine	Search
CheckLine	GetLinePtr	SelectAll
ClearMarker	GetMarkerPosition	SetCaretPosition
CopyToClipboard	GetPrintableLine	SetMarker
EffectiveColumn	GetSelection	SetMarkerAt
GetCaretPosition	GotoMarker	SetSelection

Events

❶ AfterEnter	❶ OnMouseWheel	OnTopLineChanged
❶ AfterExit	OnShowStatus	OnUserCommand

Reference Section

ActualColumn — method

```
function ActualColumn(
  Line : LongInt; EffectiveCol : Integer) : Integer;
```

Converts an effective column to an actual column.

Given a Line number and an effective column position, this method returns the actual column position. See “Actual columns vs. effective columns” on page 832.

See also: EffectiveColumn

Borders — property

```
property Borders : TOvcBorders
```

Default: All internal borders are off by default

An implementation of the TOvcBorders class.

By default, all of the optional borders are disabled and the component is bordered by a standard Windows bevel.

BorderStyle — property

```
property BorderStyle : TBorderStyle
```

Default: bsSingle

Determines the style used when drawing the border.

The possible values are bsSingle (a single line border is drawn around the component) or bsNone (no border line).

The TBorderStyle type is defined in the VCL’s Controls unit.

Caret — property

```
property Caret : TOvcCaret
```

Default: vertical line caret

Determines the characteristics of the viewing caret.

This property allows you to customize the attributes of the viewer’s caret. For additional information, see “TOvcCaret Class” on page 1147.

CaretActualPos run-time property

```
property CaretActualPos : TOvcTextPos

TOvcTextPos = record
  Line : LongInt; Col : Integer;
end;
```

Determines the actual caret position.

TOvcTextPos.Line is the line the caret is positioned on and TOvcTextPos.Col is the actual column the caret is positioned on. The actual column number indicates the position of the caret within the string, without regard for tab expansion.

See “Actual columns vs. effective columns” on page 832 for more information.

See also: CaretEffectivePos

CaretEffectivePos run-time property

```
property CaretEffectivePos : TOvcTextPos

TOvcTextPos = record
  Line : LongInt; Col : Integer;
end;
```

Determines the effective caret position.

TOvcTextPos.Line is the line the caret is positioned on and TOvcTextPos.Col is the effective column the caret is positioned on. The effective column number is the screen column at which the cursor would be placed given the contents of the string, the current TabSize, the state of the ExpandTabs property, and the actual column number.

See also: CaretActualPos, ExpandTabs, TabSize

CheckLine method

```
function CheckLine(LineNum : LongInt) : LongInt;
```

Validates the passed line number.

The CheckLine method is used to verify that a line number is valid. If LineNum is valid, the function result is the same as LineNum. Otherwise, it is a valid line number. Passing MaxLongInt as LineNum forces the viewer to read all lines of the file (if it hasn’t already been done) to determine the last valid line number. This is useful when you need to know how many lines are in the file.

ClearMarker method

```
procedure ClearMarker(N : Byte);
```

↳ Clears the specified bookmark.

This method removes the bookmark specified by N. If the bookmark was displayed, the viewer is updated to erase the bookmark bitmap. Valid values for N are 0 to 9.

If the bookmark specified by N has not been set, no action is taken.

See also: SetMarker, SetMarkerAt, ShowBookmarks

CopyToClipboard method

```
procedure CopyToClipboard;
```

↳ Copies the selection to the Windows clipboard.

This method copies the currently selected text to the clipboard. If there is no selection, CopyToClipboard does nothing.

See also: GetSelection

EffectiveColumn method

```
function EffectiveColumn(
  Line : LongInt; ActualCol : Integer) : Integer;
```

↳ Converts an actual column to an effective column.

Given a Line number and an actual column position, this method returns the effective column position. See “Actual columns vs. effective columns” on page 832 for more information.

See also: ActualColumn

ExpandTabs property

```
property ExpandTabs : Boolean
```

Default: True

↳ Determines whether tab characters are expanded to spaces.

If this property is True, all tab characters are expanded to the number of spaces specified by the TabSize property. If ExpandTabs is False, tab characters are treated as ordinary characters (their appearance on screen depends on the font).

See “Tab expansion” on page 832 for more information.

See also: TabSize

FixedFont property

```
property FixedFont : TOvcFixedFont
```

Default: FixedSys

↳ Determines which font the viewer uses.

FixedFont performs the same function as the standard Font property, except that it limits the displayed fonts (in the Name sub-property) to only fixed-pitch fonts. This is because the viewer components support only fixed-pitch fonts.

See “TOvcFixedFont Class” on page 1154 for more information.

GetCaretPosition method

```
function GetCaretPosition(var Col : Integer) : LongInt;
```

↳ Returns the current position of the caret.

This method returns the current line number as the function result and the current column number in Col. The value returned in Col is an actual column number, not an effective column number. See “Actual columns vs. effective columns” on page 832 for more information.

See also: CaretActualPos, CaretEffectivePos, SetCaretPosition

GetLine — method

```
function GetLine(LineNum : LongInt;
  Dest : PAnsiChar; DestLen : Integer) : PAnsiChar;
```

↳ Returns the specified line.

GetLine returns the text of the line specified by LineNum. Dest is the buffer for the string and DestLen is its size (including the terminating null character). A pointer to Dest is returned as the function result. If the requested line is larger than DestLen, only DestLen-1 characters are returned.

The following example obtains a copy of the fifth line in the viewer in Buf and a pointer to Buf in P. Line numbers in the viewer are zero-based, so the fifth line is line number is 4.

```
  var
    Buf : array[0..100] of AnsiChar;
    P   : PAnsiChar;
  ...
    P := GetLine(4, Buf, SizeOf(Buf));
```

See also: Lines

GetLinePtr — abstract method

```
function GetLinePtr(
  LineNum : LongInt; var Len : Integer) : PAnsiChar; abstract;
```

↳ Returns a pointer to the specified line.

This abstract method must be overridden in descendent classes to return a pointer to a null-terminated string corresponding to LineNum. The TOvcFileViewer and TOvcTextFileViewer components override this method, so if you use them, you don't have to override it.

GetMarkerPosition — method

```
function GetMarkerPosition(N : Byte; var Col : Integer) : LongInt;
```

↳ Returns the position of the specified bookmark.

This method returns the position of the bookmark specified by N. The column position is returned in Col and the line is returned as the function result. If the N'th bookmark is not set, GetMarkerPosition returns -1 for the function result. Valid values for N are 0 through 9.

The following example sets a bookmark at the current caret position and then retrieves its position, storing it in R and C:

```
var
  R : LongInt;
  C : Integer;
...
SetMarker(3);
R := GetMarkerPosition(3, C);
```

See also: SetMarker

GetPrintableLine method

```
function GetPrintableLine(LineNum : LongInt;
  Dest : PAnsiChar; DestLen : Integer) : Integer;
```

↳ Returns a line suitable for printing.

This method returns the string at the line specified by LineNum in a format suitable for printing (i.e., with tabs expanded to spaces). Dest is the buffer for the string, and DestLen is its size. The function result indicates the length of the null-terminated string placed in Dest.

The following example shows how to use this method. S contains the text of line 1, with all tabs expanded, and SLen contains the length of the string:

```
var
  S    : array[0..1024] of AnsiChar;
  SLen : Integer;
...
  SLen := GetPrintableLine(1, S, SizeOf(S));
```

See also: Lines

GetSelection method

```
function GetSelection(
  var Line1 : LongInt; = var Col1 : Integer;
  var Line2 : LongInt; var Col2 : Integer) : Boolean;
```

↳ Returns True if any text is currently selected.

If the function result is True, the bounds of the selected region are the values returned in Line1, Col1, Line2, and Col2. Col1 and Col2 are actual column numbers. If False, there is no selected text in the viewer.

GetSelection is typically used in applications that provide menu items for selecting the various clipboard commands (e.g., Copy). If GetSelection returns False, these commands should be disabled.

See also: SetSelection

GotoMarker method

```
procedure GotoMarker(N : Byte);
```

↳ Moves the caret to the specified bookmark.

This method moves the caret to the bookmark, where N is a value in the range 0–9. If there is no bookmark, GotoMarker does nothing.

See “Bookmarks” on page 831 for more information.

See also: SetMarker, SetMarkerAt

HighlightColors property

```
property HighlightColors : TOvcColors
```

Default: clHighlightText and clWindow

↳ Are the colors used to display the selected portion of the viewer’s text stream.

See “TOvcColors Class” on page 1152 for more information.

The following example shows how to change the viewer’s colors at run time so that highlighted text is displayed with blue text on a white background:

```
  HighlightColors.TextColor := clBlue;
  HighlightColors.BackColor := clWhite;
```

LineCount run-time property

```
property LineCount : LongInt
```

↳ Determines the number of lines in the viewer.

In descendent components, this property must be set to tell the viewer how many lines of text there are to be viewed. TOvcFileViewer and TOvcTextFileViewer do this for you.

LineLength — run-time, read-only property

```
property LineLength[LineNum : LongInt] : Integer
```

Returns the length of the specified line.

This indexed property returns the length (tabs are not expanded) of the line specified by LineNum. The Valid range for LineNum is 0 to LineCount-1. If LineNum is invalid, -1 is returned.

See also: LineCount

Lines — run-time, read-only property

```
property Lines[LineNum : LongInt] : string
```

Returns the text of the specified line.

This indexed property returns the text of the line specified by LineNum in a format suitable for printing (i.e., with tabs expanded to spaces). If the length of the specified line is greater than 255, it is truncated to 255 characters. If the viewer contains lines longer than 255, use the GetPrintableLine method instead. The Valid range for LineNum is 0 to LineCount-1. If LineNum is invalid, an empty string is returned.

MarginColor — property

```
property MarginColor : TColor
```

Default: clWindow

Determines the color of the margin area at the left of the viewer.

This property determines the color used to paint the blank area to the left of the first character of each line in the viewer window.

OnShowStatus — event

```
property OnShowStatus : TShowStatusEvent
```

```
TShowStatusEvent = procedure(
  Sender : TObject; LineNum : LongInt; ColNum : Integer) of object;
```

Defines an event handler that is called when the caret position changes.

Sender is the object that generated the event. ColNum is the new effective caret column position. LineNum is the new line number. See the TEXTVIEW or FILEVIEW example programs for examples of status routines that use an OnShowStatus event handler.

OnTopLineChanged event

```
property OnTopLineChanged : TTopLineChangedEvent
```

```
TTopLineChangedEvent = procedure(
  Sender : TObject; Line : LongInt) of object;
```

↳ Defines an event handler that is called when the line at the top of the viewer window changes.

Sender is the object that generated the event. Line is the new line number of the line at the top of the viewer window.

OnUserCommand event

```
property OnUserCommand : TUserCommandEvent
```

```
TUserCommandEvent = procedure(
  Sender : TObject; Command : Word) of object;
```

↳ Defines an event handler that is called when a user-defined command is entered.

The method assigned to the OnUserCommand event is called when the key sequence corresponding to one of the user-defined commands (ccUser1, ccUser2, etc.) is entered.

For example, suppose you add the <CtrlF2> key sequence to the one of the active command tables and assign it to the ccUser1 command. When <CtrlF2> is pressed, the method assigned to the OnUserCommand event is called and ccUser1 is passed in Command. Sender is the object that intercepted the key sequence.

See “TOvcCommandProcessor Class” on page 50 for additional information about user-defined commands and command tables.

ScrollBars property

```
property ScrollBars : TScrollStyle
```

Default: ssBoth

↳ Determines whether vertical and horizontal scroll bars are displayed.

The TScrollStyle type is defined in the VCL’s Controls unit. Both vertical and horizontal scroll bars are displayed by default.

Search virtual method

```
function Search(
  const S : string; Options : TSearchOptionSet) : Boolean;
```

```
TSearchOptionSet = set of TSearchOptions;
```

```
TSearchOptions = (
  soFind, soBackward, soMatchCase, soGlobal,
  soReplace, soReplaceAll, soWholeWord, soSelText);
```

Searches for a string.

If the search string S is found, Search selects the matched string and positions the caret to the right of the selected text, returning True as the function result.

The following can be specified in Options:

Value	Meaning
soFind	A single find operation is requested.
soBackward	Search backward.
soMatchCase	Perform a case-sensitive search.
soGlobal	The search should start at the beginning or end of the file, depending on the state of the soBackward option, instead of at the current caret position.

The soXxx constants are defined in the OvcData unit. Any option that is not listed in the table above is ignored. The Options parameter must have the soFind option set. It can also have any of the other options set.

SelectAll method

```
procedure SelectAll(CaretAtEnd : Boolean);
```

Selects all text in the viewer.

After the call to SelectAll, all of the viewer's text is selected. The caret is positioned at the end of the selection if CaretAtEnd is True, or at (0, 0) if it is *False.*

The following example selects all text in the viewer and places the caret at (0,0):

```
  SelectAll(False);
```

See also: GetSelection, SetSelection

SetCaretPosition method

```
procedure SetCaretPosition(Line : LongInt; Col : Integer);
```

↳ Moves the caret to the specified line and column.

Col is an actual column number, not an effective column number. See “Actual columns vs. effective columns” on page 832 for more information.

The following example moves the caret to column 1 of the last line:

```
  SetCaretPosition(LineCount-1, 0);
```

See also: GetCaretPosition

SetMarker method

```
procedure SetMarker(N : Byte);
```

↳ Sets a bookmark at the current caret position.

This method sets bookmark N to point to the current position of the caret.

See “Bookmarks” on page 831 for more information.

The following example demonstrates setting bookmark 1 to point to the current position of the caret:

```
  SetMarker(1);
```

See also: ClearMarker, GetMarkerPosition, GotoMarker, SetMarkerAt, ShowBookmarks

SetMarkerAt method

```
procedure SetMarkerAt(N : Byte; Line : LongInt; Col : Integer);
```

↳ Sets a bookmark at the specified position.

This method sets bookmark N to point to the specified line and column. Col is assumed to be an actual column number, not an effective column number. See “Actual columns vs. effective columns” on page 832 for more information.

See “Bookmarks” on page 831 for more information.

The following example demonstrates setting bookmark 1 to point to the first character of the last line in the editor:

```
  SetMarkerAt(1, LineCount-1, 0);
```

See also: ClearMarker, GetMarkerPosition, GotoMarker, SetMarker, ShowBookmarks

SetSelection method

```
procedure SetSelection(Line1 : LongInt; Col1 : Integer;
  Line2 : LongInt; Col2 : Integer; CaretAtEnd : Boolean);
```

Selects a region of text.

After the call to SetSelection, the region from (Line1,Col1) to (Line2,Col2) is selected. The caret is positioned at (Line2,Col2) if CaretAtEnd is True, or at (Line1,Col1) if it is False. Col1 and Col2 are actual column numbers.

The following example selects all text in the viewer and places the caret at (0,0):

```
  SetSelection(0, 0, LineCount-1, LineLength(LineCount)-1, False);
```

See also: GetSelection, LineLength

ShowBookmarks property

```
property ShowBookmarks : Boolean
```

Default: True

Determines if the bookmark icons are displayed.

If ShowBookmarks is True, the bookmark icons are displayed. All bookmark functions operate regardless of the ShowBookmarks setting.

See “Bookmarks” on page 831 for more information.

See also: ClearMarker, GotoMarker, SetMarker, SetMarkerAt

ShowCaret property

```
property ShowCaret : Boolean
```

Default: True

Determines is the editing caret is displayed in the viewer.

If this property is True, the caret is displayed when the viewer has the focus. If False, the caret is not displayed and the behavior of the cursor movement commands is altered so that the viewer is scrolled instead of causing the caret to move.

TabSize property

```
property TabSize : Byte
```

Default: 8

↳ The number of spaces substituted for tab characters.

This property determines the number of spaces used to replace tab characters within the viewer's text if the ExpandTabs property is True. See "Tab expansion" on page 832 for more information.

See also: ExpandTabs

TopLine run-time property

```
property TopLine : LongInt
```

↳ Determines the line at the top of the viewer window.

Assigning a value between 0 and LineCount-1 to this property causes the viewer to redraw itself with that line at the top of the window.

See also: LineCount

TOvcCustomTextFileViewer Class

The TOvcCustomTextFileViewer class is the immediate ancestor of the TOvcTextFileViewer component. It implements all the methods and properties used by the TOvcTextFileViewer component and is identical to the TOvcTextFileViewer except that no properties are published.

TOvcCustomTextFileViewer is provided to facilitate creation of descendent viewer components. For property and method descriptions, see "TOvcTextFileViewer Component" on page 852.

Hierarchy

TCustomControl (VCL)

  TOvcCustomControl (OvcBase) 16

    TOvcCustomControlEx (OvcBase) 19

      TOvcBaseViewer (OvcViewr). 830

        TOvcCustomTextFileViewer (OvcViewr)

TOvcTextFileViewer Component

Since a common use for the viewer control is to display the contents of an ASCII text file, a ready-made descendant of TOvcBaseViewer, TOvcTextFileViewer, is provided. TOvcTextFileViewer overrides GetLinePtr in order to respond to the viewer control's requests for lines of text, and it handles setting the LineCount property when a file is opened.

All you have to do is to tell the viewer control which file to read. This can be done either at design time or at run time by setting the FileName property. The value of the IsOpen property determines if the file is opened and read.

For a comprehensive example of how to use the TOvcTextFileViewer component, see the TEXTVIEW sample program.

Example

This example shows how to construct a text file viewer component to load and view an existing text file.

Create a new project, add components, and set the property values as indicated in Table 28.3.

Table 28.3: *TOvcTextFileViewer example*

Component	**Property**	**Value**
TOvcTextFileViewer	FileName	C:\AUTOEXEC.BAT
	IsOpen	True

Run the project and experiment with the generated program.

Hierarchy

TCustomControl (VCL)

- ❶ TOvcCustomControl (OvcBase) 16
  - ❷ TOvcCustomControlEx (OvcBase) 19
    - ❸ TOvcBaseViewer (OvcViewr) 830
      - TOvcCustomTextFileViewer (OvcViewr) 851
        - TOvcTextFileViewer (OvcViewr)

Properties

❶ About	❸ ExpandTabs	❸ LineLength
❶ AttachedLabel	FileName	❸ Lines
❸ Borders	FileSize	❸ MarginColor
❸ BorderStyle	❸ FixedFont	❸ ScrollBars
❸ Caret	❸ HighlightColors	❸ ShowBookmarks
❸ CaretActualPos	IsOpen	❸ ShowCaret
❸ CaretEffectivePos	❶ LabelInfo	❸ TabSize
❷ Controller	❸ LineCount	❸ TopLine

Methods

❸ ActualColumn	❸ GetLine	❸ SelectAll
❸ CheckLine	❸ GetMarkerPosition	❸ SetCaretPosition
❸ ClearMarker	❸ GetPrintableLine	❸ SetMarker
❸ CopyToClipboard	❸ GetSelection	❸ SetMarkerAt
❸ EffectiveColumn	❸ GotoMarker	❸ SetSelection
❸ GetCaretPosition	❸ Search	

Events

❶ AfterEnter	❶ OnMouseWheel	❸ OnTopLineChanged
❶ AfterExit	❸ OnShowStatus	❸ OnUserCommand

Reference Section

FileName property

```
property FileName : string
```

↳ The name of the file to view.

FileName is the file in the viewer or the name of a file that will be loaded from disk. If IsOpen is True and you change FileName, the new file is automatically loaded.

The following example loads a file into the text file viewer:

```
  OvcTextFileViewer1.FileName := 'C:\AUTOEXEC.BAT';
  OvcTextFileViewer1.IsOpen := True;
```

See also: IsOpen

FileSize run-time, read-only property

```
property FileSize : LongInt
```

↳ The number of bytes in the open file.

If the file is closed, FileSize returns 0.

The following example opens a file and obtains the size of the file:

```
  OvcTextFileViewer1.FileName := 'C:\AUTOEXEC.BAT';
  OvcTextFileViewer1.IsOpen := True;
  Size := OvcTextFileViewer1.FileSize;
```

IsOpen property

```
property IsOpen : Boolean
```

Default: False

↳ Indicates whether the file is open.

The IsOpen property determines if the file specified by the FileName property is open (i.e., being viewed). Setting IsOpen to True causes the file specified by the FileName property to be loaded. Setting IsOpen to False removes any existing text in the viewer and resets the scroll bars.

File errors generate standard VCL exceptions.

See also: FileName

TOvcCustomFileViewer Class

The TOvcCustomFileViewer class is the immediate ancestor of the TOvcFileViewer component. It implements all the methods and properties used by the TOvcFileViewer component and is identical to the TOvcFileViewer except that no properties are published.

TOvcCustomFileViewer is provided to facilitate creation of descendent viewer components. For property and method descriptions, see “TOvcFileViewer Component” on page 856.

Hierarchy

TCustomControl (VCL)

  TOvcCustomControl (OvcBase) 16

    TOvcCustomControlEx (OvcBase) 19

      TOvcBaseViewer (OvcViewr). 830

        TOvcCustomFileViewer (OvcViewr)

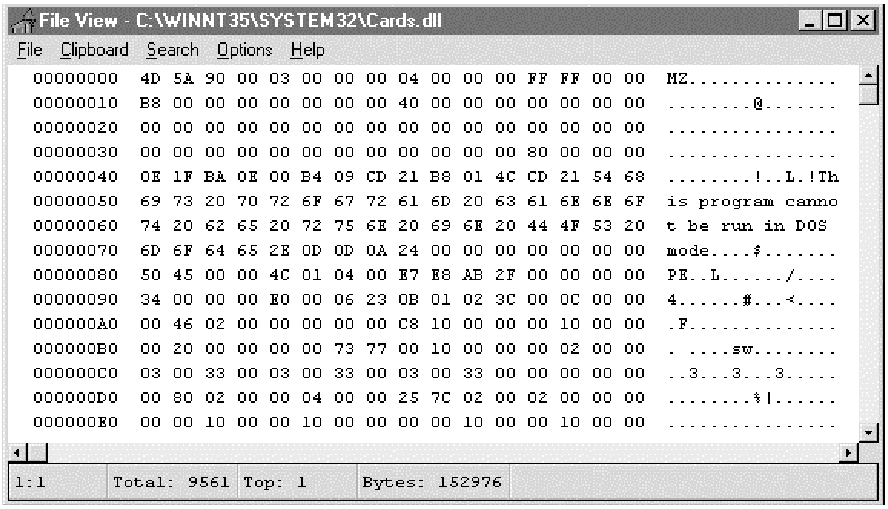
TOvcFileViewer Component

If you want to use a viewer component to implement a browser for displaying files of any kind (binary as well as text), use the TOvcFileViewer component.

Besides being able to display the contents of binary files, TOvcFileViewer has one other important advantage over its sibling: it does not try to read the entire file into memory. Instead, TOvcFileViewer reads files in 4KB segments, storing the data in a series of buffers, which are managed on a least-recently-used basis (i.e., when all of its buffers are full, it reuses the buffer that was least recently used). The main disadvantage of TOvcFileViewer is its performance. Although it is designed to respond as quickly as possible, there are cases where its response time is noticeably slow, such as when it is asked to move the cursor from the beginning of a long file to the end. In such cases, the TOvcFileViewer displays an hourglass cursor to warn the user of the possible delay.

The TOvcFileViewer is essentially identical to the TOvcTextFileViewer, except that there are a few more properties. BufferPageCount determines the number of buffers that should be allocated to store the data read from disk. In general, the larger the value specified here, the better the TOvcFileViewer will perform. The minimum value for BufferPageCount is 2, and the maximum value is 512. If the value that you specify is outside this range, the default value of 16 is assumed. Since each buffer holds 4KB of data, 16 buffers will hold 64KB of data.

The InHexMode property indicates whether you want the browser to display the data in hex mode, as shown in Figure 28.1.



```
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..............
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  ................
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ........!..L.!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  50 45 00 00 4C 01 04 00 E7 E8 AB 2F 00 00 00 00  PE..L....../....
00000090  34 00 00 00 E0 00 06 23 0B 01 02 3C 00 0C 00 00  4......#...<....
000000A0  00 46 02 00 00 00 00 00 C8 10 00 00 00 10 00 00  .F..............
000000B0  00 20 00 00 00 00 73 77 00 10 00 00 00 02 00 00  . ....sw........
000000C0  03 00 33 00 03 00 33 00 03 00 33 00 00 00 00 00  ..3...3...3.....
000000D0  00 80 02 00 00 04 00 00 25 7C 02 00 02 00 00 00  ........%|......
000000E0  00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00  ................

1:1   Total: 9561   Top: 1   Bytes: 152976
```

Figure 28.1: File Viewer

Generally, you should assign True to the InHexMode property only in cases where you know for certain that a binary file is being displayed, or where the stated purpose of your viewer window is to display the contents of binary files. In all other cases, you should provide the user a means of switching between ASCII and hex modes.

For a comprehensive example of how to use a TOvcFileViewer, see the FILEVIEW demo program.

Example

This example shows how to construct a project containing a text file viewer component to load and allow viewing an existing text file in hex mode.

Create a new project, add components, and set the property values as indicated in Table 28.4.

Table 28.4: *TOvcTextFileViewer example*

Component	**Property**	**Value**
`TOvcTextFileViewer`	`FileName`	`C:\AUTOEXEC.BAT`
	`IsOpen`	`True`
	`InHexMode`	`True`

Run the project and experiment with the generated program.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase) 16

❷ TOvcCustomControlEx (OvcBase) 19

❸ TOvcBaseViewer (OvcViewr) 830

TOvcCustomFileViewer (OvcViewr)

TOvcFileViewer (OvcViewr)

Properties

- ❶ About
- ❶ AttachedLabel
- ❸ Borders
- ❸ BorderStyle
- BufferPageCount
- ❸ Caret
- ❸ CaretActualPos
- ❸ CaretEffectivePos
- ❷ Controller
- ❸ ExpandTabs
- FileName
- FileSize
- FilterChars
- ❸ FixedFont
- ❸ HighlightColors
- InHexMode
- IsOpen
- ❶ LabelInfo
- ❸ LineCount
- ❸ LineLength
- ❸ Lines
- ❸ MarginColor
- ❸ ScrollBars
- ❸ ShowBookmarks
- ❸ ShowCaret
- ❸ TabSize
- ❸ TopLine

Methods

- ❸ ActualColumn
- ❸ CheckLine
- ❸ ClearMarker
- ❸ CopyToClipboard
- ❸ EffectiveColumn
- ❸ GetCaretPosition
- ❸ GetLine
- ❸ GetMarkerPosition
- ❸ GetPrintableLine
- ❸ GetSelection
- ❸ GotoMarker
- ❸ Search
- ❸ SelectAll
- ❸ SetCaretPosition
- ❸ SetMarker
- ❸ SetMarkerAt
- ❸ SetSelection

Events

- ❶ AfterEnter
- ❶ AfterExit
- ❶ OnMouseWheel
- ❸ OnShowStatus
- ❸ OnTopLineChanged
- ❸ OnUserCommand

Reference Section

BufferPageCount property

```
property BufferPageCount : Integer
```

Default: 16

↳ The number of buffers for the viewer.

BufferPageCount determines number of buffers that should be allocated to store the data read from disk. In general, the larger the value specified here, the better the TOvcFileViewer will perform. The minimum value for BufferPageCount is 2, and the maximum value is 512. If the value that you specify is outside this range, the default value of 16 is assumed. Since each buffer holds 4KB of data, 16 buffers will hold 64KB of data.

BufferPageCount can be changed while a file is open. The buffers are reallocated and existing buffer contents are transferred automatically.

FileName property

```
property FileName : string
```

↳ The name of the file to view.

FileName is the file in the viewer or the name of a file that will be loaded from disk. If IsOpen is True and you change FileName, the new file is automatically loaded.

The following example loads a file into the text file viewer:

```
  OvcFileViewer1.FileName := 'C:\AUTOEXEC.BAT';
  OvcFileViewer1.IsOpen := True;
```

See also: IsOpen

FileSize run-time, read-only property

```
property FileSize : LongInt
```

↳ The number of bytes in the open file.

If the file is closed, FileSize returns 0.

The following example opens a file and obtains the size of the file:

```
  OvcFileViewer1.FileName := 'C:\AUTOEXEC.BAT';
  OvcFileViewer1.IsOpen := True;
  Size := OvcFileViewer1.FileSize;
```

FilterChars property

```
property FilterChars : Boolean
```

Default: True

↳ Determines if the viewer displays non-ASCII characters.

If the InHexMode property is True and FilterChars is True, the viewer displays characters outside of the range ASCII 32–127 as a period. If InHexMode is True and FilterChars is False, the viewer displays all characters. Their appearance depends on the current font.

If InHexMode is False, this property has no effect.

See also: InHexMode

InHexMode property

```
property InHexMode : Boolean
```

Default: False

↳ Determines whether the viewer is in hex or ASCII mode.

If InHexMode is True, the viewer is in hex mode. If it is False, the viewer is in ASCII mode.

IsOpen property

```
property IsOpen : Boolean
```

Default: False

↳ Determines whether the file is open.

The IsOpen property determines if the file specified by the FileName property is open (i.e., being viewed). Setting IsOpen to True causes the file specified by the FileName property to be opened. Setting IsOpen to False removes any existing text in the viewer, deallocates the buffers, and resets the scroll bars.

The file is opened as read-only deny write (filemode $20) and any file errors generate standard VCL exceptions.

See also: FileName

Chapter 29: Timer Pool Component

The TOvcTimerPool component provides all the functions of the standard TTimer component, plus the additional benefit of using only one Windows timer for up to 32,767 timing events. The practical maximum number of timing events depends on how long it takes your application to service each timer notification.

In the 16-bit world, Windows timers were a scarce system resource. Using more timers than is absolutely necessary could adversely impact not only your application but also other applications. If a timer was not available when a program needed one, it could either terminate gracefully or it could terminate in a manner that would leave Windows in an unstable state. The OvcTimerPool was originally created to address this issue. Although the original system resource problem isn't much of an issue in the 32-bit world, the OvcTimerPool is still a pretty efficient way of managing a bunch of system timers.

TOvcTimerPool uses the concept of a timer trigger to perform the same function as TTimer's OnTimer event. Many timer triggers can be serviced by one Windows timer and this is what allows the TOvcTimerPool to provide multiple timer notifications and still use only one Windows timer.

After adding a TOvcTimerPool component to your form, you can add a timer by making a call to the timer pool's Add method. The method name passed to Add is called when the trigger interval expires.

Triggers can repeat until they are canceled (use Add) or they can be fired only once and then automatically removed (use AddOneTime).

The TimerPool is also used internally in many of the Orpheus components.

Example

This example creates a form with labels that display information about the status of several timer triggers.

Create a new project, add components, and set the property values as indicated in 29.1.

Table 29.1: *TimerPool example*

Component	Property	Value
TOvcTimerPool		
TLabel	Caption	Trigger 1
TLabel	Caption	Trigger 2
TLabel	Caption	Trigger 3
TLabel	Caption	Trigger 4
TLabel	Caption	Trigger 5
TButton	Caption	Start

Double-click on the button component and add the following method:

```
procedure TForm1.Button1Click(Sender : TObject);
begin
  {add 5 timer triggers. not saving returned handle}
  OvcTimerPool1.Add(nil, 500);
  OvcTimerPool1.Add(nil, 1000);
  OvcTimerPool1.Add(nil, 1500);
  OvcTimerPool1.Add(nil, 2500);
  OvcTimerPool1.Add(nil, 3000);
  Button1.Enabled := False;
end;
```

This method adds five timer triggers. Passing nil as the first parameter to Add informs the timer pool component that there is not a specific event handler for this timer trigger. For this example, the OnAllTriggers event handler is called for all timer trigger events.

Double-click on the TOvcTimerPool component to create an OnAllTriggers event handler and add the following method:

```
procedure TForm1.OvcTimerPool1AllTriggers(
  Sender : TObject; Handle : Integer; Interval : Cardinal;
  ElapsedTime : LongInt);
var
  S : string;
begin
  try
    S := Format('Trigger handle: %d, Interval:%d,
      Elapsed Time:%d', [Handle, Interval, ElapsedTime]);
    case Handle of
      0 : Label1.Caption := S;
      1 : Label2.Caption := S;
      2 : Label3.Caption := S;
      3 : Label4.Caption := S;
      4 : Label5.Caption := S;
    end;
  except
    {in case of an error, remove all timer triggers}
    OvcTimerPool1.RemoveAll;
    raise;
  end;
end;
```

Run the project to see the label components updated with the timer trigger information.

TOvcTimerPool Component

TOvcTimerPool performs the same function as TTimer's OnTimer event but provides multiple timer notifications while using only one Windows timer.

Hierarchy

TComponent (VCL)

TOvcTimerPool (OvcTimer)

Properties

About	ElapsedTimeSec	OnAllTriggers
Count	Enabled	OnTrigger
ElapsedTime	IntervalEvents	

Methods

Add	AddOneTime	RemoveAll
AddOneShot	Remove	

Reference Section

About — read-only property

```
property About : string
```

Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

Add — method

```
function Add(
  OnTrigger : TTriggerEvent; Interval  : Cardinal) : Integer;
```

```
TTriggerEvent = procedure(
  Sender : TObject; Handle : Integer;
  Interval : Cardinal; ElapsedTime : LongInt) of object;
```

Creates a timer trigger.

The returned value is a handle for the new timer trigger. It is used to reference this timer trigger when using other TOvcTimerPool properties and methods.

Trigger handles are usually generated sequentially, starting at 0, but you cannot always depend on this. For example, if 5 timer triggers are added, they get handles 0 through 4. If the timer trigger with handle 2 is then removed, the next timer trigger added will get handle 2.

If a method is assigned to the OnTrigger parameter, it is called every Interval milliseconds until the timer trigger is removed or disabled. The method assigned to the OnAllTriggers event is also called every Interval milliseconds. If nil is passed for the OnTrigger parameter, only the method assigned to the OnAllTriggers event is called.

The following example adds a timer trigger with an interval of 5000 milliseconds (5 seconds). SomeTriggerMethod and the method assigned to the OnAllTriggers event are called every 5 seconds.

```
  H := OvcTimerPool.Add(SomeTriggerMethod, 5000);
```

See also: AddOneTime, Count, Enabled, OnAllTriggers, Remove

AddOneShot method

```
function AddOneShot(
  OnTrigger : TTriggerEvent; Interval  : Cardinal) : Integer;
```

↳ Creates or updates one timer trigger that is removed automatically after one firing.

AddOneTime method

```
function AddOneTime(
  OnTrigger : TTriggerEvent; Interval  : Cardinal) : Integer;
```

```
TTriggerEvent = procedure(
  Sender : TObject; Handle : Integer;
  Interval : Cardinal; ElapsedTime : LongInt) of object;
```

↳ Creates a timer trigger for one-time use.

When the specified interval expires, the trigger is automatically removed. The returned value is a handle for the new timer trigger. It is used to reference this timer trigger when using other TOvcTimerPool properties and methods.

If a method is assigned to the OnTrigger parameter, it is called when the specified Interval expires. The method assigned to the OnAllTriggers event is also called when the specified Interval expires. If nil is passed for the OnTrigger parameter, only the method assigned to the OnAllTriggers event is called.

The following example adds a timer trigger with an interval of 1000 milliseconds (1 second). Since nil is passed for the OnTrigger parameter, only the method assigned to the OnAllTriggers event is called. The added trigger fires once, after 1000 milliseconds has elapsed, and then it is automatically removed.

```
  H := OvcTimerPool.AddOneTime(nil, 1000);
```

See also: Add, Enabled, OnAllTriggers, Remove

Count run-time, read-only property

```
property Count : Integer
```

↳ Returns the total number of timer triggers.

The number returned by Count includes both enabled and disabled timers.

ElapsedTime — run-time, read-only property

```
property ElapsedTime[Handle : Integer] : LongInt
```

↳ Returns the number of milliseconds since the timer trigger was added.

Handle is the value returned by Add or AddOneTime. A timer trigger's elapsed time cannot be reset unless the trigger is removed and then added again.

See also: Add, AddOneTime, ElapsedTimeSec

ElapsedTimeSec — run-time, read-only property

```
property ElapsedTimeSec[Handle : Integer] : LongInt
```

↳ Returns the number of seconds since the timer trigger was added.

Handle is the value returned by Add or AddOneTime.

See also: Add, AddOneTime, ElapsedTime

Enabled — run-time property

```
property Enabled[Handle : Integer] : Boolean
```

↳ Determines whether the specified timer trigger is active.

If Enabled is True, the specified timer trigger is active and timer events are generated based on the timer trigger's interval. If Enabled is False, the timer trigger information is still maintained and updated, but no events are generated. Handle is the value returned by Add or AddOneTime.

A disabled timer trigger can be enabled by setting Enabled to True, or removed using Remove or RemoveAll.

Interval — run-time property

```
property Interval[Handle : Integer] : Cardinal
```

↳ Determines the number of milliseconds between trigger notifications.

The trigger interval is initially set when the timer trigger is added, but can be changed by assigning a new interval to this property. Handle is the value returned by Add or AddOneTime.

The elapsed time for the timer is not reset when the interval is changed.

See also: Add, AddOneTime

OnAllTriggers event

```
property OnAllTriggers : TTriggerEvent
```

```
TTriggerEvent = procedure(
  Sender : TObject; Handle : Integer;
  Interval : Cardinal; ElapsedTime : LongInt) of object;
```

↳ Defines an event handler that is called for all trigger events.

The method assigned to the OnAllTriggers event is called for each trigger event. Handle identifies the trigger that fired, Interval is the interval (in milliseconds) assigned to this trigger, and ElapsedTime is the number of milliseconds since the trigger was created.

The following example shows how to create and respond to a timer trigger from within an OnAllTriggers event handler:

```
  Trigger1 := OvcTimerPool.Add(nil, 500);
  Trigger2 := OvcTimerPool.Add(nil, 2500);
  ...
  procedure TForm1.AllTriggers(Sender : TObject; Handle : Integer;
    Interval : Cardinal; ElapsedTime : LongInt);
  begin
    if Handle = Trigger1 then
      {do something because trigger 1 fired}
    if Handle = Trigger2 then
      {do something because trigger 2 fired}
  end;
```

See also: OnTrigger

OnTrigger event

```
property OnTrigger[Handle : Integer] : TTriggerEvent
```

```
TTriggerEvent = procedure(
  Sender : TObject; Handle : Integer;
  Interval : Cardinal; ElapsedTime : LongInt) of object;
```

↳ Defines access to an array of trigger event handlers.

This indexed event allows you to assign trigger handlers (TTriggerEvent methods) for all timer triggers in the internal list. Handle is the value returned by either the Add or AddOneTime methods. Using OnTrigger with an invalid Handle index causes an EInvalidTriggerHandle exception to be raised.

The following example shows how to assign a method to a specific trigger:

```
Trigger1 := OvcTimerPool.Add(nil, 500);
Trigger2 := OvcTimerPool.Add(nil, 2500);
...
procedure TForm1.MyTrigger1EventHandler(Sender : TObject;
  Handle : Integer; Interval : Cardinal; ElapsedTime : LongInt);
begin
  {do something because trigger 1 fired}
end;
...
procedure TForm1.MyTrigger2EventHandler(Sender : TObject;
  Handle : Integer; Interval : Cardinal; ElapsedTime : LongInt);
begin
  {do something because trigger 2 fired}
end;
...
OnTrigger[Trigger1] := Form1.MyTrigger1EventHandler;
OnTrigger[Trigger2] := Form1.MyTrigger2EventHandler;
```

See also: Add, AddOneTime, OnAllTriggers

Remove method

```
procedure Remove(Handle : Integer);
```

↳ Removes the specified timer trigger.

Handle is the value returned by Add or AddOneTime. When all timer triggers have been removed, the Windows timer is automatically destroyed. It is recreated when a new timer trigger is added.

The following example first adds and then removes a timer trigger:

```
Trigger1 := OvcTimerPool.Add(nil, 500);
...
OvcTimerPool.Remove(Trigger1);
```

See also: Add, AddOneTime, RemoveAll

RemoveAll method

```
procedure RemoveAll;
```

↳ Removes all timer triggers and destroys the Windows timer.

Chapter 30: Tables

The Orpheus tables are grid-type controls for displaying and editing data in rows and columns. The table components use individual cell components to define the data that is displayed and edited in a cell, or in a column of cells. These cell components are classes, so you can define (through normal object-oriented techniques) your own cell components in the table, thus extending its functionality.

The main assumption for the table component is that the data comes from some kind of database (and that word is used in a very loose sense), and that the rows in the table map to the records in the database and the columns to the fields within each record. Hence in the table's standard state, each column has a default cell component associated with it. This default cell component is used to display and edit the data in all the cells in the column. The cell component stores information on how to display the data such as the color, font, access rights (e.g., read only), and the adjustment (where to display the data in the cell). Although the table has been designed so that there is a simple intuitive mapping between rows and 'records', and columns and 'fields', you do not have to use the table in this way.

Because the data types displayed and edited in a table can be so varied, the table does not store any data itself. You must define and implement a data storage and/or caching scheme. When the table needs to display or edit data, it triggers an event. You must provide a handler to service that event and provide the data as a pointer. All the data for one row is requested before moving on to the next row. The table does however store the information required to display and edit the data. That is done through a tightly interlocking set of classes that manage the information.

To aid in building sophisticated tables, each cell can have a different color, font, accessibility, and adjustment. The cell attribute specification is also exported as an event, so you can change the look of the cell at run time, depending on the data being displayed. For example, you could display negative monetary amounts in red. The table component goes even further: you can specify any cell to be of any type. In other words, you are not limited to the "one column equals one data type" rule, so you can easily build non-database-oriented tables.

The types of cells provided with the Orpheus table are the most-requested ones: string cells (including memos), bitmap or glyph cells, check boxes, Orpheus simple, numeric and picture fields, and combo boxes. You can expand the list further by defining other cell types through normal class inheritance methods.

Two other attributes of a table are the rows and the columns. A row is fully defined by its style, which consists of its height and an indication of whether the row is visible. The table contains an array that holds all the row styles (actually it is a class that encapsulates a sparse array). The data that is displayed in the row (and how it is displayed) is processed by the various cell components along the row, not by the row itself. Similarly, a column is fully defined by its style, which consists of its width, an indication of whether the column is visible, and the default cell component (which is used to display the data within the column). The table contains an array that holds all the columns/definitions in the table. The data that is displayed in the column is processed by the default cell for the column, not by the column itself. Therefore, neither rows nor columns have attributes for color, font, access rights, or adjustment. It is the cell components that have these attributes and do all the painting and editing of data.

There are several visually different regions in a table, the main area, the locked area, and the unused area. The main area of the table is where the data is shown. This is where the active cell is and where the cells are edited (the cells in the main area are also called normal cells).

The locked area is a set of cells that act as row and column headings. They do not scroll like the normal cells. For example, assume that a table has one locked column and one locked row. If the main part of the table scrolls left or right, the locked row (the column headings) scrolls in the same manner, but the locked column (the row headings) stays fixed. Similarly if the main area is scrolled up and down, the locked column scrolls and the locked row stays fixed. Locked cells can have a different appearance than normal cells to visually separate them. By default the locked cells have a raised appearance and the normal cells have a flat appearance. The locked cells also act as a place where operations such as row/column resizing, and row/column movements are performed by the user.

The unused area of the table is anywhere in the client rectangle of the table where there are no cells. It is the area of the table that is either beyond the last column and/or beyond the last row.

One special cell in the normal area of the table is the active cell. This cell is the focused cell of the table. It is the cell where all editing takes place. If you press <F2>, the active cell is edited. The active cell is visually differentiated from the other normal cells by two different attributes. It has a focus box around it, and it has a different background and foreground color. You can move this “highlight” around the normal area of the table by clicking with the mouse or by using the arrow and the paging keys. The active cell also has different attributes when the table does not have the focus. This helps to differentiate the active cell even when another control has the focus.

As specified in the standard Windows user interface guidelines, using the scrollbars scrolls the table without moving the active cell. It is a mouse click in the normal area, a keyboard action, or a programmatic action that causes the active cell to move. You can scroll the active cell completely off the visible part of the table by using the scrollbars. If you do so, pressing an arrow key (for example) causes the table to scroll all the way back so that the (new) active cell is shown.

Another attribute of the table is selection. The user can select a block (or several disjoint blocks) of cells in the table (maybe for printing or to copy to the clipboard), and these cells are shown with a different background and foreground colors. The user can select cells by clicking and dragging the mouse, by using the standard Windows click, shift+click, ctrl+click and ctrl+shift+click mouse operations, or by using the shift and arrow keys.

The normal Orpheus table is not data-aware, but there is a data-aware variant with slightly restricted functionality (at least with regard to the customization aspects) and this component is described later in this chapter.

Hierarchy

The hierarchy for the classes and components discussed in this chapter is somewhat complex because the Table component relies on a lot of ancillary classes to do its work.

TPersistent (VCL)

- TOvcTableColors (OvcTbClr)
- TOvcTableColumn (OvcTbCls)
- TOvcGridPen (OvcTGPns)
- TOvcGridPenSet (OvcTGPns)
- TOvcCellGlyphs (OvcTGRes)

The TOvcTableColors class encapsulates the colors that a table uses (e.g., the colors for locked cells, normal cells, and the active cell). The table creates a single object of this type when it is created.

The TOvcTableColumn class defines the style for a particular column in the table. An object of this class stores the default cell component for the column. The table does not deal with these objects individually; it uses a TOvcTableColumns object instead.

The TOvcGridPen class encapsulates a single grid pen style. The grid pen is used to draw a type of grid line in the table. The grid is defined as the lines that separate the cells from each other. A different grid pen is used for the locked cells and for the normal cells. A grid pen is also used to draw the focused box around the active cell. All the grid pens required by a table are held in an object of class TOvcGridPenSet. The table does not deal with these objects individually, but uses a TOvcGridPenSet object instead.

The TOvcGridPenSet class encapsulates all the grid pens required by a table. The table creates a single object of this type when it is created.

The TOvcCellGlyphs class encapsulates a bitmap that is divided up into glyphs. A glyph is defined to be a square subdivision of a bitmap, in the same sense that TBitBtn uses a bitmap. The class also stores the number of glyphs in the bitmap and the number of glyphs that are actually used. The class is used internally by the TOvcTCGlyph cell component.

TObject (VCL)

  TOvcSparseArray (OvcSpAry)

  TOvcTableRows (OvcTbRws)

  TOvcTableColumns (OvcTbCls)

  TOvcTableCells (OvcTCell)

The TOvcSparseArray class implements a sparse array container. The style for each row is held in a sparse array. If you specify individual cells to have individual attributes, then these are held in a sparse matrix (a sparse array of sparse arrays).

The TOvcTableRows class holds all the information about the styles of the rows. It uses a sparse array internally. The table creates a single object of this type when it is created.

The TOvcTableColumns class holds all the information about the styles of the columns. It uses an array (a TList) internally. The table creates a single object of this type when it is created.

The TOvcTableCells class holds all the information about the cells that have individual attributes. These are the cells that do not follow the standard set by the default cell component for the column in which they appear. The data is held in a sparse matrix. The table creates a single object of this type when it is created.

TComponent (VCL)

  TOvcTableCellAncestor (OvcTCmmn)

    TOvcBaseTableCell (OvcTCell)

      TOvcTCBaseBitMap (OvcTCBmp)

      TOvcTCCustomIcon (OvcTCIco)

      TOvcTCBaseString (OvcTCStr)

The TOvcTableCellAncestor class is a “place holder” in the cell component hierarchy. It has some simple functionality that maintains a relationship with the table that is using it.

The TOvcBaseTableCell class is the base ancestor to the entire cell component class hierarchy. It defines the virtual methods required for the table to interact with its cell components when painting and editing. These virtual methods are generally abstract and must be overridden in descendants to provide the proper functionality.

The TOvcTCBaseBitMap class is the base ancestor to the cell components that display a bitmap. It defines a common method of displaying a bitmap (or subdivision of a bitmap).The hierarchy for the TOvcTCBaseBitMap is:

TOvcTCBaseBitMap (OvcTCBmp)

  TOvcTCCustomGlyph (OvcTCGly)

    TOvcTCGlyph (OvcTCGly)

    TOvcTCCustomCheckBox (OvcTCBox)

      TOvcTCCheckBox (OvcTCBox)

The TOvcTCCustomBitmap cell component provides all the functionality for cells that display bitmaps, but does not publish any properties. The cells are not editable.

The TOvcTCBitmap cell component is a simple descendant of TOvcTCCustomBitmap that publishes the relevant properties.

The TOvcTCCustomGlyph cell component provides all the functionality for cells that display glyphs, but does not publish any properties. When these cells are edited, pressing the spacebar (or clicking on the cell) causes the glyphs to be cycled through sequentially. One example is the checkbox cell component (TOvcTCCheckBox). Instantiated objects of this class create a single object of type TOvcCellGlyphs. You can create glyph cell descendants from the TOvcTCCustomGlyph class and publish the relevant properties.

The TOvcTCGlyph class is a simple descendant of TOvcTCCustomGlyph that publishes the relevant properties.

The TOvcTCCustomCheckBox cell component provides all the functionality for cells that display checkboxes, but does not publish any properties. When these cells are edited, pressing the spacebar (or clicking on the cell) causes the checkbox glyphs (unchecked, checked, and grayed) to be cycled through sequentially. You can create checkbox cell descendants from this class and publish the relevant properties.

The TOvcTCCheckBox class is a simple descendant of TOvcTCCustomCheckBox that publishes the relevant properties.

The TOvcTCCustomIcon class is the base ancestor to the cell components that display icons. It provides all the functionality for displaying icons but does not publish any properties. You can create icon cell descendants from this class and publish the relevant properties. The hierarchy for TOvcTCCustomIcon is:

TOvcTCCustomIcon (OvcTCIco)

  TOvcTCIcon (OvcTCIco)

The TOvcTCIcon is a simple descendant of TOvcTCCustomIcon that publishes the relevant properties.

The TOvcTCBaseString class is the base ancestor to the cell components that display a string. It defines a common method of displaying a string. The strings can be either length-byte strings (short strings) or null-terminated strings. The hierarchy for TOvcTCBaseString is:

```
TOvcTCBaseString (OvcTCStr)
    TOvcTCCustomString (OvcTCEdt)
        TOvcTCString (OvcTCEdt)
    TOvcTCColHead (OvcTCHdr)
    TOvcTCRowHead (OvcTCHdr)
    TOvcTCCustomMemo (OvcTCEdt)
        TOvcTCMemo (OvcTCEdt)
    TOvcTCBaseEntryField (OvcTCBEF)
        TOvcTCCustomSimpleField (OvcTCSim)
            TOvcTCSimpleField (OvcTCSim)
        TOvcTCCustomPictureField (OvcTCPic)
            TOvcTCPictureField (OvcTCPic)
        TOvcTCCustomNumericField (OvcTCNum)
            TOvcTCNumericField (OvcTCNum)
```

The TOvcTCCustomString class defines a cell component that edits strings using a TEdit control. It does not publish any properties. You can create string cell descendants from this class and publish the relevant properties.

The TOvcTCString class is a simple descendant of TOvcTCCustomString that publishes the relevant properties.

The TOvcTCColHead component is an extremely simple string cell that either displays a set of column headings or displays the column number in the spreadsheet format. For the latter, if the column number is 1, it displays an 'A'; if 2 a 'B'; if 26 a 'Z'; if 27 an 'AA'; and so on. This is useful for simple column headings.

The TOvcTCRowHead component is an extremely simple string cell that displays the row number. It is useful for simple row headings.

The TOvcTCCustomMemo class defines a cell component that edits null-terminated strings using a TMemo control. It does not publish any properties. You can create memo cell descendants from this class and publish the relevant properties.

The TOvcTCMemo class is a simple descendant of TOvcTCCustomMemo that publishes the relevant properties.

The TOvcTCBaseEntryField class encapsulates an Orpheus TOvcBaseEntryField (and its descendants the simple, picture, and numeric fields) as a cell component. It contains most of the functionality required for such a cell component.

The TOvcTCCustomSimpleField class encapsulates a simple field as a cell component. It is derived from TOvcTCBaseEntryField and does not publish any properties.

The TOvcTCSimpleField class is a simple descendant of TOvcTCCustomSimpleField that publishes the relevant properties.

The TOvcTCCustomPictureField class encapsulates a picture field as a cell component. It is derived from TOvcTCBaseEntryField and does not publish any properties.

The TOvcTCPictureField class is a simple descendant of TOvcTCCustomPictureField that publishes the relevant properties.

The TOvcTCCustomNumericField class encapsulates a numeric field as a cell component. It is derived from TOvcTCBaseEntryField and does not publish any properties.

The TOvcTCNumericField class is a simple descendant of TOvcTCCustomNumericField that publishes the relevant properties.

```
TCustomControl (VCL)
TOvcCustomControl (OvcBase)
   TOvcCustomControlEx (OvcBase)
        TOvcTableAncestor (OvcTCmmn)
             TOvcCustomTable (OvcTable)
                  TOvcTable (OvcTable)
```

The TOvcTableAncestor class is a “place holder” in the table class hierarchy. It has simple functionality to interact with the cell components that are used with it.

The TOvcCustomTable class provides all the functionality of the Orpheus table class. Following the VCL standards for custom controls, it does not publish any properties but keeps them protected. It is the TOvcTable class that publishes these properties and can be included on the Component palette.

This chapter describes all the non-visual ancillary classes first then the various cell classes and components, and finally the Orpheus table itself (this is the same order as in the preceding hierarchy diagrams). As a first read-through you might find it easier to skip the ancillary classes, quickly browse through the cell classes and components, and concentrate on the table component. Then when you are ready to create your own cell components, you can go back and review the descriptions of the cells.

TRowNum and TColNum

There are two types that are pervasive in the Orpheus table and its associated classes: the TRowNum and TColNum types (declared in the OvcTCmmn unit).

```
TRowNum = LongInt;
```

This is the type of a row number. The actual usable range is 0 to 2 billion. Row numbers are zero based (row 0 is the first row). Sometimes row numbers can have negative values, but these are special values and these values have a constant declaration to define them. When a row number can have a special value for a property or method, the declaration of that property or method notes the special constant values.

Although row numbers can range from 0 to 2 billion, you can only specify individual row styles for the first 320,000 rows due to the underlying data structure. This should not be a problem in practice.

```
TColNum = Integer;
```

This is the type of a column number. The actual usable range is 0 to 16,383. Column numbers are zero based (column 0 is the first column). Sometimes column numbers can have negative values, but these are special values and these values have a constant declaration to define them. If a column number can have a special value for a property or method, the declaration for that property or method notes the special constant values.

Example

This example shows how to use an Orpheus table and associated cell components to design a form with a table.

Create a new project, add components, and set the property values as indicated in the following table:

Table 30.2: *Example project components and property values*

Component	**Property**	**Value**
OvcTable1	LockedRowsCell	OvcTCColHead1
OvcTCPictureField1	Adjust	otaCenterRight
	DataType	pftWord
	PictureMask	"iii"
	ColCount	3

Table 30.2: *Example project components and property values (continued)*

Component	**Property**	**Value**
OvcTCString1	MaxLength	49
OvcTCColHead1		
OvcTCRowHead1	ShowActiveRow	True

Enter the Columns property editor for the table by clicking on its ellipsis button (or by right-clicking on the table and selecting Column Editor). For column 0, set the Default Cell to OvcTCRowHead1 and the Width to 40. Select Apply. For column 1, set the Default Cell to OvcTCString1. Select Apply. For column 2, set the Default Cell to OvcTCPictureField1 and the Width to 60. Select Done.

You now need to define an OnGetCellData event handler that will provide the data for the table at run time. Select the Events page in the Object Inspector for the table and double-click on the OnGetCellData event. You are transferred to the code editor where you can enter the following code:

```
procedure
 TForm1.OvcTable1GetCellData(Sender : TObject; RowNum : LongInt;
                             ColNum : Integer; var Data :Pointer;
                             Purpose : TOvcCellDataPurpose);

type
  TTestRec = record
    Name  : string[49];
    Age   : Word;
  end;
  TTestDataBase = array [1..9] of TTestRec;

const
  TestDatabase : TTestDataBase = (
    (Name:'Bucknall, Julian'; Age:38),
    (Name:'Blanton, Phillip'; Age:36),
    (Name:'Huffman, Fred';    Age:45),
    (Name:'Welch, Mike';      Age:33),
    (Name:'Reisdorph, Kent';  Age:31),
    (Name:'Frerking, Gary';   Age:27),
    (Name:'DelRossi, Robert'; Age:42),
    (Name:'Jones, Zack';      Age:34),
    (Name:'';                 Age:0),
    (Name:'';                 Age:0));
begin
  Data := nil;
  if (1 <= RowNum) and (RowNum <= 9) then begin
    case ColNum of
      1 : Data := @TestDatabase[RowNum].Name;
      2 : Data := @TestDatabase[RowNum].Age;
    end;{case}
  end;
end;
```

This code declares a record type to hold the data, an array type to act as a “database” and a constant that holds the data for the database. A pointer to the relevant piece of data that’s defined by the row and column number is returned. Column 1 is the name and Column 2 is the age. A data pointer is returned only if the row number is between 1 and 9. Row 0 is the locked row and column 0 is the locked column—their data is provided automatically by the OvcTCColHead1 and OvcTCRowHead1 components respectively.

Save the project, compile, and run. You can experiment with the generated table. Move the active cell, scroll the table up and down, and resize the rows and columns (move the mouse over the locked cells and when the cursor changes shape you can click and drag a sizing line). Press <F2> to edit a cell, then use <F2> to save or <Esc> to abandon. You can even start typing a new value without pressing <F2>, the table switches you into edit mode automatically. Terminate the program by the system menu Close option.

The column headings are just the spreadsheet type letters, which are not very informative. To change them so that they describe the type of data in the column, select the TOvcTCColHead component and open up the Headings property so that you can edit the string list. Enter the following lines:

```
TP
Name
Age
```

If you compiled and ran this example, the text “Age” would be left justified. To make it right justified to match the numeric data, select the Events page of the Object Inspector for the table again. Double-click on OnGetCellAttributes and enter the following code:

```
procedure
  TForm1.OvcTable1GetCellAttributes
    (Sender : TObject; RowNum : LongInt; ColNum : Integer;
     var CellAttr : TOvcCellAttributes);
begin
  if (RowNum = 0) and (ColNum = 2) then
    CellAttr.caAdjust := otaCenterRight;
end;
```

This sets the adjustment value for row 0, column 2 to be otaCenterRight (center it vertically, adjust on the right horizontally).

Save the project, compile, and run. Experiment as before.

In the EXTBL02U.DFM project is a simple example that shows the use of all the cell components provided by Orpheus in a table. It will not be discussed here; you are invited to peruse the source code, especially for hints on the use of the Data parameter in your table’s OnGetCellData method with the different varieties of cell components.

TOvcTableColors Class

The TOvcTableColors class encapsulates the colors used by the Orpheus table component to simplify the Object Inspector display in the IDE.

Hierarchy

TPersistent (VCL)

  TOvcTableColors (OvcTbClr)

Properties

ActiveFocused	Editing	Selected
ActiveFocusedText	EditingText	SelectedText
ActiveUnfocused	Locked	
ActiveUnfocusedText	LockedText	

Reference Section

ActiveFocused property

```
property ActiveFocused : TColor
```

Default: clHighlight

↳ The background color for the active cell when the table has the focus.

ActiveFocusedText property

```
property ActiveFocusedText : TColor
```

Default: clHighlightText

↳ The font color for the active cell when the table has the focus.

ActiveUnfocused property

```
property ActiveUnfocused : TColor
```

Default: clHighlight

↳ The background color for the active cell when the table does not have the focus.

ActiveUnfocusedText property

```
property ActiveUnfocusedText : TColor
```

Default: clHighlightText

↳ The font color for the active cell when the table does not have the focus.

Editing property

```
property Editing : TColor
```

Default: clBtnFace

↳ The background color for the active cell when it is being edited.

EditingText property

```
property EditingText : TColor
```

Default: clWindowText

↳ The font color for the active cell when it is being edited.

Locked — property

```
property Locked : TColor
```

Default: clBtnFace

The background color for locked cells.

LockedText — property

```
property LockedText : TColor
```

Default: clWindowText

The font color for locked cells.

Selected — property

```
property Selected : TColor
```

Default: clHighlight

The background color for selected cells (except the active cell).

SelectedText — property

```
property SelectedText : TColor
```

Default: clHighlightText

The font color for selected cells (except the active cell).

TOvcTableColumn Class

The TOvcTableColumn class defines the attributes of a single column of a table. The three attributes for a column are width, whether it is hidden or not, and the default cell type. The default cell type defines how all the fields in the column are displayed and edited.

A table's column objects are stored in an instance of a TOvcTableColumns class. The TOvcTable class creates an instance of a TOvcTableColumns class and publishes it as the Columns property.

Hierarchy

TPersistent (VCL)

  TOvcTableColumn (OvcTbCls)

Properties

DefaultCell	Table
Hidden	Width

Methods

Create	Destroy

Reference Section

Create constructor

```
constructor Create(ATable : TOvcTableAncestor);
```

↳ A table column.

ATable is the table that will own this column object. A column can be owned by only one table. A column cannot be moved from one table to another. The link between table and column cannot be changed for the life of the column object. The column is created with the following attributes:

Attribute	**Result**
visible	Hidden is set to false.
no default cell	DefaultCell is set to nil. You must link a cell to the column by altering the DefaultCell property.
150 pixels wide	Width is set to 150.

See also: DefaultCell, Hidden, Width

DefaultCell run-time property

```
property DefaultCell : TOvcBaseTableCell
```

Default: nil

↳ The cell that the column uses for displaying and editing data.

DefaultCell defines how the column is seen. The column uses the default cell to display and edit data in the column.

When the DefaultCell property is set, the current DefaultCell's reference count is decremented (if there is a current default cell) and the new DefaultCell's reference count is incremented (if it is non-nil). If the new value for the DefaultCell property is a cell that refers to another table object, then nothing happens.

See also: TOvcBaseTableCell.References

Destroy destructor

```
destructor Destroy;
```

↳ The column object.

If the DefaultCell is non-nil, Destroy decrements the number of references for the cell (it does not destroy the cell), and then destroys the column instance.

See also: TOvcBaseTableCell.References

Hidden run-time property

```
property Hidden : Boolean
```

Default: False

↳ Indicates whether the column is hidden or visible.

If Hidden is True, the column is hidden from view. It still exists, but it is not shown. If a column is hidden, no data is requested for the default cell to display.

Table run-time, read-only property

```
property Table : TOvcTableAncestor
```

↳ The owner of the column.

Table is a read-only property set by the Create constructor. Once a column is created, you cannot change its owner.

Width run-time property

```
property Width : Integer
```

Default: 150

↳ The width of the column in pixels.

The minimum value for Width is 5.

TOvcGridPen Class

The TOvcGridPen class encapsulates the pen attributes for drawing a grid on the table. A grid is the set of horizontal and vertical lines that separate the cells. The table has five possible grid styles or effects: none, vertical lines only, horizontal lines only, vertical and horizontal lines, and a 3D grid.

The table uses four different grid pens: one for normal cells, one for locked cells, one for the focus box around the active cell when the table has the focus, and one for the focus box around the active cell when the table does not have the focus. These four grid pens are encapsulated in the TOvcGridPenSet class (see page 891). The table creates and publishes an object of TOvcGridPenSet class.

Hierarchy

TPersistent (VCL)

  TOvcGridPen (OvcTGPns)

Properties

Effect	SecondColor
NormalColor	Style

Effect property

```
property Effect : TGridEffect
```

```
TGridEffect = (
  geNone, geVertical, geHorizontal, geBoth, ge3D);
```

Default: geBoth

Defines the style of grid.

The styles of grid are:

Value	Description
geNone	No grid is drawn.
geVertical	Vertical lines are drawn in NormalColor.
geHorizontal	Horizontal lines are drawn in NormalColor.
geBoth	Both vertical and horizontal lines are drawn in NormalColor.
ge3D	A 3D-style grid with the top and left hand lines drawn in SecondColor and the bottom and right lines in NormalColor. If SecondColor is clWhite and NormalColor is clBlack, the cells appear raised. If SecondColor is clBlack and NormalColor is clWhite, the cells appear lowered.

The grid lines are drawn with a pen width of 1.

See also: NormalColor, SecondColor

NormalColor property

```
property NormalColor : TColor
```

Default: clBtnShadow

The normal color of the grid pen.

NormalColor is used to draw the grid lines when the Effect property is geVertical, geHorizontal, or geBoth. When Effect is ge3D, NormalColor is used to draw the bottom and right lines.

See also: Effect, SecondColor

SecondColor property

```
property SecondColor : TColor
```

Default: clBtnHighlight

The secondary color of the grid pen.

SecondColor is used to draw the top and left grid lines when the Effect property is ge3D. Otherwise, SecondColor is not used.

See also: Effect, NormalColor

Style property

```
property Style : TPenStyle
```

Default: psSolid

Defines the style of the pen used when drawing grid lines.

See the TPenStyle type in the VCL documentation for a list of the pen styles. If the pen style is a dotted or dashed type, the background color of the pen is set to the same color as the Color property of the TOvcTable component.

The grid lines are drawn with a pen width of 1.

See also: Effect

TOvcGridPenSet Class

An Orpheus table requires four distinct grid pens and these are encapsulated into a TOvcGridPenSet. The four types are the grid lines around locked cells, the grid lines around the active cell when the table has the focus, the grid lines around the active cell when the table does not have the focus, and the grid lines in the main body of the table. The TOvcTable class creates an object of this type and publishes it as the GridPenSet property.

Hierarchy

TPersistent (VCL)

  TOvcGridPenSet (OvcTGPns)

Properties

CellWhenFocused	LockedGrid
CellWhenUnfocused	NormalGrid

Reference Section

CellWhenFocused property

`property CellWhenFocused : TOvcGridPen`

Default: solid, black, both lines (horizontal and vertical)

↳ The grid pen used for the active cell when the table has the focus.

This grid pen is used to draw a box around the active cell when the table has the focus. The box is drawn inside the normal grid.

CellWhenUnfocused property

`property CellWhenUnfocused : TOvcGridPen`

Default: dashed, black, both lines (horizontal and vertical)

↳ The grid pen used for the active cell when the table does not have the focus.

This grid pen is used to draw a box around the active cell when the table does not have the focus. The box is drawn inside the normal grid.

LockedGrid property

`property LockedGrid : TOvcGridPen`

Default: solid, button shadow and highlight colors, raised 3D effect

↳ The grid pen used for the locked cells in the table.

NormalGrid property

`property NormalGrid : TOvcGridPen`

Default: dotted, button shadow color, both lines (horizontal and vertical)

↳ The grid pen used for the normal part of the table.

TOvcCellGlyphs Class

The TOvcCellGlyphs class provides bitmap resource management and encapsulation for the glyph table cell components (TOvcTCGlyph and TOvcTCCheckBox).

The class conserves resources by reusing a set of glyphs between many cell components. The glyphs are equal-sized square subdivisions of a single bitmap. Each subdivision is a single glyph, and the cell cycles through the glyphs on user demand. The layout of the bitmap must be a set of glyphs placed side by side in a line, all equal width and equal height. This is the same layout as the glyphs in the bitmaps for VCL's TSpeedButton objects.

TOvcCellGlyphs has a default state that uses the three normal checkbox glyphs (unchecked, checked, and grayed). This bitmap is part of the resource file for the table and is linked automatically into your application.

The glyphs are painted onto the table with the BrushCopy method of the VCL's TCanvas class, so you can specify a "transparent" color as the bottom left pixel of each glyph. When the glyph is painted, all pixels of this color in the glyph are changed to the background color of the canvas on which the glyph is drawn. See the VCL documentation for more details.

Hierarchy

TPersistent (VCL)

    TOvcCellGlyphs (OvcTGRes)

Properties

ActiveGlyphCount	GlyphCount
Bitmap	IsDefault

30

Reference Section

ActiveGlyphCount property

```
property ActiveGlyphCount : Integer
```

↳ The number of glyphs in the bitmap that are cycled through at run time.

ActiveGlyphCount allows you to display a subset of the total number of glyphs when the glyph cell is edited at run time. For example, a checkbox has a total of three glyphs that can be displayed (unchecked, checked and grayed). If you do not want the grayed checkbox to be displayed, set ActiveGlyphCount to 2.

ActiveGlyphCount is forced to be greater than zero, and less than or equal to GlyphCount.

See also: BitMap, GlyphCount

BitMap property

```
property BitMap : TBitmap
```

Default: the set of checkbox glyphs

↳ The collection of glyphs as a single bitmap.

When you assign a new bitmap to BitMap, the object resets the GlyphCount and ActiveGlyphCount properties. It does this by assuming that all the glyphs are square and the same size and by doing a simple calculation. The IsDefault property is automatically changed to False.

If you set BitMap to nil, IsDefault is set to True and the default bitmap resource is reallocated to the object.

See also: ActiveGlyphCount, GlyphCount, IsDefault

GlyphCount property

```
property GlyphCount : Integer
```

↳ The total number of glyphs in the bitmap.

GlyphCount specifies the total number of glyphs that could be displayed when the glyph cell is edited at run time. The actual number that will be displayed depends on ActiveGlyphCount.

Setting GlyphCount to zero or less has no effect.

See also: ActiveGlyphCount, BitMap

IsDefault property

```
property IsDefault : Boolean
```

Default: True

Indicates whether the object is in its default state.

When the object is in its default state, it refers to the checkbox glyphs. If you set IsDefault to True, the current bitmap resource is freed.

See also: BitMap

TOvcSparseArray Class

The TOvcSparseArray class is a one-dimensional sparse array implementation. A sparse array is an array where only the “used” elements have memory allocated to them. Unused elements are assumed to be empty and no memory is allocated to them. For example, if you declare an array of 1000 integer elements 4000 bytes of memory are allocated. If you use only the 789th element, you waste a lot of memory.

The TOvcSparseArray class mimics the VCL’s TList class, but is not descended from it. It uses many of the names of the TList methods. Like TList, it stores pointers only. Because unused elements in the sparse array are assumed to be nil, you must be careful if you store typecast long integers in a sparse array. Due to its internal structures and assumptions, TOvcSparseArray has a maximum limit of 320,000 pointers (this is defined as a constant called MaxSparseArrayItems in the OvcSpAry unit).

The TOvcSparseArray is used extensively by the Orpheus table for storing information about rows and individual cells. For individual cells, a sparse matrix (a sparse array of sparse arrays) is used.

TOvcSparseArray is not a data container that owns its data. The sparse array does not dispose its elements or call their destructors (if they have one). To do this, you must write a wrapper class that contains a sparse array. See the implementation of TOvcTableRows in the source code for an example.

See the source code for the TOvcTableRows class in the OvcTbRws unit for an example of how to use a sparse array.

Hierarchy

TObject (VCL)

  TOvcSparseArray (OvcSpAry)

Properties

ActiveCount	Count	Items

Methods

Add	Destroy	IndexOf
Clear	Exchange	Insert
Create	First	Last
Delete	ForAll	Squeeze

Reference Section

ActiveCount — run-time, read-only property

```
property ActiveCount : LongInt
```

↳ The number of non-nil elements in the sparse array.

ActiveCount is a measure of how populated the sparse array is. For example, if a sparse array has only one non-nil element at index 999, then ActiveCount is 1 and Count is 1000.

ActiveCount is implemented as a call to ForAll.

ActiveCount is a read-only property. It changes when you Add, Delete, or Insert items in the sparse array (or use the Items array property).

See also: Count, ForAll

Add — method

```
function Add(Item : Pointer) : LongInt;
```

↳ An element to the end of the sparse array.

Item is added to the sparse array and the new item pointer is placed at index Count, and Count is then incremented. Add returns the index of the added item.

If the array is at its maximum size already (MaxSparseArrayItems), an ESAEAtMaxSize exception is raised.

See also: Insert

Clear — method

```
procedure Clear;
```

↳ The sparse array.

After this call, both ActiveCount and Count are zero. Element pointers are not disposed of by this routine. You must dispose of the data.

See also: ActiveCount, Count

Count — run-time, read-only property

```
property Count : LongInt
```

↳ The nominal number of elements in the sparse array.

Count is equal to one more than IndexOf(Last). It is the theoretical number of elements in the array. For example, if a sparse array has one non-nil element at index 999, then Count is 1000 (elements 0-998 are assumed to be there but empty).

Count is a read-only property. It changes when you Add, Delete, or Insert items in the sparse array (or use the Items array property). Reading Count is resolved by the compiler to accessing a field; no calculations are made.

See also: ActiveCount, IndexOf, Last

Create — constructor

```
constructor Create;
```

↳ A sparse array.

Create allocates the internal structures for the sparse array, and sets the internal variables so that Count and ActiveCount return zero. The constructor initially allocates 40 bytes.

See also: Destroy

Delete — method

```
procedure Delete(Index : LongInt);
```

↳ An element from the sparse array.

The element at Index is deleted from the sparse array. All the elements with larger indexes have their indexes decremented, and Count is decremented.

If Index is less than 0 or greater than the maximum number of elements in the sparse array (MaxSparseArrayItems), an ESAEOutOfBounds exception is raised. If Index is greater than or equal to Count, nothing is done.

The data pointed to by the element is not disposed.

See also: Clear, Count

Destroy destructor

```
destructor Destroy;
```

↳ Disposes of a sparse array.

Destroy calls the Clear method to clear the sparse array and then disposes of all the internal structures allocated for the sparse array. Data objects in the array are not disposed by this destructor.

See also: Create

Exchange method

```
procedure Exchange(Index1, Index2 : LongInt);
```

↳ Switches the locations of two sparse array elements.

If Index1 or Index2 is less than 0 or greater than the maximum number of elements in a sparse array (MaxSparseArrayItems), an ESAEOutOfBounds exception is raised.

First method

```
function First : Pointer;
```

↳ Returns the first non-nil pointer in the sparse array.

If the array has no non-nil elements (ActiveCount is zero), First returns nil.

First is implemented as a call to ForAll.

See also: ActiveCount, ForAll, Last

ForAll method

```
function ForAll(Action : TSparseArrayFunc;
  Backwards : Boolean; ExtraData : Pointer) : LongInt;
```

```
TSparseArrayFunc = function(Index : LongInt;
  Item : Pointer; ExtraData : Pointer) : Boolean;
```

↳ Iterates through all the active elements in the sparse array.

Action is the routine that is called for each active (i.e., non-nil) element in the sparse array. It must be of type TSparseArrayFunc. Index is the index in the array. Item is the pointer itself. ExtraData is a pointer to a user-defined structure that you pass to ForAll. The Action function returns True if the ForAll iteration should continue and False if ForAll should return immediately. Pass True for Backwards to iterate through the array backwards (from the highest index to the lowest index).

ExtraData is a pointer to a user-defined structure. ForAll passes this pointer unchanged to the Action routine. You can use it to declare any common data that must be used in the iteration and use it in the iterator routine.

ForAll is a function that returns the index of the element that caused the Action routine to return False. If no call to Action returned False, ForAll returns -1.

The combination of ForAll and your Action routine can perform traditional “for each element” processing (Action should always return True), find the first element that satisfies a condition (Action should return False if it finds the element that satisfies the condition), or find the last element that satisfies a condition (set Backwards to True).

IndexOf method

```
function IndexOf(Item : Pointer) : LongInt;
```

Returns the index of the specified item.

IndexOf iterates sequentially through the array looking for an element that is equal to the Item pointer. If an element is found, IndexOf returns that index number, otherwise it returns -1.

IndexOf is implemented as a call to ForAll.

See also: ForAll

Insert method

```
procedure Insert(Index : LongInt; Item : Pointer);
```

A new element at the specified position in the sparse array.

The Item pointer is inserted at Index and each element from that point forward, have their indexes incremented by one. Count is incremented by one.

If Index is less than 0 or greater than the maximum number of elements in the sparse array (MaxSparseArrayItems), an ESAEOutOfBounds exception is raised. If the array is at its maximum size already, an ESAEAtMaxSize exception is raised.

See also: Add, Delete

Items — run-time property

```
property Items[Index : LongInt] : Pointer
```

↳ The array of elements in the sparse array.

Items is the "front-end" to the sparse array. It maps an ordinary array onto the sparse array so that reading an element either returns the element pointer from the sparse array, or nil if the element is not present. Writing a pointer to the Items array will cause the sparse array to store the new pointer.

To remove an item from the array, set it to nil. This could cause the sparse array to call Squeeze to minimize the amount of heap space used.

If Index is less than 0 or greater than the maximum number of elements in the sparse array (MaxSparseArrayItems), an ESAEOutOfBounds exception is raised.

See also: Squeeze

Last — method

```
function Last : Pointer;
```

↳ Returns the last non-nil pointer in the sparse array.

If the array has no non-nil elements (ActiveCount is zero), Last returns nil.

Last is implemented as a call to ForAll.

See also: ActiveCount, First, ForAll

Squeeze — method

```
procedure Squeeze;
```

↳ Minimizes the amount of memory used by the sparse array.

After inserts and deletes are made in a sparse array, it might be using more memory than it needs (due to the internal algorithm used). Squeeze removes the unused memory, returning it to the heap.

Delete calls Squeeze automatically, but Insert does not. Writing to the Items array property causes Squeeze to be called under certain circumstances. Therefore you should not need to call Squeeze explicitly.

See also: Delete, Insert, Items

TOvcTableRows Class

TOvcTableRows is a class that holds all the styles for the rows in a table. There are two attributes that define a row's style: the row's height and whether it is hidden or not. At the simplest level, the TOvcTableRows class can be viewed as an array of row styles (this is, in fact, the List property). The rest of this section discusses the class in this manner. The array has a lower limit of zero and an upper limit that's defined by the Limit property. The properties and methods perform range checks on row numbers and an exception is raised if a row number is out of range.

TOvcTableRows encapsulates a sparse array (an instance of a TOvcSparseArray) to hold the style for each row. If there is no custom style for a row, it is assumed to be visible and have a default height. If the style for a row is changed from a custom style to be visible and have the default height, the class deletes the explicit row style for that row to minimize the total amount of memory required for the class.

Because the individual row styles are held in a TOvcSparseArray instance, you can specify individual characteristics only for the first 320,000 rows.

The table class (TOvcTable) creates a single instance of a TOvcTableRows class and interfaces it as a property called Rows.

Rows Editor

Nearly all the methods and properties supported by the TOvcTableRows class are accessible through the Rows Editor. This is made available to the TOvcTable component at design time through its Rows property or through the pop-up menu for the table. The Rows Editor lets you set the major attributes for the rows in your table at design time.

Previous row button

This button decrements the row number, and displays the style for that row.

Next row button

This button increments the row number, and displays the style for that row.

First row button

This button sets the row number to zero, and displays the style for that row.

Last row button

This button sets the row number to the maximum number of rows minus 1, and displays the style for that row.

Insert row button

This button creates a new row (not hidden and using the default height) and inserts it at the displayed row number. Rows whose number is equal to or greater than the inserted row have their number incremented. For example, inserting a row 4 means that the old row 4 becomes row 5, and so on.

Delete row button

This button deletes the row at the displayed row number. Rows whose number is greater than the deleted row have their number decremented. For example, deleting row 4 means that the old row 5 becomes row 4, and so on.

Row number

The number of the row whose data is displayed in the Selected row details box. Use the spin button (or the previous or next speedbuttons) to increment or decrement the row number.

Selected row details

The details for the row whose number is in the Row Number field. If Hidden is checked, the row is hidden. The height of a row can be either the default height for all the rows (Use default height radio button), or it can be a customized height just for this row (Use custom height radio button). When the Use default height radio button is selected, the height field is enabled and you can enter a new height for the row.

Overall row details

The details for all rows except those for which the style has been customized. Default Height is the default height for standard uncustomized rows. Maximum rows is the total number of rows tracked in the TOvcTableRows object. The Row number field is allowed to vary from 0 to Maximum rows - 1.

Apply button

Applies the selected row details to the row whose number is in the Row number field. The table underneath is altered automatically. In general it is not necessary to use the Apply button. Changes are applied automatically when required (for example, pressing the Next Row speedbutton causes the changed data for the previous row to be saved before showing the data for the next row).

Reset button

Resets all rows to the default (visible and default height). The change is visible immediately in the underlying table.

Done button

Applies the latest Selected row details to the row whose number is displayed and closes the dialog box.

Hierarchy

TObject (VCL)

TOvcTableRows (OvcTbRws)

Properties

Count	Hidden	RowIsSpecial
DefaultHeight	Limit	
Height	List	

Methods

Append	Delete	Insert
Clear	Exchange	Reset

Reference Section

Append method

```
procedure Append(const RS : TRowStyle);
```

```
PRowStyle = ^TRowStyle;
```

```
TRowStyle = record
  Height : Integer;
  Hidden : Boolean;
end;
```

Adds a new row style.

Append adds a row to the end of the rows array List. The added row has a number equal to the current value of Limit, and Limit is incremented by one.

To append a row with a default style, set RS.Height to -1 and RS.Hidden to False. However, it is easier to increment the Limit property.

See also: Limit

Clear method

```
procedure Clear;
```

Removes all the explicit row styles.

After calling Clear, all rows have the same height (DefaultHeight) and are visible.

Count run-time, read-only property

```
property Count : TRowNum
```

The number of rows with an explicit style.

Count is the number of rows that have a style that is not the default style.

See also: Limit

DefaultHeight property

```
property DefaultHeight : Integer
```

Default: 30

The height in pixels for all rows that do not have an explicit height.

See also: Height

Delete method

```
procedure Delete(RowNum : TRowNum);
```

A row style.

The row style is removed from the array and rows below RowNum have their numbers decremented. Limit is reduced by one.

See also: Limit, Insert

Exchange method

```
procedure Exchange(const RowNum1, RowNum2 : TRowNum);
```

Switches the locations of two row styles.

After a call to this routine, RowNum1 will have RowNum2's style and vice versa.

Height run-time property

```
property Height[RowNum : TRowNum] : Integer
```

An array of row heights, one for each row.

The Height array property provides access to each row's height individually. The minimum value for a row height is 5.

See also: DefaultHeight

Hidden run-time property

```
property Hidden[RowNum : TRowNum] : Boolean
```

An array of flags that indicate whether each row is hidden or visible.

The Hidden array property provides access to each row's hidden attribute individually.

Insert method

```
procedure Insert(
  const RowNum : TRowNum; const RS : TRowStyle);
```

```
PRowStyle = ^TRowStyle;
```

```
TRowStyle = record
  Height : Integer;
  Hidden : Boolean;
end;
```

↳ A new row style.

The row style is inserted into the array at RowNum and the rows at RowNum and below have their numbers incremented. Limit is incremented by one. To insert a row with a default style, set RS.Height to -1 and RS.Hidden to False.

Limit property

```
property Limit : TRowNum
```

Default: 10

↳ The maximum number of rows.

Limit is the total number of rows that the object tracks. The table's RowLimit property reads and writes this value.

If you reduce Limit, the row styles between the new and old values are not lost or disposed of. If you later increase Limit, they will be available again.

Limit has a minimum value of 1.

See also: Count

List run-time property

```
property List[RowNum : TRowNum] : TRowStyle
```

```
TRowStyle = record
  Height : Integer;
  Hidden : Boolean;
end;
```

↳ The array of row styles.

The Height field of the returned TRowStyle value is the actual height of the row. If the row has the default style, this property returns DefaultHeight for Height.

See also: DefaultHeight, Height, Hidden

Reset method

```
procedure Reset(RowNum : TRowNum);
```

Sets a row style to the default.

The row's style is set to the default state (visible and default height).

See also: List

RowIsSpecial run-time, read-only property

```
property RowIsSpecial [RowNum : TRowNum] : Boolean
```

An array of flags that indicate whether a row has an explicit style.

RowIsSpecial is a read-only array property. If the row has an explicit style, the value of RowIsSpecial is True, otherwise it is False. The property is used by the table's streaming code to determine whether a row has any data to stream.

TOvcTableColumns Class

The TOvcTableColumns class is an array of all the columns for a table. Each of the elements of the array is an instance of a TOvcTableColumn. The array has a lower limit of zero and an upper limit that's returned by the Count property. The properties and methods perform range checks on column numbers and the operation is ignored if a column number is out of range.

The underlying data structure for the TOvcTableColumns class is a TList and the maximum number of columns is limited to 16,384.

Columns Editor

Nearly all the methods and properties supported by the TOvcTableColumns class are accessible through the Columns Editor. This is made available to the TOvcTable component at design time through its Columns property or by use of the pop-up menu for the table. The Columns Editor lets you set the major attributes for the columns in your table at design time.

Previous column button

This button decrements the column number, and displays the style for that column.

Next column button

This button increments the column number, and displays the style for that column.

First column button

This button sets the column number to zero, and displays the style for that column.

Last column button

This button sets the column number to the maximum number of columns minus 1, and displays the style for that column.

Insert column button

This button creates a new column (not hidden, using the default width of 150, and a nil DefaultCell) and inserts it at the displayed column number. Columns whose number is equal to or greater than the inserted column number have their number incremented. For example, inserting a column 4 means that the old column 4 becomes column 5, and so on.

Delete column button

This button deletes the column at the displayed column number. Columns whose number is greater than the deleted column have their number decremented. For example, deleting column 4 means that the old column 5 becomes column 4, and so on.

Column number

The number of the column whose data is displayed in the Column details box. Use the spin button (or the previous or next buttons) to increment or decrement the column number.

Column details

The details for the column whose number is in the Column number field. The DefaultCell field is a listbox of the names of all the cell components that are present in the parent form, plus a special item called (None). Select the cell component to be used by this column and the cell component is linked to both the column and the table. Selecting (None) means that the DefaultCell property is set to nil. If Hidden is checked, the column is hidden. Width is the width of the column.

Apply button

Applies the selected column details to the column whose number is in the Column number field. The table underneath is altered automatically. In general it is not necessary to use the Apply button. Changes are applied automatically when required (for example, pressing the Next Column button causes the changed data for the previous column to be saved before showing the data for the next column).

Done button

Applies the latest Column details to the column whose number is displayed and closes the dialog box.

Hierarchy

TObject (VCL)

  TOvcTableColumns (OvcTbCls)

Properties

Count	Hidden	Table
DefaultCell	List	Width

30

Methods

Append	Delete	Insert
Clear	Destroy	
Create	Exchange	

Reference Section

Append method

```
procedure Append(C : TOvcTableColumn);
```

↳ Adds a new column to the end of the array.

The new column number will be the current value of Count, and then Count is incremented by one. The column object C must have already been created and must be of the same class type (or a descendant class) as that specified by the Create constructor. If C has a different table owner than the column array, the operation is ignored.

The following example appends a new column to the end of the columns array.

```
{Columns is a property of the table}
MyColumn := TOvcTableColumn.Create(OvcTable1);
with MyColumn do begin
  Width := 60;
  DefaultCell := OvcTCString1;
end;
OvcTable1.Columns.Append(MyColumn);
```

See also: Create, Delete, Insert

Clear method

```
procedure Clear;
```

↳ Removes all the columns from the columns array.

After Clear is called, Count is set to zero. The column objects in the columns array are disposed, but the cell components referenced by their DefaultCell properties are not.

Count run-time property

30

```
property Count : Integer;
```

↳ The number of column objects in the column array.

The elements in the array have indexes that range from 0 to Count-1.

Reading this property returns the current number of column objects in the internal array. Writing the property will do one of three things. If the new value equals the current one, nothing is done. If the new value is greater than the old one, the required number of default column objects are created and appended to the internal array (for a definition of the values used by a default column object see the TOvcTableColumn properties). If the new value is less than the current one, the required number of column objects are disposed.

Create constructor

```
constructor Create(ATable : TOvcTableAncestor;
  ANumber : Integer; AColumnClass : TOvcTableColumnClass);
```

```
TOvcTableColumnClass = class of TOvcTableColumn;
```

↳ A new columns array.

Create allocates a new array with ANumber columns. The DefaultCell property for all the columns is nil. ATable is the owner of the columns array and all the automatically created columns. AColumnClass is the class type of the column objects that will be used in the array: all columns that are appended or inserted into the array must be of this class type.

See also: Append, Insert, List

DefaultCell run-time property

```
property DefaultCell[ColNum : TColNum] : TOvcBaseTableCell
```

↳ An array of the default cells for the columns.

This array property provides a convenient way to access all the default cells for the column objects.

Delete method

```
procedure Delete(ColNum : TColNum);
```

↳ A column object from the columns array.

The deleted column object is destroyed (although its default cell is not). Columns with an index greater than ColNum are moved up to replace the deleted column.

The following example deletes the sixth column from the columns array.

```
  {Columns is a property of the table}
  OvcTable1.Columns.Delete(5);
```

See also: Append, Insert

Destroy destructor

```
destructor Destroy;
```

↳ A columns array.

Destroy calls Clear and then disposes of the array.

See also: Clear

Exchange method

```
procedure Exchange(ColNum1, ColNum2 : TColNum);
```

↳ Two column objects in the column array.

The two indexes specified by ColNum1 and ColNum2 must be in range for the array (i.e. not less than zero and less than the current value of Count for the array), and must not be equal. If so, the method swaps over the elements for ColNum1 and ColNum2. If not, nothing is done and in particular no exception is raised.

The following example exchanges the second and third column from the columns array.

```
  {Columns is a property of the table}
  OvcTable1.Columns.Exchange(1, 2);
```

Hidden run-time property

```
property Hidden[ColNum : TColNum] : Boolean
```

↳ An array of the visibility flags for the columns.

This array property provides a convenient way to access all the hidden properties for the column objects.

Insert method

```
procedure Insert(const ColNum : TColNum; C : TOvcTableColumn);
```

↳ A new column into the columns array.

The columns starting at ColNum are moved down one position to make room for the new column. The new column object should already have been created.

If the column C has a different table owner than the columns array or if ColNum is out of range, the operation is ignored. It will also be ignored if it is of a different class type than that specified in the Create constructor.

The following example inserts a new sixth column in the columns array.

```
  {Columns is a property of the table}
  MyColumn := TOvcTableColumn.Create(OvcTable1);
  with MyColumn do begin
    Width := 60;
    DefaultCell := OvcTCString1;
  end;
  OvcTable1.Columns.Insert(5, MyColumn);
```

See also: Append, Delete

List <span>run-time property</span>

```
property List[ColNum : TColNum] : TOvcTableColumn
```

↳ The array of column objects.

If you try to set an element of the array to a column that has a different table owner than the columns array, the operation is ignored. Similarly an attempt to set an element of the array to a column of a different class type than that specified in the Create constructor will be ignored.

Table — run-time, read-only property

```
property Table : TOvcTableAncestor
```

↳ The owner of the columns array.

Table is a read-only property whose value is set by the Create constructor. Once a column array is created, you cannot change its owner.

Width — run-time property

```
property Width[ColNum : TColNum] : Integer
```

↳ An array of the widths for the columns.

This array property provides a convenient way to access all the widths for the column objects.

TOvcTableCells Class

The TOvcTableCells class encapsulates a sparse matrix of individual cell attributes and objects. It is provided for tables that do not fit the “one row equals one record, one column equals one field” mold. To accommodate this more “freestyle” display, the table has a property called Cells that is an instance of the TOvcTableCells class. This class defines individual cell objects and cell attributes on a cell-by-cell basis. You can then define each cell individually without having to conform to the default cell for the column.

There is no design-time editor for the TOvcTableCells class, it is for run-time use only.

In the TOvcTableCells properties and methods, a cell is identified by its row number (RowNum) and column number (ColNum).

Hierarchy

TObject (VCL)

TOvcTableCells (OvcTCell)

Properties

Access	Cell	Font
Adjust	Color	

Methods

DeleteCol	ExchangeRows	ResetCell
DeleteRow	InsertCol	ResolveFullAttr
ExchangeCols	InsertRow	

Reference Section

Access — run-time property

```
property Access[
  RowNum : TRowNum; ColNum : TColNum] : TOvcTblAccess;
```

Is a matrix of cell access rights.

If no special access rights have been set for a cell, this property returns otxDefault. If special access rights have been set for a cell, you can clear them by setting the cell's Access property to otxDefault.

The following example sets the access rights for the cell at row 5, column 2 to otxReadOnly if they are currently set to otxNormal, otherwise the rights are set to otxNormal.

```
  with OvcTable1.Cells do
    if Access[5,2] = otxNormal then
      Access[5,2] := otxReadOnly
    else
      Access[5,2] := otxNormal;
```

See also: TOvcBaseTableCell.Access

Adjust — run-time property

```
property Adjust[
  RowNum : TRowNum; ColNum : TColNum] : TOvcTblAdjust
```

Is a matrix of data adjustment attributes.

If no special adjustment attribute has been set for a cell, this property returns otaDefault. If a special adjustment attribute has been set for a cell, you can clear it by setting the cell's Adjust property to otaDefault.

See also: TOvcBaseTableCell.Adjust

Cell — run-time property

```
property Cell[
  RowNum : TRowNum; ColNum : TColNum] : TOvcBaseTableCell
```

↳ Is a matrix of cell objects.

If no special cell object has been set for a cell, this property returns nil. If a special cell object has been set for a cell, you can clear it by setting the cell's Cell property to nil.

The display information in a special cell object is overridden by any special display information for that same cell. For example, assume the cell at row 1 column 1 has a special cell component that normally uses white as its background. If you also set the color for the cell to black (via the Color matrix property) then this will override the color in the cell object at run- time and the cell will be displayed in black.

The following example creates a new cell component (an instance of TOvcTCString) and then sets the cell at row 15, column 4 to it.

```
  MyCell := TOvcTCString.Create(Form1);
  OvcTable1.Cells.Cell[15,4] := MyCell;
```

Color — run-time property

```
property Color[
  RowNum : TRowNum; ColNum : TColNum] : TColor
```

↳ Is a matrix of cell background colors.

If no special color has been set for a cell, this property returns clOvcTableColorDefault (2FFFFFF, defined in OvcTCmmn). If a special color has been set for a cell, you can clear it by setting the cell's Color property to clOvcTableColorDefault. This color overrides any special color calculated due to the cell's position (for example, a locked cell).

DeleteCol — method

```
procedure DeleteCol(ColNum : TColNum);
```

↳ Deletes a column from the sparse matrix of special cells.

All special style attributes for the cells in the column are discarded (in particular the font is destroyed ; the cell object is not). Columns with numbers greater than ColNum are moved up one, so that, for example, deleting column 4 will result in column 5 moving up to become column 4, and so on. In other words the cells in the new column 4 will have the style of the cells in the old column 5 and so on.

DeleteRow method

```
procedure DeleteRow(RowNum : TRowNum);
```

Deletes a row from the sparse matrix of special cells.

All special style attributes for the cells in the row are discarded (in particular the font is destroyed ; the cell object is not). Rows with numbers greater than RowNum are moved up one, so that, for example, deleting row 4 will result in row 5 moving up to become row 4, and so on. In other words the cells in the new row 4 will have the style of the cells in the old row 5 and so on.

ExchangeCols method

```
procedure ExchangeCols(ColNum1, ColNum2 : TColNum);
```

Swaps two columns in the sparse matrix of special cells.

ExchangeRows method

```
procedure ExchangeRows(RowNum1, RowNum2 : TRowNum);
```

Swaps two rows in the sparse matrix of special cells.

Font run-time property

```
property Font[
  RowNum : TRowNum; ColNum : TColNum] : TFont
```

Is a matrix of cell fonts.

If no special font has been set for a cell, this property returns nil. If a special font has been set for a cell, you can clear it by setting the cell’s Font property to nil.

InsertCol method

```
procedure InsertCol(ColNum : TColNum);
```

Inserts a column into the sparse matrix of special cells.

All special cell styles whose column number is greater than ColNum are moved along by one position. A set of blank cell styles are inserted as the new column. For example inserting a new column 4 would result in the cells in the new column 5 having the style of the cells in the old column 4 and so on.

InsertRow method

```
procedure InsertRow(RowNum : TRowNum);
```

↳ Inserts a row into the sparse matrix of special cells.

All special cell styles whose row number is greater than ColNum are moved along by one position. A set of blank cell styles are inserted as the new row. For example inserting a new row 4 would result in the cells in the new row 5 having the style of the cells in the old row 4 and so on.

ResetCell method

```
procedure ResetCell(RowNum : TRowNum; ColNum : TColNum);
```

↳ Deletes any special display information for a cell.

After ResetCell is called, the cell is displayed according to the default cell object of the column. All special attributes for the cell (access, adjust, cell object, color and font) are discarded (note that the cell object will not be destroyed, whereas the font will be).

ResolveFullAttr method

```
procedure ResolveFullAttr(RowNum : TRowNum;
  ColNum : TColNum; var SCA : TOvcSparseAttr);
```

```
TOvcSparseAttr = record
  scaAccess : TOvcTblAccess;
  scaAdjust : TOvcTblAdjust;
  scaColor  : TColor;
  scaFont   : TFont;
  scaCell   : TOvcTableCellAncestor;
end;
```

↳ Gets a cell's attributes from the sparse matrix.

This method is used internally by the table to get the full set of attributes for a cell. If no special attributes exist for the cell, default values are returned. This method exists because it is faster than separate accesses to the individual property arrays.

See also: Access, Adjust, Cell, Color, Font

TOvcTableCellAncestor Class

The TOvcTableCellAncestor class is the true ancestor to the Orpheus table cell types. It exists to provide low-level interaction between the table and its cells. This is a true abstract ancestor. You should not create any instances of this class, but use one of the descendants instead.

The class will not be described any further in this manual, for details please see the source code in OVCTCMMN.PAS.

TOvcBaseTableCell Class

The TOvcBaseTableCell class is the functional ancestor to the Orpheus table cell types, in the sense of painting and editing services. All cells in the table use instances of this class and its descendants to display and edit the table data. This is aTrueabstract ancestor. You should not create any instances of this class, but use one of the descendants instead.

The TOvcBaseTableCell class is described here because if you define any cell descendants you will need know about the flow of data between a cell and its owning table. You might even have to descend from this class rather than its two main children, the TOvcTCBaseString class (for string cells) and the TOvcTCBaseBitMap (for bitmap cells).

A cell object belongs to only one table. You cannot share cell objects between tables. However, a table can use the same cell object many times, through different references. For example, two columns could use the same cell object as their default cell. Every time a cell is linked to a table (no matter through which route) its reference count is incremented. Similarly, every time a cell object link is broken, the reference count is decremented. When the count is zero, the cell object is not linked to the table at all (and the internal pointer to the table is reset). For example, deleting a column causes the column object to be destroyed, and the column's destructor decrements the default cell object's reference count. Setting a default cell for a column causes the current default cell's reference count to be decremented and the new cell's reference count to be incremented.

A cell object stores the display information for the cells it refers to. It does not store the actual data. The table provides the cell object's paint method with a pointer to the data to display when the cell is painted. For example, if the table is repainting column 3, the cell object is called to paint the cell at row 0 column 3 with the data for that cell, and then it is called to paint the cell at row 1 column 3 with the data for that cell, and so on. The display information that a cell object stores is the background color, font data, access rights (normal, read-only, invisible), and adjustment attribute (where to paint the data in the cell).

Although a cell object has the color, font, and display information, they are only treated as defaults. The table can override the cell object's display information (for example, the cell could be in a locked part of the table, and hence have a different color and font). To determine the actual display information for a cell address, the cell interrogates the table for any overrides and supplies its own values as defaults (see the ResolveAttributes method). In addition, the table publishes an event that also gets triggered so you can even install a handler to override the display information provided by the table. When a cell needs to be painted, the process is:

1. The table calls the cell object's ResolveAttributes method.

2. ResolveAttributes builds a record with the default display information for the cell object. It then calls a method of the table so that the table gets a chance to override any or all this data.
3. If you have one installed, this method of the table calls your event handler, which can override any or all the display information.
4. ResolveAttributes then makes sure that any default values still present are set properly.

When the user edits data in a cell, the table calls the StartEditing method in the cell object to start the editing process. StartEditing must create a control (or activate it if it is pre-created but quiescent and hidden) that actually edits the data. The editing control must be a windowed control (it must have a window handle). The table makes sure that this editing control is placed properly over the cell, that it gets the focus when required, and that it is hidden or visible when necessary (it does some of this through methods of the cell object). The editing control must pass all the keystrokes it receives through the WM_KEYDOWN message to the table. The table categorizes the keystroke message into one of three sets:

- The keystroke must be processed by the table (for example, <Esc> to abandon editing the cell). The original message is ignored by the editing control and it is passed back to the table.
- The keystroke could be processed by the table if the editing control does not recognize it (for example, <PgDn> is not used in a normal Windows single-line edit control, but the table uses it to page the table data down). The editing control must decide whether to process the message itself or pass it back to the table.
- The keystroke would be ignored by the table anyway. The editing control is free to take any action on the keystroke.

When the table determines that the editing session is over for the cell (for example, the user clicks elsewhere in the table), the CanSaveEditedData method of the cell object is called. The cell object must determine, in conjunction with its editing control, whether the edited data can be saved. Generally this means a test to see if the edited data is valid. If it is, CanSaveEditedData returns True, and the table calls the StopEditing method, which causes the cell object to save the edited data (if required) and destroy or hide the editing control.

The table (or rather, the programmer) can ask the cell component to save the edited data at any time by calling the SaveEditedData method of either the table or the cell component. The difference between the two is that the table's method will ensure that the data is valid first by calling the CanSaveEditedData method of the cell component, and then only saving the data if this call returned True.

The components that actually edit the data are not documented because they are simple descendants of VCL and Orpheus controls. For more details, see the source code.

Hierarchy

TComponent (VCL)

  TOvcTableCellAncestor (OvcTCmmn) 921

    TOvcBaseTableCell (OvcTCell)

Properties

About	Color	TableColor
AcceptActivationClick	Font	TableFont
Access	Margin	TextHiColor
Adjust	References	TextStyle
CellEditor	Table	

Methods

CanSaveEditedData	EditMove	SendKeyToTable
CanStopEditing	FilterTableKey	StartEditing
DecRefs	IncRefs	StopEditing
DoOwnerDraw	Paint	TableWantsEnter
EditHandle	ResolveAttributes	TableWantsTab
EditHide	SaveEditedData	Updated

Events

OnOwnerDraw	OnXxx

Reference Section

About — read-only property

```
property About : string
```

↳ Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

AcceptActivationClick — property

```
property AcceptActivationClick : Boolean
```

Default: False

↳ Determines whether a click on a cell is passed to the cell editor.

If the table is in always editing mode and AcceptActivationClick is True, clicking on a cell will cause the active cell to move there, the cell to be switched into edit mode, and the mouse click event to be passed to the cell editor. This means for example that check box cells can be clicked on and the checkbox state changed with one click.

If the table is not in always editing mode and AcceptActivationClick is True for the active cell, clicking on the active cell will cause the cell to be switched into edit mode, and the mouse click event to be passed to the cell editor.

```
property Access : TOvcTblAccess
```

```
TOvcTblAccess = (
  otxDefault, otxNormal, otxReadOnly, otxInvisible);
```

Default: otxDefault

Determines the cell access rights.

The possible values for Access are:

Value	Description
otxDefault	The cell takes its access rights from the table.
otxNormal	The cell can be read and edited.
otxReadOnly	The cell can be read but not edited.
otxInvisible	The cell cannot be read or edited and is invisible (the cell is displayed as a blank rectangle in the background color).

The table can override the value of Access at run time when displaying or editing a cell. A call to ResolveAttributes determines the correct value at run time. If you want the cell to have the table's access rights value, use otxDefault.

See also: Adjust, Color, Font, ResolveAttributes

```
property Adjust : TOvcTblAdjust
```

```
TOvcTblAdjust =(otaDefault, otaTopLeft, otaTopCenter,
  otaTopRight, otaCenterLeft, otaCenter, otaCenterRight,
  otaBottomLeft, otaBottomCenter, otaBottomRight);
```

Default: otaDefault

Defines where the data is displayed in the cell.

The possible values for Adjust are:

Value	Description
otaDefault	The cell takes its adjustment value from the table.
otaTopLeft	The data is displayed in the top left corner of the cell.
otaTopCenter	The data is centered horizontally at the top of the cell.
otaTopRight	The data is displayed in the top right corner of the cell.
otaCenterLeft	The data is centered vertically at the left of the cell.
otaCenter	The data is centered vertically and horizontally in the cell.
otaCenterRight	The data is centered vertically at the right of the cell.
otaBottomLeft	The data is displayed in the bottom left corner of the cell.
otaBottomCenter	The data is centered horizontally at the bottom of the cell.
otaBottomRight	The data is displayed in the bottom right corner of the cell.

The table can override the value of Adjust at run time when displaying or editing a cell. A call to ResolveAttributes determines the correct value at run time. If you want the cell to have the table's adjustment value, use otaDefault.

See also: Access, Color, Font, ResolveAttributes

CanSaveEditedData virtual method

```
function CanSaveEditedData(SaveValue : Boolean) : Boolean;
```

↳ Determines whether the edited data is valid and could be saved.

CanSaveEditedData is called by the table either when the user is finished editing the cell, or the data needs to be saved. This method gives the cell object a chance to make sure that the edited data is valid. SaveValue is True if the edited data will be saved, or False if the edited data will be discarded. CanSaveEditedData can use this value to determine whether to do any validation or not. It can display a warning or error message box if appropriate. If the data is valid and can be saved, CanSaveEditedData returns True. If the data is invalid, CanSaveEditedData returns False. If the table is attempting to stop editing for the cell, it then calls the cell object's StopEditing method to actually save the data. If the data is going to be saved (but editing will continue) the cell object's SaveEditedData will be called by the table's method of the same name. The default implementation of CanSaveEditedData always returns True.

The following example shows how CanSaveEditedData could be written for a cell component that uses a TEdit-based editing control for entering a numeric field.

```
function TMyCellComponent.CanSaveEditedData
  (SaveValue : Boolean) : Boolean;
var
  V, ec : Integer;
begin
  Result := True;
  if SaveValue then begin
    {CellEditor is a TEdit descendant}
    Val(TEdit(OvcTCString1.CellEditor).Text, V, ec);
    if (ec <> 0) then
      {the text in the edit field is not numeric}
      Result := False;
  end;
end;
```

See also: SaveEditedData, StartEditing, StopEditing

CanStopEditing method

```
function CanStopEditing(SaveValue : Boolean) : Boolean;
```

↳ Determines whether the edited data is valid and editing can be stopped.

CanStopEditing is only present for compatibility with Orpheus 1.0 code and should not be used. All it does is make a call to CanSaveEditedData.

See also: CanSaveEditedData

CellEditor run-time, read-only property

```
property CellEditor : TControl
```

↳ The control that is doing the editing of the cell data.

The default value of CellEditor is nil. Descendant cell components override the read access method (GetCellEditor) for CellEditor to return the instance of the control that is editing the cell data. CellEditor returns a valid control instance only if the cell is being edited. CellEditor exists so that the table's OnEndEdit event has a way of getting to the edited data in order to validate it (OnEndEdit gets the instance of the cell component).

See also: TOvcTable.OnEndEdit

Color property

```
property Color : TColor
```

Default: clBtnFace

↳ The background color for the cell.

The table can override this value at run time when displaying or editing a cell. A call to ResolveAttributes determines the correct value at run time. To use the table's color as the background color, set the TableColor property to True.

See also: Access, Adjust, Font, ResolveAttributes, TableColor

DecRefs method

```
procedure DecRefs;
```

↳ Decrements the reference count.

This routine is called when a cell object is unlinked from a table or from an object that belongs to a table (for example, a column). The number of references is decremented by one. If it reaches zero, then the internal pointer to the table is set to nil. For an example of how to use DecRefs, see the IncRefs method.

See also: IncRefs, References

EditHandle abstract method

```
function EditHandle : THandle; virtual; abstract;
```

↳ Returns the handle of the editing control.

When you create a descendant of a cell component, this method must be overridden. EditHandle is called between a call to StartEditing and a call to StopEditing. It is called by the table to obtain the handle of the editing control.

EditHide abstract method

```
procedure EditHide;
```

↳ Hides the editing control.

When you create a descendant of a cell component, this method must be overridden.

EditHide is called between a call to StartEditing and a call to StopEditing. It is called by the table to hide the editing control. It could be used when the user is viewing a page of the table where the cell being edited is not present.

The following example shows how the EditHide method is implemented for a normal cell component, in this case a TOvcTCCustomGlyph component. FEdit is the cell editor object. The call to the Windows API SetWindowPos routine causes the cell editor object to be hidden.

```
procedure TOvcTCCustomGlyph.EditHide;
begin
  if Assigned(FEdit) then
    with FEdit do
      WinProcs.SetWindowPos(Handle, HWND_TOP, 0, 0, 0, 0,
                            SWP_HIDEWINDOW or SWP_NOREDRAW or
                            SWP_NOZORDER);
end;
```

See also: EditMove

```
procedure EditMove(CellRect : TRect); virtual; abstract;
```

↳ Moves, resizes, and shows the editing control.

When you create a descendant of a cell component, this method must be overridden.

EditMove is called between a call to StartEditing and a call to StopEditing. It is called by the table to resize and move the editing control over the cell that is being edited. The cell object must resize and move the editing control and make it visible, if necessary. If this doesn't occur, the underlying cell is not repainted.

The following example shows how the EditMove method is implemented for a normal cell component, in this case a TOvcTCCustomGlyph component. FEdit is the cell editor object. The call to the Windows API SetWindowPos routine causes the cell editor to be moved to the given rectangle, grown (or shrunk) to fit and to be shown. The calls to InvalidateRect and UpdateWindow force the cell editor to repaint itself.

```
procedure TOvcTCCustomGlyph.EditMove(CellRect : TRect);
var
  EditHandle : hWnd;
begin
  if Assigned(FEdit) then begin
    EditHandle := FEdit.Handle;
    with CellRect do
      WinProcs.SetWindowPos(EditHandle, HWND_TOP, Left, Top,
                            Right-Left, Bottom-Top,
                            SWP_SHOWWINDOW or SWP_NOREDRAW or
                            SWP_NOZORDER);
    InvalidateRect(EditHandle, nil, false);
    UpdateWindow(EditHandle);
  end;
end;
```

See also: EditHide

FilterTableKey virtual method

```
function FilterTableKey(
  var Msg : TWMKey) : TOvcTblKeyNeeds; virtual;
```

```
TOvcTblKeyNeeds = (otkDontCare, otkWouldLike, otkMustHave);
```

↳ Passes the keystroke message to the table for filtering.

The editing control calls its cell object's FilterTableKey for each WM_KEYDOWN message that it receives. FilterTableKey then calls a method called FilterKey in the cell's owning table. The table decides whether it wants the keystroke or whether it would allow the cell's editing control to use it. The return values are:

Value	**Description**
otkDontCare	The table does not use the key and would ignore it anyway.
otkWouldLike	The table does use the key, but is willing to let the editing control decide whether it wants it or not.
otkMustHave	The table must have the keystroke.

The editing control can ignore the otkMustHave replies, and use the keystroke as it sees fit. However, the only two keystrokes in the "must have" category are <Esc> (abandon cell edit) and <F2> (save the data and stop editing), and to ignore these would imperil the standard user interface for the table.

The otkWouldLike category includes keys like the arrow keys and paging keys. Of course, the editing control could use some of these anyway. For example, when the right arrow key is used in a normal Windows edit control, you expect to move between characters in the string and not between cells.

Once the editing control gets a reply from the table, it can pass the message to the table for actual processing by calling SendKeyToTable.

The following example shows how the FilterTableKey method is implemented in the TOvcBaseTableCell class. This is generally sufficient for most cell components. FTable is the internal field to TOvcBaseTableCell that holds the owning table. Since FTable is of type TOvcTableAncestor (which has no explicit methods), it must be typecast to a TOvcTable first.

```
function TOvcBaseTableCell.FilterTableKey(
  var Msg : TWMKey): TOvcTblKeyNeeds;
begin
  if not Assigned(FTable) then
    Result := otkDontCare
  else
    Result := TOvcTable(FTable).FilterKey(Msg);
end;
```

See also: SendKeyToTable

Font property

```
property Font : TFont
```

Default: the standard font defined by VCL

↳ The font for string data displayed in the cell.

The table can override this value at run time when displaying or editing a cell. A call to ResolveAttributes determines the correct value at run time. This property only makes sense for string type cell components.

See also: Access, Adjust, Color, ResolveAttributes, TableFont

IncRefs method

```
procedure IncRefs;
```

↳ Increments the reference count.

This method is called when a cell object is linked to a table or to an object that belongs to a table. For example, when the DefaultCell property for a new column is set, the IncRefs method for the cell component that is being used is called. If the References property for the cell component was 0 before the call, this is the first time that this cell has been used by a table and it is up to the calling routine to set the Table property for the cell to link it to the correct table.

The classes that make up the Orpheus table call this method when required. It is documented in case you need to add extra functionality to a descendant of a TOvcTable.

The following example shows how the IncRefs and DecRefs methods are used in the write access method for the DefaultCell property of the TOvcTableColumn class. FDelCell is the field “behind” the DefaultCell property. BTC is the new cell component to which DefaultCell is being set.

```
if Assigned(FDefCell) then
  FDefCell.DecRefs;
FDefCell := BTC;
if Assigned(FDefCell) then begin
  if (FDefCell.References = 0) then
    FDefCell.Table := FTable;
  FDefCell.IncRefs;
end;
```

See also: DecRefs, References

Margin property

```
property Margin : Integer
```

Default: 4

↳ The number of pixels to leave between the data drawn in the cell and the cell boundary.

The cell object’s Paint method uses this value to position the data within the cell. The margin is applied to all sides of the cell—half on the top and bottom and half on the left and right. For example, if Margin equals 4, there is a two pixel clear margin along the top, right, bottom, and left sides of the cell.

The property’s write method ensures that Margin is zero or positive; no exception is generated however.

See also: Adjust, Paint

OnOwnerDraw event

```
property OnOwnerDraw : TCellPaintNotifyEvent

TCellPaintNotifyEvent = procedure (
  Sender : TObject; TableCanvas : TCanvas;
  const CellRect : TRect; RowNum : TRowNum;
  ColNum : TColNum; const CellAttr : TOvcCellAttributes;
  Data : pointer; var DoneIt : Boolean) of object;

TOvcCellAttributes = record
  caAccess      : TOvcTblAccess;   {access rights}
  caAdjust      : TOvcTblAdjust;   {data adjustment}
  caColor       : TColor;          {background color}
  caFont        : TFont;           {text font}
  caFontColor   : TColor;          {text color}
  caFontHiColor : TColor;          {text highlight color}
  caTextStyle   : TOvcTextStyle;   {text style}
end;

TOvcTextStyle = (tsFlat, tsRaised, tsLowered);
```

Defines an event handler that is called when the cell needs repainting.

If a method is assigned to the OnOwnerDraw event then it will be called every time the cell needs repainting. This gives you the opportunity to paint the cell in any fashion that you wish. When OnOwnerDraw is called no painting in the cell has been done and either OnOwnerDraw must paint the whole cell rectangle or it must defer the painting to the standard paint method.

Sender is the cell component object that needs painting. TableCanvas is the canvas of the owning table and CellRect is the rectangle in that canvas that needs painting. The method must only paint inside this rectangle. (Note that you could write to the entire canvas; in other words, the rectangle is not part of any clipping region.) RowNum and ColNum form the address of the cell that needs to be painted.

CellAttr is a fully resolved set of attributes for the current cell. The attributes are the access rights, adjustment value, background color, font, font color, font highlight color for 3D styles and text style. The table collates this information through a prior call to ResolveAttributes.

Data is the address of the data that needs painting inside the cell and each cell component will use a different format for this data (see each individual cell component for details on the structure of their data).

Caution: the value of Data can be nil, and your OnOwnerDraw method must take this into account.

DoneIt is a Boolean that you have to set to state whether you painted the cell (set DoneIt to True) or not (set DoneIt to False). If, on return from calling your OnOwnerDraw method, DoneIt is True then the cell component does no more painting and returns to the table's paint routine so that the grid lines can be drawn. If, on the other hand, DoneIt is False on return from your OnOwnerDraw event, the cell component will paint itself in its usual fashion.

Note that the cell is painted before the table's grid lines are drawn, so if you want to draw your own grid lines for each cell (something the Orpheus table does not support) you will have to set the table's grid lines to "none" and then paint your special grid lines in your OnOwnerDraw event, cell by cell.

See also: Paint, ResolveAttributes

OnXxx event

```
property OnClick     : TNotifyEvent
property OnDblClick  : TNotifyEvent
property OnDragDrop  : TDragDropEvent
property OnDragOver  : TDragDropEvent
property OnEndDrag   : TEndDragEvent
property OnEnter     : TNotifyEvent
property OnExit      : TNotifyEvent
property OnKeyDown   : TKeyEvent
property OnKeyPress  : TKeyPressEvent
property OnKeyUp     : TKeyEvent
property OnMouseDown : TMouseEvent
property OnMouseMove : TMouseMoveEvent
property OnMouseUp   : TMouseEvent
```

↳ Events for the cell component's editing control.

These OnXxx events apply to the editing control when it is created, not to the cell component itself (the cell component has no visual representation at run time). The StartEditing method will, once it has created the editing control or enabled it, set its event handlers equal to the cell component's event handlers so that the editing control can trigger the required events.

The events have the same meanings as for the normal VCL components (e.g. OnMouseDown is triggered when a mouse button is pressed while the cursor is over the editing control).

Paint method

```
procedure Paint(
  TableCanvas : TCanvas; const CellRect : TRect;
  RowNum : TRowNum; ColNum : TColNum;
  const CellAttr : TOvcCellAttributes; Data : pointer);
```

```
TOvcCellAttributes = record
  caAccess      : TOvcTblAccess;   {access rights}
  caAdjust      : TOvcTblAdjust;   {data adjustment}
  caColor       : TColor;          {background color}
  caFont        : TFont;           {text font}
  caFontColor   : TColor;          {text color}
  caFontHiColor : TColor;          {text highlight color}
  caTextStyle   : TOvcTextStyle;   {text style}
end;
```

↳ The cell in the table.

This method paints the cell data. Firstly the OnOwnerDraw method is checked. If it is present then it is called so that any owner drawing of the cell can take place. If it is not present, or it is present but returns False to say that no painting took place, then an internal virtual method called tcPaint is called to perform the actual painting of the cell.

When you are creating your own cell component class descendants you will override the internal tcPaint virtual method to perform the painting for your cell type. We recommend that you peruse the source code for the standard Orpheus cell components to see how tcPaint has been implemented for each class The subsequent sections will also describe the painting method used by each cell component.

For details on the passed parameters to Paint please see OnOwnerDraw.

See also: OnOwnerDraw

References run-time, read-only property

```
property References : Integer
```

↳ The number of links between a cell and its table.

References is a read-only property. Its value is changed by IncRefs and DecRefs.

See also: DecRefs, IncRefs

ResolveAttributes virtual method

```
procedure ResolveAttributes(RowNum : TRowNum;
  ColNum : TColNum; var CellAttr : TOvcCellAttributes);

TOvcCellAttributes = record
  caAccess      : TOvcTblAccess;  {access rights}
  caAdjust      : TOvcTblAdjust;  {data adjustment}
  caColor       : TColor;         {background color}
  caFont        : TFont;          {text font}
  caFontColor   : TColor;         {text color}
  caFontHiColor : TColor;         {text highlight color}
  caTextStyle   : TOvcTextStyle;  {text style}
end;
```

↳ Calculates the attributes for a cell.

ResolveAttributes is called by the table when the cell needs painting or when it is about to be edited. The attributes are the access rights, adjustment value, background color, font, font color, font highlight color for 3D styles and text style.

ResolveAttributes first sets up the CellAttr record with the cell object's internal values and then calls the table's ResolveCellAttributes method. ResolveCellAttributes changes the relevant attributes for cases such as locked cells (which have a different color), when the cell object is deferring to the table for its access rights, or when there is an OnGetCellAttributes event handler set up for the table.

ResolveAttributes then makes sure that all fields are set up. If the access value is still otxDefault, it is reset to otxNormal. If the adjustment is still otaDefault, it is reset to otaCenterLeft. If the color is still clOvcDefaultTableColor (2FFFFFF), it is reset to clBtnFace.

SaveEditedData abstract method

```
procedure SaveEditedData(Data : pointer);
```

↳ Saves the data being edited.

When you create a descendant of a cell component, this method must be overridden. SaveEditedData saves the data that is currently being edited by the cell editor to the address provided by Data. It can be (and should be) assumed that at this point the data is valid (in other words, CanSaveEditedData will have been called and will have returned True).

See also: CanSaveEditedData, StopEditing

```
procedure SendKeyToTable(var Msg : TWMKey);
```

Sends a keystroke message to the table for processing.

The editing control for a cell object calls this method to send a WM_KEYDOWN message to the table. The editing control should have already called FilterTableKey for the message and received either an otkMustHave reply (the editing control must pass the message to the table for processing) or an otkWouldLike reply (the editing control can decide whether to pass the message on or not, but the table could use it).

For example, assume that the edit control is a normal Windows edit control. If the caret is at the start of the string, pressing the left arrow could result in the active cell being changed to the cell immediately to the left, but anywhere else in the string would result in the caret moving position one character to the left (the normal processing for an edit control). The call to FilterTableKey returns otkWouldLike and the edit control must determine where the caret is and call SendKeyToTable if it is at the start of the string.

The following example is extracted from the WM_KEYDOWN message handler for the editing control for a TOvcTCString cell. The method first sets up some defaults and calls the owning cell's FilterTableKey method (the owning cell is a field called CellOwner) to find out what importance the table places on the keystroke. If the table wants the keystroke (the return value is otkMustHave) then the editing control passes the keystroke right back to the table by calling SendKeyToTable. If the table would like the keystroke, the editing control makes a decision based on the keystroke. An example is given for <Left>: if the caret is at the start of the edited text and there is no selection, the table gets the keystroke (provided the correct property has been set). At the end of the routine, if the table did not use the keystroke, it is passed on to the editing control's window procedure by use of the inherited statement.

```
begin
  GridUsedIt := false;
  GridReply := otkDontCare;
  if (CellOwner <> nil) then
    GridReply := CellOwner.FilterTableKey(Msg);
  case GridReply of
    otkMustHave :
      begin
        CellOwner.SendKeyToTable(Msg);
        GridUsedIt := True;
      end;
    otkWouldLike :
      case Msg.CharCode of
        VK_LEFT :
            if TOvcTCCustomString(CellOwner).AutoAdvanceLeftRight
            then begin
            GetSelection(SStart, SEnd);
            if (SStart = SEnd) and (SStart = 0) then begin
              CellOwner.SendKeyToTable(Msg);
              GridUsedIt := True;
            end;
          end;
        ...other case selections...
      end;{case}
  end;{case}
  if not GridUsedIt then
    inherited;
end;
```

See also: FilterTableKey

30

StartEditing abstract method

```
procedure StartEditing(RowNum : TRowNum;
  ColNum : TColNum; CellRect : TRect;
  const CellAttr : TOvcCellAttributes;
  CellStyle : TOvcTblEditorStyle; Data : Pointer);
  virtual; abstract;
```

```
TOvcTblEditorStyle = (tesNormal, tesBorder, tes3D);
```

↳ Creates the editing control for the cell.

When you create a descendant of a cell component, this method must be overridden.

StartEditing should create (or show) the editing control for the cell. RowNum is the row number for the cell and ColNum is its column number. CellRect is the rectangle of the cell editor itself with respect to the table. Data is a pointer to the data.

CellAttr is a fully resolved set of attributes for the current cell. The attributes are access rights, adjustment value, background color, font, font color, font highlight color for 3D styles, and text style. The table collates this information through a prior call to ResolveAttributes.

CellStyle is a suggested style for the cell editor. The editor can ignore the suggested style if it does not support the style (e.g., the combo box cell does not support a border).

StartEditing should create a control that can edit the data. Specific properties for the control can be set as desired. The control's Parent should be the table itself, and CellRect should define the control's position and size. StartEditing could save the row and column numbers in fields of the object if they are required during the editing process.

See also: ResolveAttributes, StopEditing

StopEditing abstract method

```
procedure StopEditing(
  SaveValue : Boolean; Data : Pointer); virtual; abstract;
```

↳ Saves the data and destroys the editing control for the cell.

When you create a descendant of a cell component, this method must be overridden.

If SaveValue is True, StopEditing should save the edited data to the address specified by Data. After the data is saved, StopEditing must destroy (or hide) the editing control. CanSaveEditedData is called before StopEditing. If CanSaveEditedData returns False, StopEditing is not called. If CanSaveEditedData returns True, StopEditing is called and must do its job since you cannot back out of the editing mode at that point.

See also: CanSaveEditedData, StartEditing

Table property

```
property Table : TOvcTableAncestor
```

The owning table.

If you use Table to change the table that owns the cell, the current table is notified that this cell is changing allegiance to another table. This causes all current links to the cell to be removed. Then the internal table pointer is set to the passed table, however the reference count remains at zero. Ideally, you should use this property only to set the table to nil.

TableColor property

```
property TableColor : Boolean
```

Default: True

Indicates whether the background color of the cell is the same as the color of the table.

If you set TableColor to True, this instance's Color property is changed to match the Color property of the table. If you set TableColor to False, the Color property is not changed.

If the Color property for this instance is changed, TableColor is set to False.

If the Color property of the table is changed and TableColor is True, the Color property for this instance is changed to reflect the new Color value for the table. This happens only if the cell object is linked to a table.

Linking a cell to a table (changing its Table property) also causes the Color property for this instance to be modified if TableColor is True.

See also: Color, Table, TableFont

TableFont property

```
property TableFont : Boolean
```

Default: True

Indicates whether the font for the cell is the same as the font for the table.

If you set TableFont to True, this instance's Font property is changed to match the Font property of the table. If you set TableFont to False, the Font property is not changed.

If the Font property for this instance is changed, TableFont is set to False.

If the Font property of the table is changed and TableFont is True, the Font property for this instance is changed to reflect the new font value for the table. This happens only if the cell object is linked to a table.

Linking a cell to a table (by changing its Table property) also causes the Font property for this instance to be modified if TableFont is True.

See also: Color, Table, TableColor

TableWantsEnter method

```
function TableWantsEnter : Boolean;
```

Determines whether the owning table requires <Enter> keystrokes.

This method is called by the editing control to find out if the table requires <Enter> keystrokes to implement its "Enter to Arrow" feature. TableWantsEnter returns True if the table requires the <Enter> keystrokes.

See also: TableWantsTab, TOvcTable.Options

TableWantsTab method

```
function TableWantsTab : Boolean;
```

Determines whether the owning table requires tab characters.

This method is called by the editing control to find out if the table requires tab characters to implement its "Tab to Arrow" feature. TableWantsTab returns True if the table requires the tab characters.

See also: TableWantsEnter, TOvcTable.Options

TextHiColor property

```
property TextHiColor : TColor
```

Default: clBtnHighlight

↳ The highlight color for raised or lowered text.

See also: TextStyle

TextStyle property

```
property TextStyle : TOvcTextStyle
```

```
TOvcTextStyle = (tsFlat, tsRaised, tsLowered);
```

Default: tsFlat

↳ The appearance of the text drawn by the cell component.

Text drawn in a cell can appear in three different styles. tsFlat is the normal appearance (it looks “flat” on the screen), tsRaised makes the text appear embossed in a raised fashion, tsLowered makes the text appear embossed in a lowered fashion. For the 3D styles the highlighted color is TextHiColor.

See also: TextHiColor

TOvcTCBaseBitMap Class

The TOvcTCBaseBitMap class is the ancestor for all cell classes that display bitmaps.

The only change from the TOvcBaseTableCell is the internal method for painting the cell. The data pointer passed to the painting method refers to a record structure containing the bitmap to be displayed, the index of the glyph to display, a count of the number of glyphs (square subdivisions of the bitmap), and a count of the number of active glyphs that can be displayed. Descendants must initialize this data record and call the painting method for this class to do the actual painting of the glyph (if there is only one glyph, this equates to the whole bitmap). For example, the checkbox cell component uses a bitmap with three glyphs (unchecked, checked and grayed), and sometimes only two of these glyphs are used.

This class is designed as an abstract class, so you should not create any instances of it.

Paint/Edit Data Structure

The Data parameter passed to the base bitmap cell class for painting or editing purposes is a pointer to a TCellBitMapInfo record:

```
PCellBitMapInfo = ^TCellBitMapInfo;
TCellBitMapInfo = record
  BM          : TBitMap;    {bitmap object to display}
  Count       : Integer;    {number of glyphs}
  ActiveCount : Integer;    {number of active glyphs}
  Index       : Integer;    {index of glyph to display}
end;
```

BM is the bitmap to display. The bitmap is assumed to be a set of square glyphs (or sub-bitmaps) side by side. Count is the number of glyphs that are present in the bitmap (if Count is one then the bitmap is a single entity). ActiveCount is the number of glyphs that will be cycled through on editing, Index is the zero-based number of the glyph to display.

For example, the checkbox cell component is derived from this class. It uses a bitmap of 3 glyphs (Count is 3), and only uses 2 of those if its AllowGrayed property is false (Active Count is 2). Setting Index to 0 displays the first glyph (the unchecked glyph); setting Index to 1 displays the second glyph (the checked glyph).

When painting the component blanks the cell rectangle and then paints the relevant glyph on the canvas in the position indicated by CellAttr.Adjust. The glyph is not stretched to fit in the cell. The TCanvas.BrushCopy method is used to paint the glyph. This means that the glyphs can use the transparent color feature used by VCL's TSpeedbuttons if the bottom left pixel of the glyph is the transparent color. For more details on this feature, please see the VCL documentation.

Hierarchy

TComponent (VCL)

    TOvcTableCellAncestor (OvcTCmmn) 921

        ❶TOvcBaseTableCell (OvcTCell). 922

            TOvcTCBaseBitMap (OvcTCBmp)

TOvcTCCustomGlyph Class

The TOvcTCCustomGlyph class is the immediate ancestor of the TOvcTCGlyph component. It implements all the methods and properties used by the TOvcTCGlyph component and is identical to TOvcTCGlyph except that no properties are published.

TOvcTCCustomGlyph is provided to facilitate creation of descendent glyph cell components. For property and method descriptions, see “TOvcTCGlyph Component” on page 948.

Hierarchy

TComponent (VCL)

    TOvcTableCellAncestor (OvcTCmmn) 921

        ❶TOvcBaseTableCell (OvcTCell). 922

            TOvcTCBaseBitMap (OvcTCBmp). 945

                TOvcTCCustomGlyph (OvcTCGly)

30

TOvcTCGlyph Component

The TOvcTCGlyph component defines a cell type that displays one of a set of glyphs in a bitmap and, in edit mode, cycles through those glyphs. Glyphs are equal-sized subdivisions of a bitmap.

The class is designed to be more general than the checkbox cell component. The checkbox is a more restricted class in that it has three states and 2 or 3 of them can be cycled through in edit mode. TOvcTCGlyph, on the other hand, can display any number of glyphs and cycle through all them in edit mode using the spacebar.

The glyphs that are used by the component are stored in a TOvcCellGlyphs class (see page 893).

Paint/Edit Data Structure

The Data parameter transferred to a TOvcTCGlyph cell component for the painting and editing methods is a pointer to an integer. The value of this integer is the zero-based index into the set of glyphs in the bitmap.

The component does not do any painting itself, instead it creates a TCellBitMapInfo structure (see TOvcTCBaseBitMap: the BM field is the CellGlyphs.Bitmap property, Count is set to CellGlyphs.Count and ActiveCount to CellGlyphs.ActiveCount, and Index is set the Data integer) and then calls the ancestor class’ painting method with the address of this structure.

Hierarchy

TComponent (VCL)

TOvcTableCellAncestor (OvdTCmmn) 921

❶TOvcBaseTableCell (OvcTCell) 922

TOvcTCBaseBitMap (OvcTCBmp) 945

TOvcTCCustomGlyph (OvcTCGly) 947

TOvcTCGlyph (OvcTCGly)

Properties

❶ About	CellGlyphs	❶ Table
❶ AcceptActivationClick	❶ Color	❶ TableColor
❶ Access	❶ Font	❶ TableFont
❶ Adjust	❶ Margin	❶ TextHiColor
❶ CellEditor	❶ References	❶ TextStyle

Methods

CanAssignGlyphs	❶ EditMove	StartEditing
❶ CanSaveEditedData	❶ FilterTableKey	StopEditing
❶ CanStopEditing	❶ IncRefs	❶ TableWantsEnter
❶ DecRefs	❶ Paint	❶ TableWantsTab
❶ DoOwnerDraw	❶ ResolveAttributes	❶ Updated
❶ EditHandle	SaveEditedData	
❶ EditHide	❶ SendKeyToTable	

Events

❶ OnOwnerDraw	❶ OnXxx

Reference Section

CanAssignGlyphs — virtual method

```
function CanAssignGlyphs(CBG : TOvcCellGlyphs) : Boolean;
```

↳ Determines whether the instance can accept a glyph resource.

This method is defined for the checkbox cell component (TOvcTCCheckBox, a descendant of this class), which can accept a bitmap only if it has three glyphs. By default the return value from CanAssignGlyphs is True. The CellGlyphs property write routine calls this method before assigning the glyph resource. If CanAssignGlyphs returns false, the resource is not assigned to the instance.

See also: CellGlyphs

CellGlyphs — property

```
property CellGlyphs : TOvcCellGlyphs
```

Default: the checkbox glyph resource

↳ The glyph resource for the cell.

This property defines the glyphs for the instance. It is of type TOvcCellGlyphs which is defined in "TOvcCellGlyphs Class" on page 893. The resource defines the bitmap, the number of glyphs in the bitmap, and the number of active glyphs in the bitmap; all which are required for painting the cell. The glyphs are painted onto the table with the BrushCopy method of the TCanvas class, so you can specify a "transparent" color as the bottom left pixel of each glyph. When the glyph is painted, all pixels of this color in the glyph are changed to the background color of the canvas on which the glyph is drawn. See the compiler's documentation for more details.

When assigning to the property, the write routine calls the CanAssignGlyphs method to determine whether the glyph resource can be accepted. If CanAssignGlyphs returns False, the assignment does not take place.

See also: CanAssignGlyphs

SaveEditedData — virtual method

```
procedure SaveEditedData(Data : pointer);
```

↳ Saves the data being edited.

SaveEditedData saves the index of the edited glyph back to the Data pointer.

See also: StopEditing

StartEditing virtual method

```
procedure StartEditing(RowNum : TRowNum; ColNum : TColNum;
  CellRect : TRect; const CellAttr : TOvcCellAttributes;
  Data : Pointer);
```

Creates an editor for the glyph cell.

The editor that is created lets the user cycle through the active glyphs in the resource using the left mouse button or the spacebar. <Backspace> can be used to cycle through the glyphs in reverse order.

RowNum is the row number for the cell and ColNum is its column number. CellRect is the rectangle of the cell with respect to the table. CellAttr is a fully resolved set of attributes for the current cell.

For a description of the Data parameter, see the introduction to this section.

See also: Paint

StopEditing virtual method

```
procedure StopEditing(SaveValue : Boolean; Data : Pointer);
```

Saves the glyph index and destroys the editing control.

If SaveValue is True, StopEditing saves the index of the edited glyph back to the Data pointer.

For a description of the Data parameter, see the introduction to this section.

See also: StartEditing

TOvcTCCustomBitmap Class

The TOvcTCCustomBitmap class is the immediate ancestor of the TOvcTCBitmap component. It implements all the methods and properties used by the TOvcTCBitmap component and is identical to TOvcTCBitmap except that no properties are published.

TOvcTCCustomBitmap is provided to facilitate creation of descendent bitmap cell components. For property and method descriptions, see “TOvcTCBitmap Component” on page 953.

Hierarchy

TComponent (VCL)

    TOvcTableCellAncestor (OvdTCmmn) 921

        ❶TOvcBaseTableCell (OvcTCell) 922

            TOvcTCBaseBitMap (OvcTCBmp) 945

                TOvcTCCustomBitmap (OvcTCBmp)

TOvcTCBitmap Component

The TOvcTCBitmap component encapsulates a bitmap inside a cell. These bitmap cells are not editable; they are for display purposes only.

The data that is passed to the bitmap cell in the Paint method is a TBitmap instance. You must create these instances prior to displaying a table containing bitmaps.

Paint/Edit Data Structure

The Data parameter passed to a TOvcTCBitmap cell component is a TBitmap. To be more precise, the value in the Data parameter is a TBitmap object variable typecast as a pointer. In your table's OnGetCellData event handler, instead of using the @ address operator to return the value for Data, use a typecast to a pointer instead:

```
  Data := pointer(MyBitmapInstance);
```

In an OnOwnerDraw method for example you would typecast it as follows:

```
  var
    MyBitMap : TBitMap absolute Data;
```

The component does not do any painting itself, instead it creates a TCellBitMapInfo structure (see TOvcTCBaseBitMap: the BM field is the bitmap, both Count and ActiveCount are 1 and Index is 0) and then calls the ancestor class' painting method with the address of this structure.

Hierarchy

TComponent (VCL)

    TOvcTableCellAncestor (OvdTCmmn). 921

        ❶TOvcBaseTableCell (OvcTCell). 922

            TOvcTCBaseBitMap (OvcTCBmp). 945

                TOvcTCCustomBitmap (OvcTCBmp) 952

                    TOvcTCBitMap (OvcTCBmp)

30

TOvcTCCustomIcon Class

The TOvcTCCustomIcon class is the immediate ancestor of the TOvcTCIcon component. It implements all the methods and properties used by the TOvcTCIcon component and is identical to TOvcTCIcon except that no properties are published.

TOvcTCCustomIcon is provided to facilitate creation of descendent icon cell components. For property and method descriptions, see “TOvcTCIcon Component” on page 955.

Hierarchy

TComponent (VCL)

    TOvcTableCellAncestor (OvdTCmmn) 921

        ❶TOvcBaseTableCell (OvcTCell) 922

            TOvcTCCustomIcon (OvcTCIco)

TOvcTCIcon Component

The TOvcTCIcon component encapsulates an icon inside a cell. These icon cells are not editable; they are for display purposes only. No new methods or properties are documented in this class.

Paint/Edit Data Structure

The Data parameter passed to a TOvcTCIcon cell component is a TIcon. To be more precise, the value in the Data parameter is a TIcon object variable typecast as a pointer. In your table's OnGetCellData event handler, instead of using the @ address operator to return the value for Data, use a typecast to a pointer instead:

```
Data := pointer(MyIconInstance);
```

In an OnOwnerDraw method (for example) you would typecast it as follows:

```
var
  MyIcon : TIcon absolute Data;
```

The component paints the icon in the cell.

Hierarchy

TComponent (VCL)

  TOvcTableCellAncestor (OvdTCmmn) 921

    ❶TOvcBaseTableCell (OvcTCell) 922

      TOvcTCCustomIcon (OvcTCIco) 954

        TOvcTCIcon (OvcTCIco)

TOvcTCCustomCheckBox Class

The TOvcTCCustomCheckBox class is the immediate ancestor of the TOvcTCCheckBox component. It implements all the methods and properties used by the TOvcTCCheckBox component and is identical to TOvcTCCheckBox except that no properties are published.

TOvcTCCustomCheckBox is provided to facilitate creation of descendent checkbox cell components. For property and method descriptions, see “TOvcTCCheckBox Component” on page 957.

Hierarchy

TComponent (VCL)

    TOvcTableCellAncestor (OvdTCmmn) 921
        ❶TOvcBaseTableCell (OvcTCell) 922
            TOvcTCBaseBitMap (OvcTCBmp) 945
                TOvcTCCustomGlyph (OvcTCGly) 947
                    TOvcTCCustomCheckBox (OvcTCBox)

30

TOvcTCCheckBox Component

The TOvcTCCheckBox component defines a cell type for checkboxes. Like a normal checkbox, you can define whether the checkbox accepts the grayed state or not. Otherwise, the states for the cell are limited to unchecked or checked.

Paint/Edit Data Structure

The Data parameter passed to the TOvcTCCheckBox cell component for the painting and editing methods is a pointer to a variable of type TCheckBoxState. See the VCL documentation for details on this type.

The cell component does not do any painting itself, it generates an integer value from the passed data and passes the address of that onto the ancestor (TOvcTCGlyph). For the conversion to an integer: cbUnchecked is converted to 0, cbChecked is converted to 1, and cbGrayed is converted to 2.

Hierarchy

```
TComponent
  TOvcTableCellAncestor (OvdTCmmn) .................... 921
    ❶TOvcBaseTableCell (OvcTCell) ..................... 922
      TOvcTCBaseBitMap (OvcTCBmp) ..................... 945
        TOvcTCCustomGlyph (OvcTCGly) .................. 947
          ❷TOvcTCGlyph ................................ 948
            TOvcTCCustomCheckBox (OvcTCBox) ........... 956
              TOvcTCCheckBox (OvcTCBox)
```

30

Properties

- ❶ About
- ❶ AcceptActivationClick
- ❶ Access
- ❶ Adjust
- AllowGrayed
- ❶ CellEditor
- ❷ CellGlyphs
- ❶ Color
- ❶ Font
- ❶ Margin
- ❶ References
- ❶ Table
- ❶ TableColor
- ❶ TableFont
- ❶ TextHiColor
- ❶ TextStyle

Methods

- ❷ CanAssignGlyphs
- ❶ CanSaveEditedData
- ❶ CanStopEditing
- ❶ DecRefs
- ❶ DoOwnerDraw
- ❶ EditHandle
- ❶ EditHide
- ❶ EditMove
- ❶ FilterTableKey
- ❶ IncRefs
- ❶ Paint
- ❶ ResolveAttributes
- SaveEditedData
- ❶ SendKeyToTable
- StartEditing
- StopEditing
- ❶ TableWantsEnter
- ❶ TableWantsTab
- ❶ Updated

Events

- ❶ OnOwnerDraw
- ❶ OnXxx

Reference Section

AllowGrayed property

```
property AllowGrayed : Boolean
```

Default: False

↳ Indicates whether the grayed checkbox state can be used.

If you set AllowGrayed to True, CellGlyphs.ActiveCount is forced to 3. If you set AllowGrayed to False, CellGlyphs.ActiveCount is forced to 2.

Since AllowGrayed drives the value of CellGlyphs.ActiveCount, the class' methods do not allow you to alter the value of CellGlyphs.ActiveCount.

CanAssignGlyphs virtual method

```
function CanAssignGlyphs(CBG : TOvcCellGlyphs) : Boolean;
```

↳ Determines whether the instance can accept a glyph resource.

Since a checkbox has one of three states, this routine returns True only if the number of glyphs in the glyph resource (CBG.Count) is 3. For any other value of CBG.Count, CanAssignGlyphs returns False.

Therefore, if you want to use a different set of glyphs for a checkbox cell you will have to create a bitmap with exactly three glyphs. The first glyph must be for the unchecked state, the second for the checked state, and the third for the grayed state.

SaveEditedData method

```
procedure SaveEditedData(Data : pointer); override;
```

↳ Saves the data being edited.

SaveEditedData saves the check box value back to the Data pointer. The ancestor (which is in control of the editing session) uses an integer value as its index, so the SaveEditedData method must convert this integral value back into a TCheckBoxState value and save it in the original Data pointer. The value 0 is converted to cbUnChecked, 1 to cbChecked, and 2 to cbGrayed.

For a description of the Data parameter, see the introduction to this section.

See also: StopEditing

StartEditing virtual method

```
procedure StartEditing(RowNum : TRowNum; ColNum : TColNum;
  CellRect : TRect; const CellAttr : TOvcCellAttributes;
  Data : Pointer);
```

Creates an editor for the checkbox cell.

The method lets the ancestor's StartEditing method do most of the work. Before calling the inherited method, StartEditing must first convert Data.

For a description of the Data parameter, see the introduction of this section.

See also: Paint, StopEditing

StopEditing virtual method

```
procedure StopEditing(SaveValue : Boolean; Data : Pointer);
```

Saves the checkbox data and destroys the editing control.

If SaveValue is True, this method saves the checkbox data. The ancestor (which has been in control of the editing session) uses an integer value as its index, so the StopEditing method must convert this integral value back into a TCheckBoxState value and save it in the original Data pointer. The value 0 is converted to cbUnChecked, 1 to cbChecked, and 2 to cbGrayed.

See also: StartEditing

TOvcTCBaseString Class

The TOvcTCBaseString class is an abstract class that defines the main functions of a cell component that displays strings. It supports both length-byte strings and null-terminated strings.

TOvcTCBaseString essentially consists of a painting method for painting strings in a cell. The painting algorithm can paint the strings either word wrapped or not, and in all the nine possible adjustment positions. It is assumed that all string-type cell components will descend from this class and call its paint method to do the actual painting.

This class is designed as an abstract class, so you should not create any instances of it.

Paint/Edit Data Structure

The Data parameter passed to the TOvcTCBaseString cell component for the painting and editing methods is a pointer to a string. If the UseASCIIZStrings property is True, this string is a null terminated string. If the UseASCIIZStrings property is False, this string is a short string. The TOvcTCBaseString cell component does not support the VCL's long strings.

Hierarchy

TComponent (VCL)

  TOvcTableCellAncestor (OvdTCmmn) 921

    ❶ TOvcBaseTableCell (OvcTCell) 922

      TOvcTCBaseString (OvcTCStr)

Properties

- ❶ About
- ❶ AcceptActivationClick
- ❶ Access
- ❶ Adjust
- ❶ CellEditor
- ❶ Color
- ❶ Font
- ❶ Margin
- ❶ References
- ❶ Table
- ❶ TableColor
- ❶ TableFont
- ❶ TextHiColor
- ❶ TextStyle
- UseASCIIZStrings
- UseWordWrap

Methods

- ❶ CanSaveEditedData
- ❶ CanStopEditing
- ❶ DecRefs
- ❶ DoOwnerDraw
- ❶ EditHandle
- ❶ EditHide
- ❶ EditMove
- ❶ FilterTableKey
- ❶ IncRefs
- ❶ Paint
- ❶ ResolveAttributes
- ❶ SaveEditedData
- ❶ SendKeyToTable
- ❶ StartEditing
- ❶ StopEditing
- ❶ TableWantsEnter
- ❶ TableWantsTab
- ❶ Updated

Events

- OnChange
- ❶ OnOwnerDraw
- ❶ OnXxx

Reference Section

OnChange — event

```
property OnChange : TNotifyEvent
```

↳ Is triggered when the data in the string editing control changes.

The OnChange event applies to the editing control when it is created, not to the string cell component itself (the cell component has no visual representation at run time). This event has the same meaning as that for TEdit component. The StartEditing method will set the newly created editing control's OnChange event handler equal to this one.

TNotifyEvent is defined in the VCL's Classes unit.

UseASCIIZStrings — property

```
property UseASCIIZStrings : Boolean
```

Default: False

↳ Determines whether the cell uses null-terminated or length-byte strings.

If UseASCIIZStrings is True, the painting, StartEditing, and StopEditing methods all expect their Data parameter to point to a null-terminated string. If it is False, they expect a length-byte string.

See also: UseWordWrap

UseWordWrap — property

```
property UseWordWrap : Boolean
```

Default: False

↳ Determines whether text wraps within the cell.

If UseWordWrap is True, the painting method will use word wrap for the string passed to it. The painting method uses a call to the Windows API DrawText routine, which can only left adjust, right adjust, or center adjust the text. If the Adjust property is otaTopLeft, otaCenterLeft, or otaBottomLeft, the left adjustment method is used. If it is otaTopCenter, otaCenter, or otaBottomCenter, the center adjustment method is used. If it is otaTopRight, otaCenterRight, or otaBottomRight, the right adjustment method is used.

See also: UseASCIIZStrings

TO32TCFlexEdit Component

The TO32TCFlexEdit cell component is the FlexEdit component wrapped up for use in the OvcTable. It has most of the benefits of the O32FlexEdit component except it can be used in the OvcTable.

Hierarchy

TComponent (VCL)

  TOvcTableCellAncestor (OvdTCmmn) 921

    ❶ TOvcBaseTableCell (OvcTCell) 922

      ❷ TOvcTCBaseString(OvcTCStr) 961

        TO32TCFlexEdit (O32TCFlx)

Properties

- ❶ About
- ❶ AcceptActivationClick
- ❶ Access
- ❶ Adjust
- ❶ CellEditor
- ❶ Color
- ❶ Font
- ❶ Margin
- MaxLength
- ❶ References
- ❶ Table
- ❶ TableColor
- ❶ TableFont
- ❶ TextHiColor
- ❶ TextStyle
- ❷ UseASCIIZStrings
- ❷ UseWordWrap
- WantReturns
- WantTabs
- WordWrap

Methods

- ❶ CanSaveEditedData
- ❶ CanStopEditing
- CreateEditControl
- ❶ DecRefs
- ❶ DoOwnerDraw
- EditorBorders
- ❶ EditHandle
- ❶ EditHide
- ❶ EditMove
- EditorOptions
- ❶ FilterTableKey
- ❶ IncRefs
- ❶ Paint
- ❶ ResolveAttributes
- SaveEditedData
- ❶ SendKeyToTable
- StartEditing
- StopEditing
- ❶ TableWantsEnter
- ❶ TableWantsTab
- ❶ Updated

Events

- ❷ OnChange
- ❶ OnOwnerDraw
- ❶ OnXxx

Reference Section

CreateEditControl virtual method

```
function CreateEditControl(
  AOwner : TComponent) : TO32TCFledEditEditor; virtual;
```

↳ Creates an editing control based on TO32FlexEdit.

This method is virtual so that you can override the creation of the editing control, usually so that you can replace the default control with a control instantiated from a descendent type. CreateEditControl creates a FLex Edit editing control and returns it as the function result. It is called from the StartEditing method.

For more information on the TO32TCFledEditEditor type, see the source code in the O32TCFlx unit.

See also: StartEditing

EditorBorders property

```
property EditorBorders : Integer
```

↳ An implementation of the TO32Borders class.

The EditorBorders property allows you to enable or disable each of the four border sides. The borders can also be displayed in six different styles, Channel, Flat, Ridge, Raised, Lowered, or None. If a Flat border style is selected, then the border's color can be set using the FlatColor property. Otherwise, the border will use current system colors.

```
property EditorOptions : TO32TCEditorProperties

O32TCEditorProperties = class(TPersistent)
  protected
    FAlignment       : TAlignment;
    FBorders         : TO32Borders;
    FButtonGlyph     : TBitmap;
    FColor           : TColor;
    FCursor          : TCursor;
    FMaxLines        : Integer;
    FShowButton      : Boolean;
    FPasswordChar    : Char;
    FReadOnly        : Boolean;
  public
    constructor Create; virtual;
    destructor Destroy; override;

  property Borders: TO32Borders
    read FBorders write FBorders;

  published
    {$IFDEF VERSION4}
    property Alignment: TAlignment
      read FAlignment write FAlignment;
    {$ENDIF}
    property ButtonGlyph: TBitmap
      read FButtonGlyph write FButtonGlyph;
    property Color: TColor
      Read FColor write FColor;
    property Cursor: TCursor
      read FCursor write FCursor;
    property MaxLines: Integer
      read FMaxLines write FMaxLines;
    property PasswordChar: Char
      read FPasswordChar write FPasswordChar;
    property ReadOnly: Boolean
      Read FReadOnly write FReadOnly;
    property ShowButton: Boolean
      read FShowButton write FShowButton;
  end;
```

↳ An implementation of the TO32TCEditorProperties class. Its purpose is to surface the properties of the standard FlexEdit component in the O32TCFlexEdit.

Some of the properties that are available in the standard FlexEdit, like the MouseOverLines, are not available in the table cell version of the component because they don't make sense in the context of a table cell.

For information on the individual properties available in EditorOptions, see the TO32FlexEdit component's documentation on page 439.

MaxLength property

```
property MaxLength : Integer
```

Default: 0

↳ The maximum number of characters that can be edited in the cell.

The value of MaxLength is used to set the MaxLength property of the editing control when it is created in the StartEditing method. It does not, however, limit the number of characters that can be displayed in the cell.

Warning: Unlike the TEdit MaxLength property, the value 0 is not allowed and you must specify a positive value. For a TEdit control, a MaxLength value of 0 implies no limit to the number of characters that can be edited. The TO32TCFlexEdit component requires you to explicitly state the number of characters. For this reason, the default value has been overridden to 255. If you need to save more than 255 characters in your underlying data structure then you must change this number. The TO32TCFlexEdit control will save up to MaxLength+1 bytes, that is the string characters plus the length byte or the null terminator, when it saves data. The Data pointer that is passed to the StopEditing method must point to a variable that is at least MaxLength+1 bytes in size. If it is smaller, a memory overwrite will occur.

See also: StartEditing

SaveEditedData method

```
procedure SaveEditedData(Data : pointer);
```

↳ Saves the string data being edited.

See also: StopEditing

StartEditing method

```
procedure StartEditing(
  RowNum : TRowNum; ColNum : TColNum; CellRect : TRect;
  const CellAttr : TOvcCellAttributes;
  Data : Pointer); override;
```

Creates the editing control for the cell.

StartEditing calls CreateEditControl to create the editing control and sets the properties so that the control can be displayed in the table at the correct position.

RowNum is the row number of the cell being edited and ColNum is its column number. CellRect is the rectangle of the cell with respect to the displayed table. CellAttr is a fully resolved set of attributes for the cell.

Data is a pointer to a string. If UseASCIIZStrings is True, the string is a null-terminated string. If UseASCIIZStrings is False, the string is a length-byte string.

See also: MaxLength, StopEditing, TOvcTCBaseString.StartEditing,
TOvcTCBaseString.UseASCIIZStrings

StopEditing virtual method

```
procedure StopEditing(
  SaveValue : Boolean; Data : Pointer); virtual;
```

Saves the string and destroys the editing control.

If SaveValue is True, StopEditing saves the edited string back to the Data pointer.

StopEditing writes up to MaxLength+1 bytes to the Data pointer and you must ensure that the memory block is large enough. The string memory block pointed to by data must be at least MaxLength+1 bytes long (the string itself will have a length less than or equal to MaxLength). If the memory block is any smaller, the StopEditing method will cause a memory overwrite.

See also: MaxLength, StartEditing, TOvcTCBaseString.StopEditing

WantReturns property

```
property WantReturns : Integer
```

Defines whether the FlexEdit will accept returns into the edit control or pass them on to the table.

See also: TO32FlexEdit.WantReturns.

WantTabs property

```
property WantTabs : Integer
```

↳ Defines whether the FlexEdit will accept tabs into the edit control or pass them on to the Table.

See also: TO32FlexEdit.WantTabs

WordWrap property

```
property WordWrap : Integer
```

↳ Defines whether the FlexEdit will automatically wrap the text entered in to the editor.

See also:TO32FlexEdit.WordWrap

TOvcTCCustomString Class

The TOvcTCCustomString class is the immediate ancestor of the TOvcTCString component. It implements all the methods and properties used by the TOvcTCString component and is identical to TOvcTCString except that no properties are published.

TOvcTCCustomString is provided to facilitate creation of descendent string cell components. For property and method descriptions, see “TOvcTCString Component” on page 972.

Hierarchy

TComponent (VCL)

  TOvcTableCellAncestor (OvdTCmmn) 921

    TOvcBaseTableCell (OvcTCell) 922

      TOvcTCBaseString (OvcTCStr). 961

        TOvcTCCustomString (OvcTCEdt)

30

TOvcTCString Component

The TOvcTCString cell component is the simplest cell component that displays and edits strings. It uses an editing control descended from the TEdit control, so the control appears as a single line of text. If the string is longer than the cell's width, the edit control will scroll the text horizontally.

Paint/Edit Data Structure

The Data parameter passed to a TOvcTCString component is the same as that for TOvcTCBaseString.

Hierarchy

TComponent (VCL)

    TOvcTableCellAncestor (OvdTCmmn) 921
        ❶TOvcBaseTableCell (OvcTCell) 922
            ❷TOvcTCBaseString (OvcTCStr) 961
                TOvcTCCustomString (OvcTCEdt) 971
                    TOvcTCString (OvcTCEdt)

30

Properties

- ❶ About
- ❶ AcceptActivationClick
- ❶ Access
- ❶ Adjust
- AutoAdvanceChar
- AutoAdvanceLeftRight
- ❶ CellEditor
- ❶ Color
- ❶ Font
- ❶ Margin
- MaxLength
- ❶ References
- ❶ Table
- ❶ TableColor
- ❶ TableFont
- ❶ TextHiColor
- ❶ TextStyle
- ❷ UseASCIIZStrings
- ❷ UseWordWrap

Methods

- ❶ CanSaveEditedData
- ❶ CanStopEditing
- CreateEditControl
- ❶ DecRefs
- ❶ DoOwnerDraw
- ❶ EditHandle
- ❶ EditHide
- ❶ EditMove
- ❶ FilterTableKey
- ❶ IncRefs
- ❶ Paint
- ❶ ResolveAttributes
- SaveEditedData
- ❶ SendKeyToTable
- StartEditing
- StopEditing
- ❶ TableWantsEnter
- ❶ TableWantsTab
- ❶ Updated

Events

- ❷ OnChange
- ❶ OnOwnerDraw
- ❶ OnXxx

Reference Section

AutoAdvanceChar property

```
property AutoAdvanceChar : Boolean
```

Default: False

Determines what happens if the caret is at the end of the editing control and you enter another character.

If AutoAdvanceChar is True, the editing control sends (via SendKeyToTable) a WM_KEYDOWN message to the table containing the VK_RIGHT virtual keystroke. The table stops editing the current cell and moves the active cell right one column.

If AutoAdvanceChar is False, the caret remains where it is.

See also: AutoAdvanceLeftRight

AutoAdvanceLeftRight property

```
property AutoAdvanceLeftRight : Boolean
```

Default: False

Determines what happens if the caret is at the end of the editing control and you enter a caret movement command.

There are two cases to consider when AutoAdvanceLeftRight is True.

First is when the caret is at the beginning of the field and the user presses the left arrow key. In this case the editing control sends (via SendKeyToTable) a WM_KEYDOWN message to the table containing the VK_LEFT virtual keystroke. The effect of this will be for the table to stop editing the current cell and move the active cell left one column.

Second is when the caret is at the end of the field and the user presses the right arrow key. A similar action occurs with the effect that the table stops editing the current cell and moves the active cell right one column.

If AutoAdvanceLeftRight is False in either of these two situations, the caret remains where it is.

See also: AutoAdvanceChar

CreateEditControl virtual method

```
function
  CreateEditControl(AOwner : TComponent) : TOvcTCStringEdit;
```

Creates an editing control based on TEdit.

This method is virtual so that you can override the creation of the editing control, usually so that you can replace the default control with a control instantiated from a descendent type. CreateEditControl creates an editing control and returns it as the function result. It is called from the StartEditing method.

For more information on the TOvcTCStringEdit type, see the source code for the OvcTCEdt unit.

See also: StartEditing

MaxLength property

```
property MaxLength : Integer
```

Default: 0

The maximum number of characters that can be edited in the cell.

The value of MaxLength is used to set the MaxLength property of the editing control when it is created in the StartEditing method. It does not, however, limit the number of characters that can be displayed in the cell.

Warning: Unlike the TEdit MaxLength property, the value 0 is not allowed and you must specify a positive value. For a TEdit control, a MaxLength value of 0 implies no limit to the number of characters that can be edited. The TOvcTCString component requires you to explicitly state the number of characters. For this reason, the default value is not acceptable and you must change it. If you do not change it, no characters will be saved by the StopEditing method (it saves up to MaxLength+1 bytes, that is the string characters plus the length byte or the null terminator).

The Data pointer that is passed to the StopEditing method must point to a variable that is at least MaxLength+1 bytes in size. If it is smaller, a memory overwrite will occur.

See also: StartEditing

SaveEditedData method

```
procedure SaveEditedData(Data : pointer);
```

Saves the string data being edited.

See also: StopEditing

StartEditing method

```
procedure StartEditing(
  RowNum : TRowNum; ColNum : TColNum; CellRect : TRect;
  const CellAttr : TOvcCellAttributes;
  Data : Pointer); override;
```

↳ Creates the editing control for the cell.

StartEditing calls CreateEditControl to create the editing control and sets the properties so that the control can be displayed in the table at the correct position.

RowNum is the row number of the cell being edited and ColNum is its column number. CellRect is the rectangle of the cell with respect to the displayed table. CellAttr is a fully resolved set of attributes for the cell.

Data is a pointer to a string. If UseASCIIZStrings is True, the string is a null-terminated string. If UseASCIIZStrings is False, the string is a length-byte string.

See also: MaxLength, StopEditing, TOvcTCBaseString.StartEditing,
TOvcTCBaseString.UseASCIIZStrings

StopEditing virtual method

```
procedure StopEditing(
  SaveValue : Boolean; Data : Pointer);
```

↳ Saves the string and destroys the editing control.

If SaveValue is True, StopEditing saves the edited string back to the Data pointer.

StopEditing writes up to MaxLength+1 bytes to the Data pointer and you must ensure that the memory block is large enough. The string memory block pointed to by data must be at least MaxLength+1 bytes long (the string itself will have a length less than or equal to MaxLength). If the memory block is any smaller, the StopEditing method will cause a memory overwrite.

See also: MaxLength, StartEditing, TOvcTCBaseString.StopEditing

TOvcTCColHead Component

The TOvcTCColHead component is a simple descendant of the TOvcTCBaseString class, expressly designed for column headings. The component does not allow editing.

The component has several states. In the first it ignores all data passed to it and instead displays spreadsheet-like letters for the cell. The first column after the locked columns has the heading ‘A’, the next is ‘B’, and so on until ‘Z’. Then the sequence starts with ‘AA’ and continues until ‘AZ’, followed by ‘BA’ on to ‘BZ’, and so on.

In the second state it will display either the string for the current column from its Headings stringlist property or it will accept the data passed to it.

The LockedRowsCell property of the TOvcTable component (see page 1023) can be set to a component of this type, causing the columns in the table to be easily identified.

Paint/Edit Data Structure

The Data parameter passed to a TOvcTCColHead component is the same as that for TOvcTCBaseString.

If the ShowLetters property is True the cell component displays spreadsheet style letters and ignores the Data parameter completely (it generates an internal string representing the column as one or more letters and calls the ancestor’s painting method to paint it). If ColNum is greater than or equal to the number of locked columns in the table, the method generates a string for that column number according to this table:

Column number range	**String**
`LockedCols..LockedCols+25`	`'A'..'Z'`
`LockedCols+26..LockedCols+51`	`'AA'..'AZ'`
`LockedCols+52..LockedCols+77`	`'BA'..'BZ'`
`LockedCols+78..LockedCols+103`	`'CA'..'CZ'`
`...and so on, so on, so forth ...`	

If the ShowLetters property is False the painting method looks at the Data pointer. If this is non-nil, the inherited painting method is called passing on the Data pointer. If the Data pointer is nil, it will get the string to display from the Headings string list using the passed ColNum value as index. If there is a string at this index in the Headings list the inherited painting method is called with its address; otherwise the inherited method is called with the address of an empty string instead.

Hierarchy

TComponent (VCL)

- TOvcTableCellAncestor (OvdTCmmn) 921
  - ❶TOvcBaseTableCell (OvcTCell) 922
    - ❷TOvcTCBaseString (OvcTCStr) 961
      - TOvcTCColHead (OvcTCHdr)

Properties

❶	About		Headings	❶	TableFont
❶	AcceptActivationClick	❶	Margin	❶	TextHiColor
❶	Access	❶	References	❶	TextStyle
❶	Adjust		ShowActiveCol	❷	UseASCIIZStrings
❶	CellEditor		ShowLetters	❷	UseWordWrap
❶	Color	❶	Table		
❶	Font	❶	TableColor		

Methods

❶	CanSaveEditedData	❶	EditMove	❶	StartEditing
❶	CanStopEditing	❶	FilterTableKey	❶	StopEditing
	Create	❶	IncRefs	❶	TableWantsEnter
❶	DecRefs	❶	Paint	❶	TableWantsTab
❶	DoOwnerDraw	❶	ResolveAttributes	❶	Updated
❶	EditHandle	❶	SaveEditedData		
❶	EditHide	❶	SendKeyToTable		

Events

❷	OnChange	❶	OnOwnerDraw	❶	OnXxx

Reference Section

Create constructor

```
constructor Create(AOwner : TComponent);
```

Makes an instance of a column heading cell component.

This constructor forces the Access property to otxReadOnly, UseASCIIZStrings to False, and UseWordWrap to False.

Headings property

```
property Headings : TStringList
```

Is a list of column heading strings.

If the ShowLetters property is False and the Data pointer passed to the painting method is nil, the cell component will use the relevant string in this string list for display purposes. The first string in the list (element 0) will be used for column 0, element 1 for column 1 and so forth.

This property enables you to display column headings at design time. Use the string list editor to enter a set of strings, one line for each column, and set the ShowLetters property off.

See also: ShowLetters

ShowActiveCol property

```
property ShowActiveCol : Boolean
```

Indicates whether an arrow should be displayed for the active column.

If ShowActiveCol is True, a down arrow is displayed for the active column. If it is False, the normal column heading (be it letter or string) is displayed for the active column.

ShowLetters property

```
property ShowLetters : Boolean
```

Default: True

↳ Determines whether spreadsheet style letters are displayed.

If ShowLetters is True the painting method will display spreadsheet style letters for the columns (i.e. ‘A’ will be displayed for the first unlocked column, the next will be ‘B’ and so on). If ShowLetters is False the painting method will display a string, either from the Headings string list property or the passed Data pointer.

TOvcTCRowHead Component

The TOvcTCRowHead component is a simple descendant of the TOvcTCString class, expressly designed for row headings. The component ignores all data passed to it and instead displays row numbers for the cell. The cell component does not allow editing.

The first row after the locked rows has the heading ‘1’, the next is ‘2’, and so on.

The DefaultCell property of the first locked column can be set to an instance of this component, causing the first column to easily show the numbers of the rows.

Paint/Edit Data Structure

The TOvcTCRowHead cell component ignores all data passed to it via the Data parameter.

If the row number is greater than or equal to the number of locked rows in the table, the painting method generates a string for that row by subtracting the number of locked rows from the passed row number, adding 1, and then converting to a string. The inherited painting method is then called to paint the string. This algorithm means that the first row after the locked rows has the heading ‘1’, the next is ‘2’ and so on.

Hierarchy

TComponent (VCL)

  TOvcTableCellAncestor (OvdTCmmn) 921

    ❶TOvcBaseTableCell (OvcTCell) 922

      ❷TOvcTCBaseString (OvcTCStr) 961

        TOvcTCRowHead (OvcTCHdr)

Properties

❶	About	❶	Font	❶	TextHiColor
❶	AcceptActivationClick	❶	Margin	❶	TextStyle
❶	Access	❶	References	❷	UseASCIIZStrings
❶	Adjust	❶	Table	❷	UseWordWrap
❶	CellEditor	❶	TableColor		
❶	Color	❶	TableFont		

Methods

❶	CanSaveEditedData	❶	EditMove		ShowActiveRow
❶	CanStopEditing	❶	FilterTableKey		ShowNumbers
	Create	❶	IncRefs	❶	StartEditing
❶	DecRefs	❶	Paint	❶	StopEditing
❶	DoOwnerDraw	❶	ResolveAttributes	❶	TableWantsEnter
❶	EditHandle	❶	SaveEditedData	❶	TableWantsTab
❶	EditHide	❶	SendKeyToTable	❶	Updated

Events

❷	OnChange	❶	OnOwnerDraw	❶	OnXxx

Reference Section

Create constructor

```
constructor Create(AOwner : TComponent);
```

Makes an instance of a row heading cell component.

This constructor forces the Access property to otxReadOnly, UseASCIIZStrings to false, and UseWordWrap to False.

ShowActiveRow property

```
property ShowActiveRow : Boolean
```

Default: False

Indicates whether an arrow should be displayed for the active row.

If ShowActiveRow is True, a right arrow is displayed for the active row. If it is false, the row number is displayed for the active row.

ShowNumbers property

```
property ShowNumbers : Boolean
```

Default: True

Determines whether row numbers are displayed.

If ShowNumbers is True then the row number passed to the painting method is displayed in the cell. If False, the row number is not displayed. Use this property in conjunction with the ShowActiveRow property to have a locked column that just displays an arrow for the active row (i.e. ShowActiveRow is True, ShowNumbers is False).

See also: ShowActiveRow

TOvcTCCustomMemo Class

The TOvcTCCustomMemo class is the immediate ancestor of the TOvcTCMemo component. It implements all the methods and properties used by the TOvcTCMemo component and is identical to TOvcTCMemo except that no properties are published.

TOvcTCCustomMemo is provided to facilitate creation of descendent memo cell components. For property and method descriptions, see "TOvcTCMemo Component" on page 985.

Hierarchy

TComponment (VCL)

  TOvcTableCellAncestor (OvdTCmmn) 921

    ❶TOvcBaseTableCell (OvcTCell) 922

      ❷TOvcTCBaseString (OvcTCStr) 961

        TOvcTCCustomMemo (OvcTCEdt)

30

TOvcTCMemo Component

The TOvcTCMemo component is a cell component that enables the user to edit a memo in a cell.

Although descended from a TOvcTCBaseString class, this class makes two very important restrictions: the UseWordWrap property must be set to True and the UseASCIIZStrings property must be set to True. The Create constructor ensures that these two properties are set True. The memo cell component must display its data (a null-terminated string) in a word wrapped format.

The memo editor that is created fits exactly into the cell boundaries. This means that the space is extremely cramped and the text in the memo control must be scrolled to see it all. However, since space is at a premium, the memo editor is created without scrollbars, so you can only scroll the text by use of the keyboard.

Paint/Edit Data Structure

The Data parameter that is passed to the painting and StartEditing methods is a pointer to a null-terminated string. The class has no support for the alternative TMemo data representations (the length byte string and the TStringList).

Hierarchy

TComponent (VCL)

  TOvcTableCellAncestor (OvdTCmmn) 921

    ❶TOvcBaseTableCell (OvcTCell) 922

      ❷TOvcTCBaseString (OvcTCStr) 961

        TOvcTCCustomMemo (OvcTCEdt) 984

          TOvcTCMemo (OvcTCEdt)

Properties

- ❶ About
- ❶ AcceptActivationClick
- ❶ Access
- ❶ Adjust
- ❶ CellEditor
- ❶ Color
- ❶ Font
- ❶ Margin
- MaxLength
- ❶ References
- ❶ Table
- ❶ TableColor
- ❶ TableFont
- ❶ TextHiColor
- ❶ TextStyle
- ❷ UseASCIIZStrings
- ❷ UseWordWrap

Methods

- ❶ CanSaveEditedData
- ❶ CanStopEditing
- Create
- CreateEditControl
- ❶ DecRefs
- ❶ DoOwnerDraw
- ❶ EditHandle
- ❶ EditHide
- ❶ EditMove
- ❶ FilterTableKey
- ❶ IncRefs
- ❶ Paint
- ❶ ResolveAttributes
- SaveEditedData
- ❶ SendKeyToTable
- StartEditing
- StopEditing
- ❶ TableWantsEnter
- ❶ TableWantsTab
- WantReturns
- WantTabs
- ❶ Updated

Events

- ❷ OnChange
- ❶ OnOwnerDraw
- ❶ OnXxx

Reference Section

Create — constructor

```
constructor Create(AOwner : TComponent);
```

An instance of memo cell component.

This constructor creates the instance of the cell component and forces the UseASCIIZStrings and UseWordWrap properties to True.

CreateEditControl — virtual method

```
function
  CreateEditControl(AOwner : TComponent) : TOvcTCMemoEdit;
```

Creates an editing control based on TMemo.

This method is virtual so that you can override the creation of the memo editing control, usually so that you can replace the default control with a control instantiated from a descendent type. CreateEditControl creates an editing control based on TMemo and returns it as the function result. It is called from the StartEditing method.

For more information on the TOvcTCMemoEdit type, see the source code.

See also: StartEditing

MaxLength — property

```
property MaxLength : Word
```

Default: 0

The maximum number of characters that can be edited in the cell.

The value of MaxLength is used to set the MaxLength property of the memo editing control when it is created in the StartEditing method. It does not, however, limit the number of characters that can be displayed in the cell.

Unlike the TMemo MaxLength property, the value 0 is not allowed and you must specify a positive value. For a TMemo control, a MaxLength value of 0 implies no limit to the number of characters that can be edited. The TOvcTCMemo component requires you to explicitly state the number of characters. For this reason, the default value is not acceptable and you must change it.

The Data pointer that is passed to the StopEditing method must point to a variable that is at least MaxLength+1 bytes in size, or a memory overwrite will occur.

See also: StartEditing

SaveEditedData virtual method

```
procedure SaveEditedData(Data : pointer);
```

Saves the memo data being edited.

See also: StopEditing

StartEditing virtual method

```
procedure StartEditing(
  RowNum : TRowNum; ColNum : TColNum; CellRect : TRect;
  const CellAttr : TOvcCellAttributes; Data : Pointer);
```

Creates the memo editing control for the cell.

StartEditing calls CreateEditControl to create the memo editing control and sets the properties so that the control can be displayed in the table at the correct position.

RowNum is the row number of the cell being edited and ColNum is its column number. CellRect is the rectangle of the cell with respect to the displayed table. CellAttr is a fully resolved set of attributes for the cell.

Data is a pointer to a null terminated string.

See also: MaxLength, StopEditing, TOvcTCBaseString.StartEditing

StopEditing virtual method

```
procedure StopEditing(SaveValue : Boolean; Data : Pointer);
```

Saves the memo and destroys the editing control.

If SaveValue is True, StopEditing saves the edited memo string back to the Data pointer.

StopEditing writes up to MaxLength+1 bytes to the Data pointer and you must ensure that the memory block is large enough. The string memory block being pointed to must be at least MaxLength+1 bytes long (the string itself will have a length of MaxLength characters or less); if the memory block is any less, the StopEditing method will cause a memory overwrite.

See also: StartEditing, TOvcTCBaseString.StopEditing

WantReturns property

```
property WantReturns : Boolean
```

Default: False

↳ Determines whether carriage returns are inserted in the memo text when <Enter> is pressed.

This property is used to set up the memo editing control after StartEditing creates it. If WantReturns is True and <Enter> is pressed, a carriage return is inserted in the memo text. If WantReturns is False and <Enter> is pressed, the default button for the form is selected. A carriage return can still be inserted into the text by pressing <CtrlEnter>. This property corresponds to the TMemo WantReturns property.

See also: WantTabs

WantTabs property

```
property WantTabs : Boolean
```

Default: False

↳ Determines whether tab characters are inserted in the memo text.

This property is used to set up the memo editing control after StartEditing creates it. If WantTabs is True and <Tab> is pressed, a tab character is inserted in the memo text. If WantTabs is False and <Tab> is pressed, the focus is advanced to the next selectable control on the form. This property corresponds to the TMemo WantTabs property.

See also: WantReturns

TOvcTCCustomComboBox Class

The TOvcTCCustomComboBox class is the immediate ancestor of the TOvcTCComboBox component. It implements all the methods and properties used by the TOvcTCComboBox component and is identical to TOvcTCComboBox except that no properties are published.

TOvcTCCustomComboBox is provided to facilitate creation of descendent icon cell components.

Hierarchy

TComponent (VCL)

    TOvcTableCellAncestor (OvdTCmmn) 921

        ❶TOvcBaseTableCell (OvcTCell) 922

            ❷TOvcTCBaseString (OvcTCStr) 961

                TOvcTCCustomComboBox (OvcTCCBx)

TOvcTCComboBox Component

The TOvcTCComboBox component encapsulates a combo box inside a cell. The three styles of combo box that are catered for are the drop-down, drop-down list and simple styles.

The data that is passed to the combo box cell in the painting, StartEditing and StopEditing methods depends on the style of the combo box. For a drop-down list combo box, the data simply consists of an integer: the index into an internal string list. The string list is set up beforehand (for example in the form designer) or can be initialized on the fly at run time. The combo box cell will display the string at the given index in the string list.

For the other two types of combo box (drop-down and simple), there is an edit field associated with the control: the user can either select an entry in the list or enter a new string in the edit field. The data that is passed to the cell takes account of this: it is a record whose first field is an integer and the second a string.

You can use the TOvcTCComboBox with yet another mode of operation. Normally the strings displayed by the combo box cell are defined at design time in the Items property. The combo box cell supports another mode in which the strings are supplied at run time. For a drop- down list, the painting method would require the index number and a string list containing the strings to display. For a drop-down combo box, you'd need to pass the index, the string list of items to display, and the string that could be used to override the values in the items list.

An inactive combo box cell either just displays the string, or you can get the cell to display a down facing arrow as well. A combo box cell that is active will either show a representation of the combo box button or the same downfacing arrow on the right-hand side of the cell and the string is displayed on the left. Like the check box cell, this button is “inert”, you have to either press <F2> or click on the cell (when it is active) to start the editing mode.

The standard Windows combo box (from which the editing control for the combo box cell is descended) has a fixed vertical size that is proportional to the font being used. It is impossible to change the height of a combo box control, in this case to resize it to fit inside a cell boundary, and so the TOvcTCComboBox does not attempt to. It resizes the width of the combo box to fit the cell, but vertically it just places the control at the top of the cell.

Paint/Edit Data Structure

The data passed to the painting and editing methods is a variant record of the form:

```
PCellComboBoxInfo = ^TCellComboBoxInfo;
  TCellComboBoxInfo = record
    Index : integer;               {index into Items list}
    case integer of
      0 : (St      : ShortString);  {string value if Index = -1}
      1 : (RTItems : TStrings;      {run-time items list}
           RTSt    : ShortString);  {run-time string value if
                                     Index = -1}
  end;
```

The data read from (or written to) a variable of this record type is dependent on the type of the combo box, whether a run-time item list is used instead of the design-time list, and whether the string displayed in the combo box is returned when data is saved.

If you use the drop-down list type, generally only the index field is required. The Data parameter is then just a pointer to an integer that is a zero-based index into the Items string list for the cell. When the cell is painted, the painting method gets the index from the passed Data parameter and displays that element from the Items property. After editing, the combo box cell returns only the index of the item selected. If you need the string itself, you can get it from the Items property of the cell component. If you need the string returned by the cell component on saving, you must set the SaveStringValue property and also make sure that your Data parameter points to a record variable that includes the string field St. If you need to pass a run-time string list for the items that should be displayed (rather than using the design-time Items property) then you must create the string list, populate it with strings, and then pass it to the painting and editing routines by making sure that Data points to a record that includes the RTItems field. If you need both a run-time item list and to return the string, you must use a record variable that includes both the RTItems and RTSt fields.

If you use the drop-down combo box type, there is one main difference: you must provide a string field in your record variable. The Index field can be the index of a string in the Items property, or it can be -1. When it is -1, the St field is the string that is displayed by the combo box cell. This behavior supports the edit control in the combo box cell. After editing, if the user selects a string that is in the list, the combo box cell returns the index of that string and the string itself. If the user enters a new string into the edit control part of the combo box, the combo box returns -1 for the Index field and the user's string in the St field. Of course, if the cell component is using a run-time item list, the string is always provided and returned in the RTSt field.

The combo box cell does not attempt to match up strings to items in the Items property.

Hierarchy

TComponent (VCL)

  TOvcTableCellAncestor (OvdTCmmn) 921

    ❶TOvcBaseTableCell (OvcTCell) 922

      ❷TOvcTCBaseString (OvcTCStr) 961

        TOvcTCCustomComboBox (OvcTCCBx) 990

          TOvcTCComboBox (OvcTCCBx)

Properties

- ❶ About
- ❶ AcceptActivationClick
- ❶ Access
- ❶ Adjust
- AutoAdvanceChar
- AutoAdvanceLeftRight
- ❶ CellEditor
- ❶ Color
- DropDownCount
- ❶ Font
- Items
- ❶ Margin
- MaxLength
- SaveStringValue
- ShowButton
- Sorted
- Style
- ❶ References
- ❶ Table
- ❶ TableColor
- ❶ TableFont
- ❶ TextHiColor
- ❶ TextStyle
- ❷ UseASCIIZStrings
- UseRunTimeItems
- ❷ UseWordWrap

Methods

- ❶ CanSaveEditedData
- ❶ CanStopEditing
- Create
- CreateEditControl
- ❶ DecRefs
- ❶ DoOwnerDraw
- ❶ EditHandle
- ❶ EditHide
- ❶ EditMove
- ❶ FilterTableKey
- ❶ IncRefs
- ❶ Paint
- ❶ ResolveAttributes
- SaveEditedData
- ❶ SendKeyToTable
- StartEditing
- StopEditing
- ❶ TableWantsEnter
- ❶ TableWantsTab
- WantReturns
- WantTabs
- ❶ Updated

Events

- ❷ OnChange
- OnDropDown
- OnDrawItem
- OnMeasureItem
- ❶ OnOwnerDraw
- ❶ OnXxx

Reference Section

AutoAdvanceChar property

```
property AutoAdvanceChar : Boolean
```

Default: False

↳ Determines what happens if the caret is at the end of the combo box's edit control and you enter another character.

The AutoAdvanceChar property is only used for combo box cell components that will use an edit control (i.e., not the drop-down list type).

If AutoAdvanceChar is True the combo box's edit control sends (via SendKeyToTable) a WM_KEYDOWN message to the table containing the VK_RIGHT virtual keystroke. The table to stops editing the current cell and moves the active cell right one column.

If AutoAdvanceChar is False, the caret remains where it is.

See also: AutoAdvanceLeftRight

AutoAdvanceLeftRight property

```
property AutoAdvanceLeftRight : Boolean
```

Default: False

↳ Determines what happens if the caret is at the end of the combo box' edit control and you enter a caret movement command.

If the combo box style is cbDropDownList, then if AutoAdvanceLeftRight is True pressing <Left> or <Right> will cause the table to move the active cell left or right. If it is false, pressing <Left> and <Right> will select the previous or next string in the string list.

If the combo box style is not cbDropDownList, there are two cases to consider when AutoAdvanceLeftRight is True. First is when the caret is at the beginning of the field and the user presses the left arrow key. In this case the combo box' edit control sends (via SendKeyToTable) a WM_KEYDOWN message to the table containing the VK_LEFT virtual keystroke. The effect of this will be for the table to stop editing the current cell and move the active cell left one column.

Second is when the caret is at the end of the field and the user presses the right arrow key. A similar action occurs with the effect that the table stops editing the current cell and moves the active cell right one column.

If AutoAdvanceLeftRight is False in either of these two situations, the caret remains where it is.

DropDownCount property

```
property DropDownCount : Integer
```

Default: 8

The number of items visible in the drop-down list.

This is the same property as the TComboBox component's DropDownCount property.

Items property

```
property Items : TStrings
```

The list of pre-set strings for the combo box.

This is the same property as the TComboBox component's Items property.

MaxLength property

```
property MaxLength : Word
```

Default: 0

The maximum length of the string that can be edited by the cell.

This is the same property as the TComboBox component's MaxLength property.

Note: The strings in the Items property can be longer than this (only the index is returned if one of these is selected), but any string that is entered in the edit control of the combo box is forced to be less than or equal to this length.

OnChange event

```
property OnChange : TNotifyEvent
```

efines an event handler that is called when the data in the combo box cell changes.

This is the same event as the TComboBox component's OnChange event.

TNotifyEvent is defined in the VCL's Classes unit.

OnDropDown event

```
property OnDropDown : TNotifyEvent
```

efines an event handler that is called when the combo box list is dropped down.

This is the same event as the TComboBox component's OnDropDown event.

OnDrawItem event

```
property OnDrawItem : TDrawItemEvent
```

efines an event handler that is called when an item in the combo box list requires drawing.

This is the same event as the TComboBox component's OnDrawItem event.

OnMeasureItem event

```
property OnMeasureItem : TMeasureItemEvent
```

Defines an event handler that is called when an item in the combo box list requires measuring.

This is the same event as the TComboBox component's OnMeasureItem event.

SaveEditedData method

```
procedure SaveEditedData(Data : pointer);
```

Saves the edited data.

For a drop-down list style combo box, the index of the selected item (the TComboBox's ItemIndex property) is saved.

For other styles, the index of the selected item is saved in the first field of the Data record. If the value of the ItemIndex property is -1 the Text property of the combo box is saved in the second field. If the value is not -1, the string at ItemIndex in the Items string list is saved in this second field (for convenience purposes). Either way only a maximum of MaxLength characters are transferred.

See also: MaxLength

SaveStringValue property

```
property SaveStringValue : Boolean
```

Default: False

↳ Indicates whether the string is saved along with the index into the Items list.

When editing is stopped for a combo box cell component and the user has made a selection in the list of strings, the index of the selected string is returned. Setting SaveStringValue to True causes the string to be returned also. If SaveStringValue is True, you must ensure that the Data

parameter on return from the table's OnGetCellData event points to a memory block large enough to receive the string.

If the combo box cell is not a drop-down list type and the user entered a new string into the edit control part of the combo box, the string is returned, regardless of the setting of this property.

See also: StopEditing

ShowButton property

```
property ShowButton : Boolean
```

Default: True

↳ Indicates whether a combo box style button is shown in the cell non-editing state.

If ShowButton is True, a fake combo box button is drawn in the cell when the cell is active, but not being edited. If False a downward pointing arrow is drawn instead without the 3D button effect. The latter option looks better, since the combo box button that is drawn with the former is the old Windows 3.x style.

Sorted property

```
property Sorted : Boolean
```

Default: True

↳ Indicates whether the pre-set strings are sorted.

This is the same property as the TComboBox component’s Sorted event.

```
procedure StartEditing(RowNum : TRowNum; ColNum : TColNum;
  CellRect : TRect; const CellAttr : TOvcCellAttributes;
  CellStyle : TOvcTblEditorStyle; Data : pointer); override;
```

```
TOvcCellAttributes = record
  caAccess      : TOvcTblAccess;   {access rights}
  caAdjust      : TOvcTblAdjust;   {data adjustment}
  caColor       : TColor;          {background color}
  caFont        : TFont;           {text font}
  caFontColor   : TColor;          {text color}
  caFontHiColor : TColor;          {text highlight color}
  caTextStyle   : TOvcTextStyle;   {text style}
end;
```

```
TOvcTblEditorStyle = (tesNormal, tesBorder, tes3D);
```

Sets up and shows the combo box editing control for the cell.

RowNum and ColNum are the row and column numbers of the cell being edited. CellRect is the rectangle of the cell with respect to the table. CellAttr is a fully resolved set of attributes for the current cell. CellStyle is a suggested style for the cell editor.

For a description of the Data parameter, see “Paint/Edit Data Structure” on page 992.

The cell component creates a combo box control to do the editing and places it over the cell rectangle, adjusting the width if necessary. The combo box control is placed flush against the top of the cell.

For a drop-down list style combo box, only the passed index is used to set the value of the editing control’s ItemIndex property. For other styles, if the index is in range for the Items property then it is used to set ItemIndex; if it is not the passed string is used to initialize the combo box control’s Text property.

See also: Paint, StopEditing

StopEditing method

```
procedure StopEditing(
  SaveValue : Boolean; Data : pointer); override;
```

Saves the edited data and frees the editing control.

If SaveValue is True, the data in the combo box control is saved to the Data pointer. If SaveValue is False, the data is discarded. For a description of the Data parameter, see the Paint method.

For a drop-down list style combo box, the index of the selected item (the TComboBox's ItemIndex property) is saved.

For other styles, the index of the selected item is saved in the first field of the Data record. If the value of the ItemIndex property is -1 the Text property of the combo box is saved in the second field. If the value is not -1, the string at ItemIndex in the Items string list is saved in this second field (for convenience purposes). Either way only a maximum of MaxLength characters are transferred.

Style property

```
property Style : TComboBoxStyle
```

Default: csDropDown

Indicates the type of combo box.

For details on the TComboBoxStyle, please see the documentation for the VCL's TComboBox component.

UseRunTimeItems property

```
property UseRunTimeItems : Boolean
```

Default: False

Defines whether a string list is supplied at run time for the combo box.

If UseRunTimeItems is False, the combo box cell component uses its Items property for the list of strings. If UseRunTimeItems is True, you must provide an initialized list as part of the memory block pointed to by the Data parameter. See "Paint/Edit Data Structure" on page 992 for details on the structure of the memory block.

TOvcTCBaseEntryField Class

The TOvcTCBaseEntryField class is the ancestor to the cell components that are based on the Orpheus simple, picture, and numeric edit fields. It is an abstract ancestor class, so you should not instantiate any objects from it.

In common with the TOvcBaseEntryField, this class defines all the core common properties and methods used by the actual cell components. These components (the TOvcTCSimpleField, TOvcTCPictureField and TOvcTCNumericField) are simple descendants from this class and really only define different editing controls used for editing the data.

Most of the properties defined by this class are the same as the properties defined in the TOvcBaseEntryField class. Therefore they are not documented again here. See “TOvcBaseEntryField Class” on page 363 for descriptions. These properties are not used by the cell components themselves, but they initialize the editing control when the user wants to edit a cell based on TOvcTCBaseEntryField.

The TOvcTCBaseEntryField class and descendants mark a departure from the cell classes defined so far. Components derived from this class automatically create two instances of the underlying editing control when they are created. The first of these two controls is used exclusively for generating the string to be displayed in the cell and the second is the actual editing control (it is hidden until the user wants to edit a cell, when it is moved into place and shown). The other cell components discussed so far create their editing controls only when cell data needs to be edited.

There are many reasons for this departure. One reason is that having the two editing controls around means that the cell class is easier to write. It does not have to store the properties for the fields because the editing controls do that, making the storing and loading of the cell component from the DFM file easier. A second reason is that generating the string that the cell component displays is much easier if there is an Orpheus entry field control around to do the hard work. The downside to this design is that more system resources are used. Two windowed controls exist for the life of the cell, compared with one windowed control that is created on the fly when a cell needs to be edited.

Paint/Edit Data Structure

The Data parameter passed to the painting and editing methods depends entirely on the data type required by the cell component. For example if the entry field cell is for double values then Data will be a pointer to a double variable; if for integer values then Data will be a pointer to an integer variable; and so on. For painting, the cell component will pass the

Data parameter onto the internal data entry field control and obtain the string representation of the data from that control. This string is then passed onto the ancestor's painting method to be drawn inside the cell.

Hierarchy

TComponent (VCL)

TOvcTableCellAncestor (OvdTCmmn). 921

❶TOvcBaseTableCell (OvcTCell). 922

❷TOvcTCBaseString (OvcTCStr). 961

TOvcTCBaseEntryField (OvcTCBEF)

Properties

❶ About	❶ CellEditor	❶ TableColor
❶ AcceptActivationClick	❶ Color	❶ TableFont
❶ Access	❶ Font	❶ TextHiColor
❶ Adjust	❶ Margin	TextMargin
AutoAdvanceChar	MaxLength	❶ TextStyle
AutoAdvanceLeftRight	❶ References	❷ UseASCIIZStrings
AutoSelect	❶ Table	❷ UseWordWrap

Methods

❶ CanSaveEditedData	❶ EditHide	❶ SendKeyToTable
❶ CanStopEditing	❶ EditMove	StartEditing
Create	❶ FilterTableKey	StopEditing
CreateEditControl	❶ IncRefs	❶ TableWantsEnter
❶ DecRefs	❶ Paint	❶ TableWantsTab
❶ DoOwnerDraw	❶ ResolveAttributes	❶ Updated
❶ EditHandle	SaveEditedData	

Events

❷ OnChange	❶ OnOwnerDraw	❶ OnXxx

Reference Section

AutoAdvanceChar property

```
property AutoAdvanceChar : Boolean
```

Default: False

↳ Determines what happens if the caret is at the end of the entry field and you enter another character.

If AutoAdvanceChar is True, the ActiveCell is advanced to the next cell on the table. If AutoAdvanceChar is False, the caret remains where it is.

See also: AutoAdvanceLeftRight

AutoAdvanceLeftRight property

```
property AutoAdvanceLeftRight : Boolean
```

Default: False

↳ Determines what happens if the caret is at the beginning or end of the entry field and a cursor movement command is issued.

The first case to consider is when the caret is at the beginning of the field and a command that would normally move the caret to the left is issued. If AutoAdvanceLeftRight is True, the ActiveCell is moved to the previous cell on the table. If AutoAdvanceLeftRight is false, the caret remains where it is.

The second case to consider is when the caret is at the end of a field and a command that would normally move the caret to the right is issued. If AutoAdvanceLeftRight is True, the ActiveCell is advanced to the next cell on the table. If AutoAdvanceLeftRight is false, the caret remains where it is.

See also: AutoAdvanceChar

AutoSelect property

```
property AutoSelect : Boolean
```

Default: True

↳ Determines whether the field is selected when a table cell enters edit mode.

If AutoSelect is True, the contents of the field are selected when the cell enters edit mode.

CanSaveEditedData method

```
function CanSaveEditedData(SaveValue : Boolean) : Boolean;
```

↳ Determines whether the edited data is valid and can be saved.

If SaveValue is True, CanSaveEditedData calls the ValidateSelf method of the editing control, returning the result of the call (i.e., if the data is valid CanSaveEditedData returns True, otherwise it returns False). If SaveValue is False, CanSaveEditedData calls the Restore method of the editing control to clear any pending validation and then returns True.

See also: SaveEditedData, StartEditing, StopEditing,
TOvcTCBaseTableCell.CanSaveEditedData

Create constructor

```
constructor Create(AOwner : TComponent);
```

↳ An instance of a TOvcTCBaseEntryField object.

In addition to creating the instance of the cell, the Create constructor creates two instances of the editing control, one for generating the display strings, and one for editing the cell data. The controls are created by two separate calls to CreateEntryField.

See also: CreateEntryField

CreateEntryField abstract method

```
function CreateEntryField(
  AOwner : TComponent) : TOvcBaseEntryField;
```

↳ Creates an editing control for the cell component.

Although the return value is declared as a TOvcBaseEntryField, CreateEntryField must create a descendant of it instead. This is because the descendant must follow the editing control guidelines (i.e., it must pass all WM_KEYDOWN messages on to the table). The descendants of TOvcTCBaseEntryField override this method to create an editing control. For example, the TOvcTCSimpleField component overrides CreateEntryField to create an object of type TOvcTCSimpleFieldEdit (a descendant of TOvcSimpleField, which is a descendant of TOvcBaseEntryField). The TOvcTCSimpleFieldEdit class has a method that calls its owning cell's FilterTableKey method for every WM_KEYDOWN message.

See also: Create

SaveEditedData method

```
procedure SaveEditedData(Data : pointer);
```

↳ Saves the edited data.

SaveEditedData gets the edited data by calling the GetValue method of the editing control and copying the result to the Data pointer. Note that the data must be valid by this stage (i.e., CanSaveEditedData must have been called and returned True).

See also: StopEditing

StartEditing method

```
procedure StartEditing(
  RowNum : TRowNum; ColNum : TColNum; CellRect : TRect;
  const CellAttr : TOvcCellAttributes; Data : Pointer);
```

↳ Sets up and shows the editing control for the cell.

RowNum and ColNum are the row and column numbers of the cell being edited. CellRect is the rectangle of the cell with respect to the displayed table. CellAttr is a fully resolved set of attributes for the cell.

For a description of the Data parameter, see the introduction to this section.

StartEditing sets the required properties of the edit control and then makes the editing control visible. The editing control's value is set by calling its SetValue method, passing the dereferenced Data pointer.

See also: CanSaveEditedData, Paint, StopEditing, TOvcTCBaseTableCell.StartEditing

StopEditing method

```
procedure StopEditing(SaveValue : Boolean; Data : Pointer);
```

↳ Saves the edited data and hides the editing control.

If SaveValue is True, StopEditing gets the edited data by calling the GetValue method of the editing control and copying the result to the Data pointer. If SaveValue is false, the edited value is discarded.

The StopEditing method then hides the editing control.

See also: CanSaveEditedData, StartEditing, TOvcTCBaseTableCell.StopEditing

TextMargin property

```
property TextMargin : Integer
```

Default: 2

Determines the field’s display indent when editing.

The TextMargin property controls the left indent in pixels for the simple and picture entry field cells. For the numeric entry field cell, this property is the right margin since the painting is now done from right to left. The minimum value for TextMargin is 2.

Note that this property is only used when the cell is in edit mode.

TOvcTCCustomSimpleField Class

The TOvcTCCustomSimpleField class is the immediate ancestor of the TOvcTCSimpleField component. It implements all the methods and properties used by the TOvcTCSimpleField component and is identical to TOvcTCSimpleField except that no properties are published.

TOvcTCCustomSimpleField is provided to facilitate creation of descendent simple field cell components. For property and method descriptions, see “TOvcTCSimpleField Component” on page 398.

Hierarchy

TComponent (VCL)

  TOvcTableCellAncestor (OvdTCmmn) 921

    ❶TOvcBaseTableCell (OvcTCell) 922

      ❷TOvcTCBaseString (OvcTCStr) 961

        TOvcTCBaseEntryField (OvcTCBEF) 1000

          TOvcTCCustomSimpleField (OvcTCSim)

TOvcTCSimpleField Component

The TOvcTCSimpleField cell component encapsulates an Orpheus simple field (TOvcSimpleField) inside a cell. The data that is displayed and edited by the simple field cell component can be any of the supported simple field data types. The component is not restricted to just strings. You have full control over the picture mask used for displaying and editing the data. For details on the properties and for information about picture masks for simple fields, see “TOvcSimpleField Component” on page 398.

Paint/Edit Data Structure

The Data parameter passed to the painting and editing methods is the same as that for TOvcTCBaseEntryField, and depends on the DataType property.

Hierarchy

TComponent (VCL)

TOvcTableCellAncestor (OvdTCmmn) 921

❶TOvcBaseTableCell (OvcTCell) 922

❷TOvcTCBaseString (OvcTCStr) 961

❸TOvcTCBaseEntryField (OvcTCBEF) 1000

TOvcTCCustomSimpleField (OvcTCSim) 1006

TOvcTCSimpleField (OvcTCSim)

30

Properties

❶	About	❶	Color	❶	TableColor
❶	AcceptActivationClick		DataType	❶	TableFont
❶	Access	❶	Font	❶	TextHiColor
❶	Adjust	❶	Margin	❸	TextMargin
❸	AutoAdvanceChar	❸	MaxLength	❶	TextStyle
❸	AutoAdvanceLeftRight		PictureMask	❷	UseASCIIZStrings
❸	AutoSelect	❶	References	❷	UseWordWrap
❶	CellEditor	❶	Table		

Methods

❶	CanSaveEditedData	❶	EditHandle	❸	SaveEditedData
❶	CanStopEditing	❶	EditHide	❶	SendKeyToTable
❸	Create	❶	EditMove	❸	StartEditing
❸	CreateEditControl	❶	FilterTableKey	❸	StopEditing
	CreateEntryField	❶	IncRefs	❶	TableWantsEnter
❶	DecRefs	❶	Paint	❶	TableWantsTab
❶	DoOwnerDraw	❶	ResolveAttributes	❶	Updated

Events

❷	OnChange	❶	OnOwnerDraw	❶	OnXxx

Reference Section

CreateEntryField method

```
function CreateEntryField(
  AOwner : TComponent): TOvcBaseEntryField;
```

Creates an editing control based on a simple field.

See also: TOvcTCBaseEntryField.CreateEntryField

DataType property

```
property DataType : TSimpleDataType
```

Default: sftString

Determines what type of information can be displayed and edited by the cell.

The painting, StartEditing, and StopEditing methods automatically do any necessary conversion from this type. However, to prevent memory overwrites, you must ensure that the memory block pointed to by the Data pointer is a variable of the required type.

The possible values and their corresponding data types are:

Value	**Delphi**	**C++Builder**
pftString	string	AnsiString
pftChar	AnsiChar	unsigned char
pftBoolean	Boolean	bool
pftYesNo	Boolean	bool
pftLongInt	LongInt	long or int
pftWord	Word	Word (unsigned short int)
pftInteger	SmallInt	short int
pftByte	Byte	Byte
pftShortInt	ShortInt	ShortInt (signed char)
pftReal	Real	double
pftExtended	Extended	Extended (long double)
pftDouble	Double	double
pftSingle	Single	Single (float)
pftComp	Comp	Comp (double)

PictureMask property

```
property PictureMask : AnsiChar
```

Default: pmAnyChar (‘X’)

Defines which characters the field accepts and the character format.

The editing control for the cell will edit the data using this picture mask. The picture mask is also used indirectly by the painting method when data is displayed in the cell.

See “TOvcSimpleField Component” on page 398 for information about picture masks for simple fields.

TOvcTCCustomPictureField Class

The TOvcTCCustomPictureField class is the immediate ancestor of the TOvcTCPictureField component. It implements all the methods and properties used by the TOvcTCPictureField component and is identical to TOvcTCPictureField except that no properties are published.

TOvcTCCustomPictureField is provided to facilitate creation of descendent picture field cell components. For property and method descriptions, see “TOvcTCPictureField Component” on page 1012.

Hierarchy

TComponent (VCL)

    TOvcTableCellAncestor (OvdTCmmn) 921

        TOvcBaseTableCell (OvcTCell) 922

            TOvcTCBaseString (OvcTCStr) 961

                TOvcTCBaseEntryField (OvcTCBEF) 1000

                    TOvcTCCustomPictureField (OvcTCPic)

30

TOvcTCPictureField Component

The TOvcTCPictureField cell component encapsulates an Orpheus picture field (TOvcPictureField) inside a cell. The data that is displayed and edited by the picture field cell component can be any of the supported picture field data types. The component is not restricted to just strings. You have full control over the picture mask used for displaying and editing the data.

For details on the properties and information about picture masks for picture fields, see “TOvcPictureField Component” on page 411.

Paint/Edit Data Structure

The Data parameter passed to the painting and editing methods is the same as that for TOvcTCBaseEntryField, and depends on the DataType property.

Hierarchy

TComponent (VCL)

TOvcTableCellAncestor (OvdTCmmn) 921

❶ TOvcBaseTableCell (OvcTCell) 922

❷ TOvcTCBaseString (OvcTCStr). 961

❸ TOvcTCBaseEntryField (OvcTCBEF) .1000

TOvcTCCustomPictureField (OvcTCPic) 1011

TOvcTCPictureField (OvcTCPic)

Properties

- ❶ About
- ❶ AcceptActivationClick
- ❶ Access
- ❶ Adjust
- ❸ AutoAdvanceChar
- ❸ AutoAdvanceLeftRight
- ❸ AutoSelect
- ❶ CellEditor
- ❶ Color
- DataType
- ❶ Font
- ❶ Margin
- ❸ MaxLength
- PictureMask
- ❶ References
- ❶ Table
- ❶ TableColor
- ❶ TableFont
- ❶ TextHiColor
- ❸ TextMargin
- ❶ TextStyle
- ❷ UseASCIIZStrings
- ❷ UseWordWrap

Methods

- ❶ CanSaveEditedData
- ❶ CanStopEditing
- ❸ Create
- ❸ CreateEditControl
- CreateEntryField
- ❶ DecRefs
- ❶ DoOwnerDraw
- ❶ EditHandle
- ❶ EditHide
- ❶ EditMove
- ❶ FilterTableKey
- ❶ IncRefs
- ❶ Paint
- ❶ ResolveAttributes
- ❸ SaveEditedData
- ❶ SendKeyToTable
- ❸ StartEditing
- ❸ StopEditing
- ❶ TableWantsEnter
- ❶ TableWantsTab
- ❶ Updated

Events

- ❷ OnChange
- ❶ OnOwnerDraw
- ❶ OnXxx

Reference Section

CreateEntryField — method

```
function CreateEntryField(
  AOwner : TComponent): TOvcBaseEntryField;
```

Creates an editing control based on a picture field.

See also: TOvcTCBaseEntryField.CreateEntryField

DataType — property

```
property DataType : TPictureDataType
```

Default: pftString

Determines what type of information can be displayed and edited by the cell.

The painting, StartEditing, and StopEditing methods automatically do any necessary conversion from this type. However, to prevent memory overwrites, you must ensure that the memory block pointed to by the Data pointer is a variable of the required type.

The possible values and their corresponding data types are:

Value	**Delphi**	**C++Builder**
pftString	string	AnsiString
pftChar	AnsiChar	unsigned char
pftBoolean	Boolean	bool
pftYesNo	Boolean	bool
pftLongInt	LongInt	long or int
pftWord	Word	Word (unsigned short int)
pftInteger	SmallInt	short int
pftByte	Byte	Byte
pftShortInt	ShortInt	ShortInt (signed char)
pftReal	Real	double
pftExtended	Extended	Extended (long double)
pftDouble	Double	double
pftSingle	Single	Single (float)

Value	Delphi	C++Builder
pftComp	Comp	Comp (double)
pftDate	TStDate	TStDate
pftTime	TStTime	TStTime

PictureMask property

```
property PictureMask : string
```

Default: ‘XXXXXXXXXXXXXXX’

Defines which characters the field accepts and the character format.

The editing control for the cell will edit the data using this picture mask. The picture mask is also used indirectly by the painting method when data is displayed in the cell. See “TOvcPictureField Component” on page 411 for information about picture masks.

TOvcTCCustomNumericField Class

The TOvcTCCustomNumericField class is the immediate ancestor of the TOvcTCNumericField component. It implements all the methods and properties used by the TOvcTCNumericField component and is identical to TOvcTCNumericField except that no properties are published.

TOvcTCCustomNumericField is provided to facilitate creation of descendent numeric field cell components. For property and method descriptions, see “TOvcTCNumericField Component” on page 1017.

Hierarchy

TComponent (VCL)

  TOvcTableCellAncestor (OvdTCmmn) 921

    TOvcBaseTableCell (OvcTCell). 922

      TOvcTCBaseString (OvcTCStr) 961

        TOvcTCBaseEntryField (OvcTCBEF) 1000

          TOvcTCCustomNumericField (OvcTCNum)

TOvcTCNumericField Component

The TOvcTCNumericField cell component encapsulates an Orpheus numeric field (TOvcNumericField) inside a cell. The data that is displayed and edited by the numeric field cell component can be any of the supported numeric field data types. You have full control over the picture mask used for displaying and editing the data.

For details on the properties and information about picture masks for numeric fields, see “TOvcNumericField Component” on page 424.

Paint/Edit Data Structure

The Data parameter passed to the painting and editing methods is the same as that for TOvcTCBaseEntryField, and depends on the DataType property.

Hierarchy

TComponent (VCL)

    TOvcTableCellAncestor (OvdTCmmn) 921

        ❶TOvcBaseTableCell (OvcTCell) 922

            ❷TOvcTCBaseString (OvcTCStr) 961

                ❸TOvcTCBaseEntryField (OvcTCBEF) .1000

                    TOvcTCCustomNumericField (OvcTCNum) 1016

                        TOvcTCNumericField (OvcTCNum)

Properties

- ❶ About
- ❶ AcceptActivationClick
- ❶ Access
- ❶ Adjust
- ❸ AutoAdvanceChar
- ❸ AutoAdvanceLeftRight
- ❸ AutoSelect
- ❶ CellEditor
- ❶ Color
- DataType
- ❶ Font
- ❶ Margin
- ❸ MaxLength
- PictureMask
- ❶ References
- ❶ Table
- ❶ TableColor
- ❶ TableFont
- ❶ TextHiColor
- ❸ TextMargin
- ❶ TextStyle
- ❷ UseASCIIZStrings
- ❷ UseWordWrap

Methods

- ❶ CanSaveEditedData
- ❶ CanStopEditing
- ❸ Create
- ❸ CreateEditControl
- CreateEntryField
- ❶ DecRefs
- ❶ DoOwnerDraw
- ❶ EditHandle
- ❶ EditHide
- ❶ EditMove
- ❶ FilterTableKey
- ❶ IncRefs
- ❶ Paint
- ❶ ResolveAttributes
- ❸ SaveEditedData
- ❶ SendKeyToTable
- ❸ StartEditing
- ❸ StopEditing
- ❶ TableWantsEnter
- ❶ TableWantsTab
- ❶ Updated

Events

- ❷ OnChange
- ❶ OnOwnerDraw
- ❶ OnXxx

Reference Section

CreateEntryField method

```
function
  CreateEntryField(AOwner : TComponent): TOvcBaseEntryField;
```

Creates an editing control based on a numeric field.

See also: TOvcTCBaseEntryField.CreateEntryField

DataType property

```
property DataType : TNumericDataType
```

Default: nftLongInt

Determines what type of information can be displayed and edited by the cell.

The painting, StartEditing, and StopEditing methods automatically do any necessary conversion from this type. However, you must ensure that the memory block pointed to by the Data pointer is a variable of the required type, otherwise memory overwrites can occur.

The possible values and their corresponding data types are:

Value	**Delphi**	**C++Builder**
pftLongInt	LongInt	long or int
pftWord	Word	Word (unsigned short int)
pftInteger	SmallInt	short int
pftByte	Byte	Byte
pftShortInt	ShortInt	ShortInt (signed char)
pftReal	Real	double
pftExtended	Extended	Extended (long double)
pftDouble	Double	double
pftSingle	Single	Single (float)
pftComp	Comp	Comp (double)

PictureMask property

```
property PictureMask : string
```

Default: “##########”

↳ Determines which characters the field accepts and the character format.

The editing control for the cell will edit the data using this picture mask. The picture mask is also used indirectly by the painting method when data is displayed in the cell.

See “TOvcNumericField Component” on page 424 for information about picture masks for numeric fields.

TOvcTableAncestor Class

The TOvcTableAncestor class is the True ancestor of the TOvcTable component. It exists to provide some low-level interaction between the cells used by the table and the table itself (especially when loading/storing to a DFM file and when changing the cells used by a table). This is a True abstract ancestor. You should not create any instances of this class, but use one of the descendants instead.

The class will not be described any further in this manual, for details please see the source code in OVCTCMMN.PAS.

Hierarchy

TCustomControl (VCL)

  TOvcCustomControl (OvcBase) 16

    TOvcCustomControlEx (OvcBase) 19

      TOvcTableAncestor (OvcTCmmn)

TOvcCustomTable Class

The TOvcCustomTable class is the immediate ancestor of the TOvcTable component. It implements all the methods and properties used by the TOvcTable component and is identical to TOvcTable except that no properties are published.

TOvcCustomTable is provided to facilitate creation of descendent table components. For property and method descriptions, see “TOvcTable Component” on page 1023.

Hierarchy

TCustomControl (VCL)

  TOvcCustomControl (OvcBase) 16

    TOvcCustomControlEx (OvcBase) 19

      TOvcTableAncestor (OvcTCmmn) 1021

        TOvcCustomTable (OvcTable)

TOvcTable Component

The TOvcTable component is the Orpheus table control. It is a grid-like component for displaying data that naturally falls into a row and column matrix (or tabular) format. The TOvcTable component is the controlling force behind the display and editing of data in the grid. It uses a plethora of internal objects to manage this process, the majority of which were described earlier in this chapter.

As a default, the table component assumes that rows are records and columns are fields within a record. The table itself does not display or edit data, since it delegates these processes to cell components. When you design a table, each column holds the cell component that will be used for that column. You must drop cell components onto the form (they are non-visual) and then attach each to its own column. Not only do cell components specify how the data is displayed and edited, but they also specify attributes like the background color and access.

Often tables have a locked row that is used for column headings at the top. Because this situation is so common, the table control has an explicit property called LockedRowsCell that holds a reference to a cell component that is used for painting the locked cells at the top of the columns.

By default the table has no cell components attached to it or to its columns. If you compile such a table (for example by dropping a table onto a blank form and then compiling the project), no data can be displayed in the grid.

The TOvcTable component does not store any user data itself. Every time a cell needs to be painted, an event is triggered and the OnGetCellData event is called. You must write a handler to provide data for the table before anything can be displayed.

The four basic steps in designing and creating a table are:

1. Drop a TOvcTable component onto the form.
2. Drop the required cell components for the field types onto the form. Also drop a cell component (such as a TOvcTCColHead cell) for the locked row(s) onto the form.
3. Set the various properties for the cell components and for the table. Link the cell components to the columns (using the Columns property editor) and link the cell component to the LockedRowsCell property.
4. Write an OnGetCellData handler. The handler receives a row and a column number, and must return a pointer to the relevant data (the data type is determined by the cell component for the cell).

Steps 1 to 3 are relatively easy to implement. Step 4, on the other hand, can be difficult and for example could require a caching scheme. The example at the beginning of this chapter shows an extremely simple OnGetCellData handler that uses an in-memory array.

A table consists of three areas: the locked cells, the normal cells, and the unused area. The locked cells stay fixed when the table scrolls. The locked cells along the left side do not change when the table scrolls left or right, and the locked cells along the top do not change when the table scrolls up or down. The normal cells are those in the main body of the table that display and edit the data. The unused area is any portion of the table that is not covered by either locked or normal cells.

The normal and locked cells are separated from each other by grid lines. You can specify different types of grid lines for locked and normal cells. The grid lines are drawn with the grid pens (objects of type TOvcGridPen) that are described in the table's GridPenSet property.

During design mode, the locked cells of the table serve as "hot spots" for moving columns and rows or to resize their widths and heights. As you pass the mouse cursor over the locked area it changes shape to denote when a row or column could be moved or when a row or column could be resized. Click and drag to move or resize a column or row. This capability is always available at design time in the IDE, but can be turned off at run time.

One special cell appears in the normal area of the table. It is called the active cell and that is where editing takes place. It is distinguished from the other cells by three attributes. It has a special background color, a special font color (if applicable), and a special grid pen. The grid pen is used to draw a focus box around the active cell. As a further visual aid, the table uses different colors and a different grid pen when the table control has the focus and when it doesn't. The active cell can be moved around the table by clicking on a cell with the mouse or by use of the arrow keys and other cursor movement keys.

If you are editing a cell and you enter an invalid value (the CanSaveEditedData method for the cell returns False) you will not be able to scroll the table. If the value you are entering is valid (CanSaveEditedData returns True) you will be able to scroll the table without coming out of editing mode.

Getting Data for the Table

One of the most important, and yet most difficult to write, events for the table is the OnGetCellData event. Without such an event handler, the table can't display any data. With a badly written event handler (for example, one that reads data from disk for every call), the table becomes unimaginably and unmanageably slow.

One small trap that many programmers fall into is to pass back the address of a local variable in the Data parameter. This can cause some weird effects when the cell tries to display the data. The reason for this is that local variables only last as long as the routine itself, so as soon as your OnGetCellData method terminates all local variables it was using are destroyed (they are actually on the stack, and the next routine to be called will allocate its own local variables on the stack, overwriting the ones you already had). It bears repeating: do not set Data to the address of a local variable.

If you are getting data from disk then it makes sense to have some kind of caching scheme whereby some records are held in memory ready for display. To help the caching process for the OnGetCellData method, there is an event (OnEnteringRow) that gets triggered whenever the data for a new row is about to be requested. (There is also an OnEnteringColumn event, in case your data storage requirements are a little more out of the ordinary.) OnEnteringRow is called once for each row as data is displayed in the table. It gives you an opportunity to read the record that contains the data for the row (or at least to ensure that it is in your cache).

When cells need to be painted, the internal algorithm attempts to paint them in ascending row number order. This means that all the cells that need painting for one row are painted before the cells for the next row. This triggers OnEnteringRow as few times as necessary.

A good first design for a record cache is an array of records containing at least as many elements as there are rows displayed in the table. You must store the record number with each record so that you can identify records. The OnEnteringRow event handler scans the array looking for the correct record number for the requested row. If the record is found, you could save the element index (or, even better, copy the record from the array into a global variable) for use by OnGetCellData.

If the record is not found, you must read it from the disk. If there is room in the cache (at least one element of the array is unused), read the record into an unused element. If there is no room, you must discard a record from the cache and read the record there. The simplest way of finding out which record to discard is to have a least-recently-used (LRU) counter in each element of the array. Every time a record is requested by OnEnteringRow, increment the counter. This means that the element that you can reuse is the one with the smallest LRU value.

How To...

With a component as powerful as the TOvcTable control, most of the information you need falls into the “How to” category. This section covers the most common situations. Check the README.TXT file for late-breaking information.

How do I make the table read only?

Set the table’s Access property to otxReadOnly and ensure that all cell components linked to the grid have their Access properties set to otxDefault.

How do I then make the table editable?

Set the table's Access property to otxNormal and ensure that all cell components linked to the grid have their Access properties set to otxDefault.

How do I get a 3D look for the normal cells in the Table?

The 3D look is provided by the table's grid pens. At design time, open the GridPenSet property in the Object Inspector, and then open the NormalGrid subfield. Set the Effect field to ge3D, set the Style field to psSolid. For a raised 3D look, set the NormalColor field to clBlack and the SecondColor field to clWhite. For a lowered 3D look do the reverse. (Note that officially you should use clBtnShadow as the "dark" color and clBtnHighlight as the "light" color.)

How do I lock certain rows or columns?

The only method of locking rows is to set the LockedRows property. If LockedRows is set to N, the first N rows in the table (rows 0 to N-1) are deemed locked. The same is True for locking columns (in which case you use the LockedCols property). The table has no facility for setting an arbitrary row or column as locked.

How do I change the color of the locked cells?

At design time, open the Colors property in the Object Inspector and set the Locked and LockedText fields to the desired colors.

How do I change the colors of the active cell?

At design time, open the Colors property in the Object Inspector and set the ActiveFocused and ActiveFocusedText fields to the desired colors. This changes the active cell's colors when the table has the focus. To change the colors of the active cell when the table does not have the focus, change the ActiveUnfocused and ActiveUnfocusedText fields instead.

How do I prevent the table from repainting while I make multiple changes?

Set the AllowRedraw property to False before the changes and then to True afterwards. The AllowRedraw property is a cumulative property, in that you must balance all the calls that set AllowRedraw to False with an equal number of calls to set it to True. If you do not, the table will never repaint itself again.

```
begin
  with MyTable do
    begin
      AllowRedraw := False;
      ..do a lot of changes..
      AllowRedraw := True;
    end;
  ...
```

How do I delete a column from the table?

At design time, use the Columns property editor. Enter the column number in the Column Number field and then click on the minus speedbutton.

At run time, call the Delete method of the table's Columns property. The method frees the column object and reduces the number of columns by one:

```
MyTable.AllowRedraw := False;
MyTable.Columns.Delete(5);
MyTable.AllowRedraw := True;
```

This example deletes the sixth column (columns are zero-based).

Your OnGetCellData and OnGetCellAttributes event handlers must also be notified of the change so that they no longer provide data for the deleted column. This is done by writing an OnColumnsChanged event handler.

How do I temporarily hide a column?

At design time, use the Columns property editor. Enter the column number in the Column Number field and then click on the Hidden checkbox.

At run time, set the relevant element of the Hidden array property of the table's Columns property:

```
MyTable.AllowRedraw := False;
MyTable.Columns.Hidden[5] := True;
MyTable.AllowRedraw := True;
```

An alternative is to set the Hidden property of the column itself:

```
with MyTable.Columns[5] do
  Hidden := True;
```

Change the property back to False to redisplay the column.

How do I insert a column into the table?

At design time, use the Columns property editor. Enter the column number for the new column in the Column Number field, and then click on the plus speedbutton. For example, if you insert a new column at column 4, then the existing column 4 becomes column 5, the old column 5 becomes column 6, and so on.

You cannot append a column to the very end of the column array by this method, the operation performed is an “insert before” operation, the new column is inserted before the column number in the Column Number field. To append a new column to the end of the array using the Columns Editor, insert a new column before the last column, make its style the same as the last column, and then change the style of the last column as required for the appended column.

At run time, create a new column object, attach a new cell component to it (or reuse a cell component that already exists), and then call the Insert method of the Columns property of the table. This inserts the new column, increments the number of columns by one, and causes the table to rebuild and repaint itself:

```
MyTable.AllowRedraw := False;
MyNewColumn := TOvcColumn.Create(MyTable);
MyCell := TOvcTCString.Create(MyForm);
with MyNewColumn do begin
  DefaultCell := MyCell;
  Width := 50;
end;
MyTable.Columns.Insert(5, MyNewColumn);
...
MyTable.AllowRedraw := True;
```

This code creates a new column and cell component, links them, and then inserts the new column as the 6th column.

Your OnGetCellData and OnGetCellAttributes event handlers must also be notified of the change so that they can provide data for the newly inserted column. This is done by writing an OnColumnsChanged event handler.

How do I change the number of columns in the table?

Set the table's ColCount property. This can be done at both design and run-time. The effect of this will either be to destroy the columns that are no longer used (if you are reducing the number of columns) or to create the new columns in their default form and append them (if you are increasing the number of columns).

How do I delete a row from the table?

At design time, use the Rows property editor. Enter the row number in the Row Number field and then press the minus speed button.

At run time, call the Delete method of the table's Rows property. The method deletes the row information and moves the rows below up by one:

```
MyTable.AllowRedraw := False;
MyTable.Rows.Delete(5);
MyTable.AllowRedraw := True;
```

Your OnGetCellData and OnGetCellAttributes event handlers must also be notified of the change so that they no longer provide data for the deleted row. This is done by writing an OnRowsChanged event handler.

This question does not refer to deleting a row in your underlying database (if you are using the one row equals one record scheme). What gets deleted is a row style. You must delete the record in your database that corresponds to this deleted style. Rows use the default style (unhidden and the default height) defined by the Rows property, unless they have a specific style set for them.

How do I temporarily hide a row?

At design time, use the Rows property editor. Enter the row number in the Row Number field and then click on the Hidden checkbox.

At run time, set the relevant element of the Hidden array property of the table's Rows property:

```
MyTable.AllowRedraw := False;
MyTable.Rows.Hidden[5] := True;
MyTable.AllowRedraw := True;
```

Change the property back to False to redisplay the row.

How do I insert a row into the table?

At design time, use the Rows property editor. Enter the row number for the new row in the Row Number field, and then click on the plus speedbutton.

At run time, declare a local TRowStyle variable, fill in its fields, and then call the Insert method of the Rows property of the table. This inserts the new row, increments the number of rows (RowLimit) by one, and causes the table to rebuild and repaint itself:

```
var
  MyRowStyle : TRowStyle;
begin
  with MyRowStyle do begin
    Height := MyTable.Rows.DefaultHeight;
    Hidden := False;
  end;
  MyTable.AllowRedraw := False;
  MyTable.Rows.Insert(5, MyRowStyle);
  ...
  MyTable.AllowRedraw := True;
```

This code initializes a TRowStyle variable and then inserts the new row as the 6th row.

Your OnGetCellData and OnGetCellAttributes event handlers must also be notified of the change so that they can provide data for the newly inserted row. This is done by writing an OnRowsChanged event handler.

This question does not refer to inserting a new row in your underlying database (if you are using the one row equals one record scheme). What gets inserted is a new row style. You must insert a new record in your database to correspond to this new style. Rows use the default style (unhidden and the default height) defined by the Rows property, unless they have a specific style set for them.

How do I change the number of rows in the table?

Set the table's RowLimit property. This can be done at both design and run-time.

How do I display column headings?

Make sure that your table has at least one locked row to display the headings. Drop a TOvcTCString cell component onto the form. Set its properties (for example, set the Adjust property to otaCenter). Set the table's LockedRowsCell to this new cell component.

At run time, you must ensure that your OnGetCellData method returns the heading strings when called with row 0 (the locked row) as the parameter.

This is one method. The more direct and more common method follows...

How do I set and display column headings, and see them at design time as well?

Drop a TOvcTCColHead cell component onto the form; link it to the LockedRowsCell property of the table. Set the column heading cell component's ShowLetters property to False. Now go into the cell's Headings string list property editor and enter your headings. Remember that line 1 in the string list editor is for column 0, line 2 is for column 1 and so forth.

At run time, ensure that your OnGetCellData event handler does not return any data for the locked rows, otherwise your supplied strings will override those in the Headings string list property.

Note that this is a different method than that shown in the previous "How-To"

How do I change the color/font/adjustment of a cell at run time?

You must write an OnGetCellAttributes event handler and link it to the table. The handler is passed a row and column number identifying the cell, and a record structure with the attributes calculated so far. Change the color, font, and adjustment field as required.

```
procedure TForm1.OvcTable1GetCellAttributes(
  Sender : TObject; RowNum : LongInt; ColNum : Integer;
  var CellAttr : TOvcCellAttributes);
begin
  if (Row = Col) then
    CellAttr.caColor := clLime;
end;
```

This especially useful handler ensures that all the cells on the main diagonal of the table (the row number equals the column number) are painted with a lime background. See the OnGetCellAttributes event for a description of TOvcCellAttributes.

How do I set the attributes for a block of cells?

At run time, set the BlockColBegin, BlockRowBegin, BlockColEnd and BlockRowEnd properties to define the block of cells. Then set the value of the required BlockXxx property (e.g., BlockColor).

```
begin
  with MyTable do begin
    AllowRedraw := False;
    BlockColBegin := 1;
    BlockColEnd := 2;
    BlockRowBegin := 1;
    BlockRowEnd := 5;
    BlockAccess := otxInvisible;
    AllowRedraw := True;
  end;
...
```

This code forces the rectangular range of cells starting at row 1, column 1 and ending at row 5, column 2 to be invisible.

Please note however that sometimes it can be easier (and more efficient) if you write an OnGetCellAttributes method. Generally, the reason for altering the attributes for a set of cells is a condition of the form “if such-and-such a cell value is positive then make the rest of the row read only”, and for this type of case it is always better to do this via the OnGetCellAttributes method.

How do I set the cell attributes for a whole column?

Set the attribute in the DefaultCell property for the column. This could be done with the table’s block properties, but it would be a waste of memory.

```
begin
  MyTable.AllowRedraw := False;
  with MyTable.Columns[5].DefaultCell do begin
    Adjust := otaCenter;
    Access := otxReadOnly;
  end;
  MyTable.AllowRedraw := True;
...
```

This code forces the 6th column to display its data in the center of each cell, and to make the data read-only.

How do I set the cell attributes for a whole row?

Since a row does not store any cell information, you must use the table's block properties. Set the BlockRowXxx properties to the row required, and BlockColBegin to 0 and BlockColEnd to ColCount-1. Then set the relevant BlockXxx property.

This also changes the locked cell at the start of the row. If you do not want this, set BlockColBegin to LockedCols.

```
with MyTable do begin
  AllowRedraw := False;
  BlockColBegin := LockedCols;
  BlockColEnd := pred(ColCount);
  BlockRowBegin := 5;
  BlockRowEnd := 6;
  BlockAccess := otxReadOnly;
  AllowRedraw := True;
end;
...
```

This code makes the sixth and seventh rows read-only. Data is displayed there, but cannot be edited. The locked cells at the start of the rows are not changed.

Please note however that sometimes it can be easier (and more efficient) if you write an OnGetCellAttributes method. Generally, the reason for altering the attributes for a set of cells is a condition of the form "if such-and-such a cell value is positive then make the rest of the row read only", and for this type of case it is always better to do this via the OnGetCellAttributes method.

How do I programmatically change the active cell?

Set the ActiveRow and ActiveCol properties of the table. To reduce the flicker as the active cell moves, set AllowRedraw to False first and then to True afterwards.

```
with MyTable do begin
  AllowRedraw := False;
  ActiveRow := ActiveRow + 2;
  ActiveCol := ActiveCol + 1;
  AllowRedraw := True;
end;
...
```

This code moves the active cell two rows down and one column to the right. An alternative is to make a call to the SetActiveCell method.

How do I get the table to wrap to the next row when the active cell is on the last column and I press the right arrow?

Create an OnActiveCellMoving event handler as follows:

```
procedure TForm1.OvcTable1ActiveCellMoving(
  Sender : TObject; Command : Word; var RowNum : LongInt;
  var ColNum : Integer);
begin
  case Command of
    ccRight :
      with OvcTable1 do
        if (ColNum = pred(ColCount)) and
          (ColNum = ActiveCol) then begin
          ColNum := LockedCols;
          Inc(RowNum);
          if (RowNum >= RowLimit) then
            RowNum := LockedRows;
        end;
    ...
  end;{case}
end;
```

This will cause a wraparound to the first normal cell on the next row, and if there is no such row, to the first normal row. The only time this comes into play is when (1) the passed column number is the final column, and (2) the passed column number is the active column; in other words the table wants to move beyond the last column.

Note that this answer will also cater for the similar commands that can occur at the table edges (for example, pressing <Down> on the last row).

How do I programmatically set the top left cell?

Set the TopRow and LeftCol properties of the table. To reduce the flicker as the table redraws itself, set AllowRedraw to False first and then to True afterwards.

```
with MyTable do begin
  AllowRedraw := False;
  TopRow := LockedRows;
  LeftCol := LockedCols;
  AllowRedraw := True;
end;
...
```

This code forces the cell at row 1 column 1 to the top left corner of the displayed table, underneath and to the right of the locked cells (assumed to be one column wide and one row deep). An alternative is to make a call to the SetTopLeftCell method.

How do I get the table to repaint itself, if it has stopped doing so?

This problem is caused by a call that sets the AllowRedraw property to False without a balancing call to set it True again. Check your code to make sure that the calls are balanced, since this is a programming error.

As a last resort you could code a routine like this and attach it to a button:

```
while not AllowRedraw do
  AllowRedraw := True;
```

This has all the finesse of using an oxy-acetylene torch to stir-fry vegetables.

How do I force a section of the table to repaint itself?

Set the AllowRedraw property to False, use one of the InvalidateXxx methods of the table, and then set AllowRedraw to True again.

```
AllowRedraw := False;
InvalidateRow(5);
AllowRedraw := True;
```

This code forces the table to repaint the 6th row. The repainting is actually done by the call to set AllowRedraw to True. The InvalidateXxx routines do not cause any repainting themselves. They just mark certain cells as requiring repainting. The repainting is performed the next time AllowRedraw is set True.

How can I change the look of the unused area?

Provide an OnPaintUnusedArea event handler. If one exists, it is called to paint the unused area. If one does not exist, the unused area is painted in the color specified by the table's Color property.

If you provide an OnPaintUnusedArea handler, but it doesn't actually do anything, the unused area does not get repainted. The handler can do anything from a simple FillRect to a more complex tiling of a small bitmap.

How do I access the edited data in an OnUserValidation event handler?

An example of this is when you have supplied an OnUserValidation event handler and linked it to a TOvcTCSimpleField cell component. At run time your OnUserValidation event handler gets called, but how do you get the edited data (the displayed string)? Use the Sender parameter, it is the editing control itself so all you need to do is typecast it:

```
with (Sender as TOvcSimpleField) do begin
    ...
  end;
```

The same trick can be used for other cell components.

How do I access the edited data in an OnEndEdit event handler?

An example of this situation is this: you have supplied an OnEndEdit event handler and linked it to a TOvcTCString cell component. At run time your OnEndEdit event handler gets called, but how do you get hold of the edited data (the displayed string)? The answer is to use the Cell parameter of the OnEndEdit method: it is the cell component. One of the properties of the cell component is called CellEditor and this is the editing control itself (in this example, a TEdit control). You can now typecast this control to a TEdit (or whatever) to get at the data.

How do I set special properties for a user-defined selection of cells?

The situation is this: the user has selected a range of cells (presumably by click+drag with the mouse, or shift+arrow with the keyboard). Through some menu option or button, the user now wishes to change one of the attributes for that range of cells. Let us assume that the cells are to be made invisible. One way of doing this would be:

```
function Form1.SelectionIterator(
  RowNum1 : TRowNum; ColNum1 : TColNum; RowNum2 : TRowNum;
  ColNum2 : TColNum; ExtraData : pointer) : Boolean;
begin
  BlockColBegin := ColNum1;
  BlockColEnd := ColNum2;
  BlockRowBegin := RowNum1;
  BlockRowEnd := RowNum2;
  BlockAccess := otxInvisible;
  Result := True;
end;
...
with OvcTable1 do begin
  AllowRedraw := False;
  IterateSelections(SelectionIterator, nil);
  AllowRedraw := True;
end;
```

How can I create a table with no rows?

You can’t. There must be at least one normal row in the table.

How do I access the data in a table cell while I'm editing the data?

Usually this is done within an event handler such as OnChange or OnClick. Since the table cell creates a temporary editing field compatible with its type, a simple typecast of the Sender parameter is all that's required. For example, if a TOvcTCSimpleField cell is attached to a table that is used to edit a string, you could do something like this:

```
uses,
  ...
  OvcSF;

procedure TForm1.OvcTCSimpleField1Change(Sender : TObject);
var
  S : string;
begin
  S := (Sender as TOvcSimpleField).AsString;
  {do something with S}
end;
```

The OvcSF unit is required to perform the typecast.

Not so obvious, but just as simple, is how to access the data in the TOvcTCCheckBox cell. Typecast the Sender to an ancestor class of the cell, e.g., TOvcTCGlyphEdit, and reference its member variable, Value. Here's a sample OnClick event handler:

```
procedure TForm1.OvcTCCheckBox1Click(Sender : TObject);
var
  V : TCheckBoxState;
begin
  V := TCheckBoxState((Sender as TOvcTCGlyphEdit).value);
  if (V = cbChecked) then
    Label1.Caption := 'Checked'
  else
    Label1.Caption := 'Not checked';
end;
```

TCheckBoxState is a VCL type. See the compiler's online help for more information.

In cases like the combo box, you must use the TCustomxxx version of the control (Sender as TCustomComboBox).

To be certain of which class should be used in the typecast, refer to the source code for the appropriate cell and look for a class descended from either a VCL or Orpheus edit control. For example, in OvcTCPic (which declares the TOvcTCPictureField cell class and methods), there is a class declaration of:

```
TOvcTCPictureFieldEdit = class(TOvcPictureField)
```

For the typecast, use the base VCL or Orpheus control, not the descendant class, i.e., TOvcPictureField and not TOvcTCPictureFieldEdit.

Table Commands

The following commands are available in the Orpheus table component. The table uses its own special command table, the "Grid" command table. See "TOvcCommandProcessor Class" on page 50 for information on how to customize the command tables.

Table 30.3: *Commands and their functions*

Command	**Grid command table**	**Description**
ccBotOfPage	<CtrlPgDn>	Move the active cell to the bottom of the page (the column remain unchanged).
ccBotRightCell	<CtrlDown>	Move the focus to the cell at the bottom-right of the table.
ccDown	<Down>	Move the focus to the next row.
ccEnd	<End>	Move the focus to the last column.
ccExtendDown	<ShiftDown>	Extend the selection down to the next row.
ccExtendEnd	<ShiftEnd>	Extend the selection to the last column.
ccExtendHome	<ShiftHome>	Extend the selection to the first column.
ccExtendLeft	<ShiftLeft>	Extend the selection left one column.
ccExtendPgDn	<ShiftPgDn>	Extend the selection down one page.
ccExtendPgUp	<ShiftPgUp>	Extend the selection up one page.
ccExtendRight	<ShiftRight>	Extend the selection to the right by one column.
ccExtendUp	<ShiftUp>	Extend the selection up one row.
ccExtFirstPage	<CtrlShiftHome>	Extend the selection to the first row (the column remains unchanged).
ccExtLastPage	<CtrlShiftEnd>	Extend the selection to the last row (the column remains unchanged).

Table 30.3: *Commands and their functions (continued)*

Command	**Grid command table**	**Description**
ccExtWordLeft	<CtrlShiftLeft>	Extend the selection to the left by one page.
ccExtWordRight	<CtrlShiftRight>	Extend the selection to the right by one page.
ccFirstPage	<CtrlHome>	Move the focus to the first cell in the table.
ccHome	<Home>	Move the active cell to the first column.
ccLastPage	<CtrlEnd>	Move the active cell to the last cell in the table.
ccLeft	<Left>	Move the active cell left to the previous cell.
ccNextPage	<PgDn>	Move the active cell to the page following the current page.
ccPageLeft	<CtrlLeft>	Move the active cell left a page (one window width).
ccPageRight	<CtrlRight>	Move the active cell right a page (one window width).
ccPrevPage	<PgUp>	Move the active cell to the previous page.
ccRight	<Right>	Move the active cell right to the next cell.
ccTableEdit	<F2>	Enter / exit table edit mode
ccTopLeftCell	<CtrlUp>	Move the active cell to the top left cell in a table.
ccTopOfPage	<CtrlPgUp>	Move the active cell to the top of the page (the column remains unchanged).
ccUp	<Up>	Move the active cell up to the previous row.

Hierarchy

TCustomControl (VCL)

❶TOvcCustomControl (OvcBase) 16

❷TOvcCustomControlEx (OvcBase) 19

TOvcTableAncestor (OvcTCmmn) 1021

TOvcCustomTable (OvcTable). 1022

TOvcTable (OvcTable)

Properties

❶ About	BlockColor	GridPenSet
Access	BlockFont	❶ LabelInfo
ActiveCol	BlockRowBegin	LeftCol
ActiveRow	BlockRowEnd	LockedCols
Adjust	BorderStyle	LockedRows
AllowRedraw	Cells	LockedRowsCell
❶ AttachedLabel	ColCount	Options
BlockAccess	ColOffset	RowLimit
BlockAdjust	Colors	RowOffset
BlockCell	ColorUnused	Rows
BlockColBegin	Columns	ScrollBars
BlockColEnd	❷ Controller	TopRow

Methods

CalcRowColFromXY	InSelection	ResolveCellAttributes
FilterKey	InvalidateCell	SaveEditedData
GetDisplayedColNums	InvalidateColumn	SetActiveCell
GetDisplayedRowNums	InvalidateRow	SetTopLeftCell
HaveSelection	InvalidateTable	StartEditingState
IncCol	IterateSelections	StopEditingState
IncRow	MoveActiveCell	
InEditingState	ProcessScrollBarClick	

Events

❶ AfterEnter	OnDoneEdit	❶ OnMouseWheel
❶ AfterExit	OnEndEdit	OnPaintUnusedArea
OnActiveCellChanged	OnEnteringColumn	OnRowsChanged
OnActiveCellMoving	OnEnteringRow	OnSizeCellEditor
OnBeginEdit	OnGetCellAttributes	OnTopLeftCellChanged
OnClipboardCopy	OnGetCellData	OnTopLeftCellChanging
OnClipboardCut	OnLeavingColumn	OnUserCommand
OnClipboardPaste	OnLeavingRow	
OnColumnsChanged	OnLockedCellClick	

Reference Section

Access property

```
property Access : TOvcTblAccess
```

```
TOvcTblAccess = (
  otxDefault, otxNormal, otxReadOnly,   otxInvisible);
```

Default: otxNormal

Determines the cell access rights.

If a cell component in the table has otxDefault as its Access property, then it will use this Access value.

The possible values for Access are:

Value	Description
otxDefault	The same as otxNormal.
otxNormal	The cell can be read and edited.
otxReadOnly	The cell can be read but not edited.
otxInvisible	The cell cannot be read or edited and is invisible (the cell is displayed as a blank rectangle in the background color).

See also: Adjust

ActiveCol property

```
property ActiveCol : TColNum
```

Default: 1

The column number of the active cell.

The column number must be between LockedCols and ColCount-1. If it is not, ActiveCol is set to LockedCols.

If you specify a hidden column, the next visible column is selected instead. If there is no next visible column, the previous visible column is selected.

See also: ActiveRow, ColCount, LockedCols, SetActiveCell

ActiveRow property

```
property ActiveRow : TRowNum
```

Default: 1

The row number of the active cell.

The row number must be between LockedRows and RowLimit-1. If it is not, ActiveRow is set to LockedRows.

If you specify a hidden row, the next visible row is selected instead. If there is no next visible row, the previous visible row is selected.

See also: ActiveCol, LockedRows, RowLimit, SetActiveCell

```
property Adjust : TOvcTblAdjust
```

```
TOvcTblAdjust = (otaDefault, otaTopLeft, otaTopCenter,
  otaTopRight, otaCenterLeft, otaCenter, otaCenterRight,
  otaBottomLeft, otaBottomCenter, otaBottomRight);
```

Default: otaCenterLeft

↳ Defines where data is displayed in the cell.

If a cell component in the table has otaDefault as its Adjust property, then it uses this Adjust value.

The possible values for Adjust are:

Value	**Description**
otaDefault	The same as otaCenterLeft.
otaTopLeft	The data is displayed in the top left corner of the cell.
otaTopCenter	The data is centered horizontally at the top of the cell.
otaTopRight	The data is displayed in the top right corner of the cell.
otaCenterLeft	The data is centered vertically at the left of the cell.
otaCenter	The data is centered vertically and horizontally in the cell.
otaCenterRight	The data is centered vertically at the right of the cell.
otaBottomLeft	The data is displayed in the bottom left corner of the cell.
otaBottomCenter	The data is centered horizontally at the bottom of the cell.
otaBottomRight	The data is displayed in the bottom right corner of the cell.

See also: Access

AllowRedraw — run-time property

```
property AllowRedraw : Boolean
```

Default: True

Controls when the table displays itself.

When AllowRedraw is False, the table will not paint itself. If you make numerous changes to the table, you should set AllowRedraw to False before making the changes and then set it to True when you are finished. Then the table repaints itself just once, at the end.

The AllowRedraw property is a cumulative property. You can set it to False as many times as you like, but you must then set it to True an equal number of times before the table will paint itself. You must ensure that the calls to set AllowRedraw to False are balanced by calls to set AllowRedraw to True. There is no method to interrogate the underlying counter. If the calls are not balanced, then the table will never paint itself.

AllowRedraw should not be published, and the value of AllowRedraw is not saved to the DFM file.

BlockAccess — write-only property

```
property BlockAccess : TOvcTblAccess
```

```
TOvcTblAccess = (
  otxDefault, otxNormal, otxReadOnly, otxInvisible);
```

Sets the access properties for a block of cells.

BlockAccess is a write-only property (it isn't possible to read access properties for a block of cells because they could all be different). If you need to read the access property for a cell, use the Access matrix property of the Cells property.

The cell block is the rectangle defined by the BlockColBegin, BlockColEnd, BlockRowBegin, and BlockRowEnd properties. You must define the block of cells by setting these four properties and then set the BlockAccess property to the desired value. The table then applies the access value to all cells in the block.

To reset the access values for a block of cells, define the block and then set the BlockAccess value to otxDefault.

The defined block of cells is not changed by setting a value in this property.

See also: BlockAdjust, BlockColBegin, BlockColEnd, BlockRowBegin, BlockRowEnd

BlockAdjust write-only property

```
property BlockAdjust : TOvcTblAdjust
```

```
TOvcTblAdjust = (otaDefault, otaTopLeft, otaTopCenter,
  otaTopRight, otaCenterLeft, otaCenter, otaCenterRight,
  otaBottomLeft, otaBottomCenter, otaBottomRight);
```

↳ Sets the adjustment properties for a block of cells.

BlockAdjust is a write-only property (it isn't possible to read adjustment properties for a block of cells because they could all be different). If you need to read the adjustment property for a cell, use the Adjust matrix property of the Cells property.

The cell block is the rectangle defined by the BlockColBegin, BlockColEnd, BlockRowBegin, and BlockRowEnd properties. You must define the block of cells by setting these four properties and then set the BlockAdjust property to the desired value. The table then applies the adjustment value to all cells in the block.

To reset the adjustment values for a block of cells, define the block and then set the BlockAdjust value to otaDefault.

The defined block of cells is not changed by setting a value in this property.

See also: BlockAccess, BlockColBegin, BlockColEnd, BlockRowBegin, BlockRowEnd

BlockCell write-only property

```
property BlockCell : TOvcBaseTableCell
```

↳ Sets the cell component for a block of cells.

BlockCell is a write-only property (it isn't possible to read the cell components for a block of cells because they could all be different). If you need to read the cell component for a cell, use the Cell matrix property of the Cells property.

The cell block is the rectangle defined by the BlockColBegin, BlockColEnd, BlockRowBegin, and BlockRowEnd properties. You must define the block of cells by setting these four properties and then set the BlockCell property to the desired value. The table then applies the cell component to all cells in the block.

To reset the cell component for a block of cells to that defined by the column(s), define the block and then set the BlockCell value to nil.

The defined block of cells is not changed by setting a value in this property.

See also: BlockColBegin, BlockColEnd, BlockRowBegin, BlockRowEnd

BlockColBegin property

```
property BlockColBegin : TColNum
```

Default: 0

↳ The column of the top left corner of a block of cells.

The cell block is the rectangle defined by the BlockColBegin, BlockColEnd, BlockRowBegin, and BlockRowEnd properties. To use the BlockXxx write-only properties, you must first define a cell block by setting these four properties, and then set the BlockXxx property value.

If you set BlockColBegin to a higher value than BlockColEnd, then BlockColEnd is changed to the new value of BlockColBegin.

Note that the cell block is different than the selection block, which is described in the overview at the beginning of this chapter.

See also: BlockColEnd, BlockRowBegin, BlockRowEnd

BlockColEnd property

```
property BlockColEnd : TColNum
```

Default: 0

↳ The column of the bottom right corner of a block of cells.

The cell block is the rectangle defined by the BlockColBegin, BlockColEnd, BlockRowBegin, and BlockRowEnd properties. To use the BlockXxx write-only properties, you must first define a cell block by setting these four properties, and then set the BlockXxx property value.

If you set BlockColEnd to a lower value than BlockColBegin, then BlockColBegin is changed to the new value of BlockColEnd.

Note that the cell block is different than the selection block, which is described in the overview at the beginning of this chapter.

See also: BlockColBegin, BlockRowBegin, BlockRowEnd

BlockColor write-only property

```
property BlockColor : TColor
```

↳ Sets the background color for a block of cells.

BlockColor is a write-only property (it isn't possible to read the background colors for a block of cells because they could all be different). If you need to read the Color property for a cell, use the Color matrix property of the Cells property.

The cell block is the rectangle defined by the BlockColBegin, BlockColEnd, BlockRowBegin, and BlockRowEnd properties. You must define the block of cells by setting these four properties and then set the BlockColor property to the desired value. The table then applies the background color to all cells in the block.

To reset the background color for a block of cells, define the block and then set the BlockColor value to clTableColorDefault (a constant defined in the OvcTCmmn unit).

BlockFont write-only property

```
property BlockFont : TFont
```

↳ Sets the font for a block of cells.

BlockFont is a write-only property (it isn't possible to read font for a block of cells because they could all be different).

The cell block is the rectangle defined by the BlockColBegin, BlockColEnd, BlockRowBegin, and BlockRowEnd properties. You must define the block of cells by setting these four properties and then set the BlockFont property to the desired value. The table then assigns the font to all cells in the block.

To reset the font for a block of cells to the one defined by the columns' DefaultCell, define the block and then set the BlockFont value to nil.

See also: BlockColBegin, BlockColEnd, BlockRowBegin, BlockRowEnd

BlockRowBegin property

```
property BlockRowBegin : TRowNum
```

Default: 0

The row of the top left corner of a block of cells.

The cell block is the rectangle defined by the BlockColBegin, BlockColEnd, BlockRowBegin, and BlockRowEnd properties. To use the BlockXxx write-only properties, you must first define a cell block by setting these four properties, and then set the BlockXxx property value.

If you set BlockRowBegin to a higher value than BlockRowEnd, then BlockRowEnd is changed to the new value of BlockRowBegin.

Note that the cell block is different than the selection block, which is described in the overview at the beginning of this chapter.

See also: BlockColBegin, BlockColEnd, BlockRowEnd

BlockRowEnd property

```
property BlockRowEnd : TRowNum
```

Default: 0

The row of the bottom right corner of a block of cells.

The cell block is the rectangle defined by the BlockColBegin, BlockColEnd, BlockRowBegin, and BlockRowEnd properties. To use the BlockXxx write-only properties, you must first define a cell block by setting these four properties, and then set the BlockXxx property value.

If you set BlockRowEnd to a lower value than BlockRowBegin, then BlockRowBegin is changed to the new value of BlockRowEnd.

Note that the cell block is different than the selection block, which is described in the overview at the beginning of this chapter.

BorderStyle property

```
property BorderStyle : TBorderStyle
```

Default: bsSingle

Determines the style used when drawing the border.

The possible values are bsSingle (a single line border is drawn around the component) or bsNone (no border line).

The TBorderStyle type is defined in the VCL's Controls unit.

CalcRowColFromXY method

```
function CalcRowColFromXY(X, Y : Integer;
  var RowNum : TRowNum;
  var ColNum : TColNum) : TOvcTblRegion;
```

```
TOvcTblRegion = (
  otrInMain, otrInLocked, otrInUnused, otrOutside);
```

Returns the cell address for an XY coordinate.

Given an XY coordinate (for example from a mouse click), this method calculates the address of the cell and the region of the table corresponding to the coordinate. The XY coordinate is assumed to be client relative, which means that the top left pixel of the table is assumed to be at (0,0).

The following diagram shows the regions in and around a table. A table is made up of normal cells (region 6) and locked cells (region 5). The large box is the table's client rectangle

For an XY coordinate in each of the various regions, the table below shows the values for the function result and the returned row and column numbers. In this table, "row number" means that the actual row number is returned and "column number" means that the actual column number is returned. In some cases, the row number and column number are obtained by assuming that the rows and columns extend out indefinitely from the used part of the table.

Region	Function result	Row	Column
1	otrOutside	CRCFXY_RowAbove	CRCFXY_ColToLeft
2	otrOutside	CRCFXY_RowAbove	column number
3	otrOutside	CRCFXY_RowAbove	CRCFXY_ColToRight
4	otrOutside	row number	CRCFXY_ColToLeft

Region	Function result	Row	Column
5	otrInLocked	row number	column number
6	otrInMain	row number	column number
7	otrInUnused	row number	CRCFXY_ColToRight
8	otrOutside	row number	CRCFXY_ColToRight
9	otrOutside	CRCFXY_RowBelow	CRCFXY_ColToLeft
10	otrInUnused	CRCFXY_RowBelow	column number
11	otrInUnused	CRCFXY_RowBelow	CRCFXY_ColToRight
12	otrOutside	CRCFXY_RowBelow	column number
13	otrOutside	CRCFXY_RowBelow	CRCFXY_ColToRight

Cells — run-time, read-only property

```
property Cells : TOvcTableCells
```

The matrix of individual cell attributes.

This property gives you access to the individual cell matrix, and therefore to the individual attributes of each cell. See “TOvcTableCells Class” on page 916 for more information.

ColCount — property

```
property ColCount : TColNum
```

Default: 10

The number of columns in the table.

The columns are numbered from 0 to ColCount-1.

When the table is created, it creates an instance of the TOvcTableColumns class (access to it is through the Columns property) passing a pointer to itself, and the default value of this property (i.e., 10). The only reason for automatically creating 10 columns is so that columns are visible immediately in the Form Designer when a new table component is dropped onto a form. For details on the default style for these columns, see “TOvcTableColumns Class” on page 910.

When a new value is written to ColCount there are three possible cases. One, the new value is the same as the old, in which case nothing happens. Two, the new value is greater than the old, in which case the required number of columns (i.e. the difference between the old and new values) are created in the same manner as when the table is initially created. Three, the new value is less than the old value, in which case the surplus columns are disposed (the number being the difference between the old and new values).

See also: Columns, RowLimit

ColOffset — run-time, read-only property

```
property ColOffset[ColNum : TColNum] : Integer
```

↳ An array of the starting offsets of the columns.

The ColOffset array enables you to find the starting offset of each column within the client area of the table. The offset is measured in pixels from the left edge of the client area (the first displayed column in the table has an offset of 0). If a column is not displayed when ColOffset is called, the returned value of the element is -1. To find out the numbers of columns that are displayed, use GetDisplayedColNums.

The ending offset of each displayed column is the starting offset of the next column. The ending offset of the last column is equal to the starting offset plus the column's width (use the Width array property of the Columns property).

See also: GetDisplayedColNums, RowOffset

Colors — property

```
property Colors : TOvcTableColors
```

↳ The set of colors used to paint cells.

See "TOvcTableColors Class" on page 882 for details on the colors used by a table.

ColorUnused — property

```
property ColorUnused : TColor
```

Default: clWindow

↳ The color, which is used to paint the unused area of the table.

See also: Colors

Columns property

```
property Columns : TOvcTableColumns
```

↳ The array of columns for the table.

Columns is an instantiated object of type TOvcTableColumns. See “TOvcTableColumns Class” on page 885 for a description of this object. If you set the Columns property, the current internal columns object is freed and the internal columns object is set to point to the new columns object. After the assignment there is one copy of the column information, and so the source object cannot be disposed without affecting the table.

See also: Rows

FilterKey method

```
function FilterKey(
  var Msg : TWMKey) : TOvcTblKeyNeeds; virtual;
```

```
TOvcTblKeyNeeds = (otkDontCare, otkWouldLike, otkMustHave);
```

↳ Called from a cell to filter a keystroke.

For every keystroke entered into an editing control, its cell component calls this method of its owning table so that the table can filter the keystroke. The table returns the category of the keystroke as a value of type TOvcTblKeyNeeds.

If the return value is otkDontCare, the table is informing the cell (and its editing control) that it does not want to use this keystroke. The editing control can take any action on the key.

If the return value is otkWouldLike, the table would like this keystroke, but it is leaving the decision to the cell and the editing control. Keystrokes in this category are the arrow keys and the active cell movement keys. If, for example, the editing cell does not use the <Up> keystroke, it can let the table use it instead.

If the return value is otkMustHave, the table absolutely must have the keystroke. There are only two keystrokes in this category and both are used to terminate the cell editing session. <Esc> aborts the editing and <F2> saves the new data.

If the return value is otkDontCare or otkWouldLike, the cell component uses SendMessage to send a WM_KEYDOWN message containing this keystroke directly to the table.

```
procedure GetDisplayedColNums(
  var NA : TOvcTableNumberArray);
```

```
POvcTableNumberArray = ^TOvcTableNumberArray;
```

```
TOvcTableNumberArray = record
  NumElements : LongInt;
  Count       : LongInt;
  Number      : array [0..29] of LongInt; {Number array}
end;
```

Returns an array of displayed column numbers.

The method is passed a TOvcTableNumberArray record structure and returns the structure filled in with the numbers of the currently displayed columns. GetDisplayedColNums makes no distinction between locked and normal columns; it returns the numbers of them all.

Before calling GetDisplayedColNums, you must initialize the first field of the TOvcTableNumberArray (NumElements) to the number of elements available in the Number array field. For example, if you declare a TOvcTableNumberArray variable as a local variable, then the NumElements value should be set to 30. If, however, you allocate some memory on the heap for a possible 50 column numbers (i.e., GetMem(MyNumberArray, 52*sizeof(LongInt)), where MyNumberArray is a variable of type POvcTableNumberArray), then set NumElements to 50.

GetDisplayedColNums calculates the numbers of the displayed columns and puts them (in ascending order) in the Number array field of the structure. It uses the NumElements value to ensure that it doesn’t write beyond the end of the array. It puts the actual number of displayed columns (including the partially displayed column on the right if there is one) in the Count field of the structure.

If the returned Count is less than or equal to NumElements, all the column numbers were retrieved. If the returned Count is greater than NumElements, only the first NumElements column numbers were retrieved and Count is the actual number of displayed columns. You need to make another call with a larger structure to get all the column numbers.

See also: GetDisplayedRowNums

GetDisplayedRowNums method

```
procedure
  GetDisplayedRowNums(var NA : TOvcTableNumberArray);
```

```
POvcTableNumberArray = ^TOvcTableNumberArray;
```

```
TOvcTableNumberArray = record
  NumElements : LongInt;
  Count       : LongInt;
  Number      : array [0..29] of LongInt;
end;
```

Returns an array of displayed row numbers.

The method is passed a TOvcTableNumberArray record structure and returns the structure filled in with the numbers of the currently displayed rows. GetDisplayedRowNums makes no distinction between locked and normal rows, it returns the numbers of them all.

This method works just like GetDisplayedColNums, but refers to rows instead of columns.

See also: GetDisplayedColNums

GridPenSet property

```
property GridPenSet : TOvcGridPenSet
```

The set of grid pens used to draw the grid lines.

See “TOvcGridPenSet Class” on page 891 for details about the grid lines and how they are used by the table.

HaveSelection method

```
function HaveSelection : Boolean;
```

Returns True if there is a selection.

See also: InSelection

```
function IncCol(
  ColNum : TColNum; Direction : Integer) : TColNum;
```

Increments (or decrements) a column number.

IncCol is a helpful internal routine used for calculating the number of the next or previous column to a passed column. ColNum is the reference column at which you start and Direction indicates whether you want the number of the next or previous column.

If Direction is a positive number, IncCol finds the next column along from ColNum that is in the main part of the table and is not hidden. If there is no such next column, then ColNum is returned as the function result. If there is such a column, its number will be returned as the function result.

If Direction is a negative number, IncCol finds the previous column before ColNum that is in the main part of the table and is not hidden. If there is no such previous column, then ColNum is returned as the function result. If there is such a column, its number will be returned as the function result.

If Direction is 0, IncCol verifies that ColNum is a column in the main part of the table and that it is not hidden. If it isn't, IncCol will search forwards looking for the next column that satisfies this condition. If it cannot find one, it will search backwards looking for the previous column that satisfies the same condition.

IncCol is used extensively in the internal table code, especially for operations such as moving the active cell right or left.

See also: IncRow

IncRow method

```
function
IncRow(RowNum : TRowNum; Direction : Integer) : TRowNum;
```

↳ Increments (or decrements) a row number.

IncRow is a helpful internal routine used for calculating the number of the next or previous row to a passed row. RowNum is the reference row at which you start and Direction indicates whether you want the number of the next or previous row.

If Direction is a positive number, IncRow finds the next row along from RowNum that is in the main part of the table and is not hidden. If there is no such next row, then RowNum is returned as the function result. If there is such a row, its number will be returned as the function result.

If Direction is a negative number, IncRow finds the previous row before RowNum that is in the main part of the table and is not hidden. If there is no such previous row, then RowNum is returned as the function result. If there is such a row, its number will be returned as the function result.

If Direction is 0, IncRow verifies that RowNum is a row in the main part of the table and that it is not hidden. If it isn't, IncRow will search forwards looking for the next row that satisfies this condition. If it cannot find one, it will search backwards looking for the previous row that satisfies the same condition.

IncRow is used extensively in the internal table code, especially for operations such as moving the active cell right or left.

See also: IncCol

InEditingState method

```
function InEditingState : Boolean;
```

↳ Returns True if the active cell is being edited.

This function gives a simple test to find out whether the table is currently in edit mode, i.e. that the active cell is being edited.

See also: StartEditingState, StopEditingState

InSelection method

```
function InSelection(
  RowNum : TRowNum; ColNum : TColNum) : Boolean;
```

↳ Returns True if the specified cell is within the selection.

This method returns True if the specified cell falls within the current selection.

If HaveSelection returns False, InSelection returns False for any cell.

See also: HaveSelection

InvalidateCell method

```
procedure
InvalidateCell(RowNum : TRowNum; ColNum : TColNum);
```

↳ Marks the cell as requiring repainting.

This method sets an internal flag to indicate that the cell needs repainting. The next time AllowRedraw is set to True, all invalid cells will be repainted.

See also: AllowRedraw, InvalidateColumn, InvalidateRow, InvalidateTable

InvalidateColumn method

```
procedure InvalidateColumn(ColNum : TColNum);
```

↳ Marks the specified column as requiring repainting.

This method sets an internal flag to indicate that the specified column needs repainting (it makes a call to InvalidateCell for each cell in the column). The next time AllowRedraw is set to True, all invalid cells will be repainted.

See also: AllowRedraw, InvalidateCell, InvalidateRow, InvalidateTable

InvalidateRow method

```
procedure InvalidateRow(RowNum : TRowNum);
```

↳ Marks the specified row as requiring repainting.

This method sets an internal flag to indicate that the specified row needs repainting (it makes a call to InvalidateCell for each cell in the row). The next time AllowRedraw is set to True, all invalid cells will be repainted.

See also: AllowRedraw, InvalidateCell, InvalidateColumn, InvalidateTable

InvalidateTable method

```
procedure InvalidateTable;
```

↳ Marks the entire table as requiring repainting.

This method sets an internal flag to indicate that the entire table needs repainting (it makes a call to InvalidateCell for each cell in the table). The next time AllowRedraw is set to True, all invalid cells will be repainted.

See also: AllowRedraw, InvalidateCell, InvalidateColumn, InvalidateRow

IterateSelections method

```
procedure IterateSelections(
  SI : TSelectionIterator; ExtraData : pointer);
```

```
TSelectionIterator = function(RowNum1 : TRowNum;
  ColNum1 : TColNum; RowNum2 : TRowNum; ColNum2 : ColNum;
  ExtraData : pointer) : Boolean of object;
```

↳ Calls a method for each cell range selection.

SI is called for each cell range selection that the user has made. The selected cell range is given by (RowNum1, ColNum1) at the top left corner and (RowNum2, ColNum2) at the bottom right corner. All cells in between (inclusive) are selected. ExtraData is a pointer to a user-defined structure that you pass to ForAll. The SI routine returns True if the iteration through the selected ranges is to continue, False otherwise.

ExtraData is a pointer to a user-defined structure. IterateSelections passes this pointer unchanged to the SI routine. You can use it to declare any common data that must be used in the iteration and use it in the iterator routine. Pass nil is you have no extra data to use.

Note that in general IterateSelections ‘slices up’ selections column by column. In other words SI will be called for all the selected strips for column 0, then for all the selected strips in column 1 and so on. This is a direct result of the internal data structure used to store the multiple disjoint selections. For further details on this data structure, please refer to the OVCTSELL.PAS source file.

The following example shows how to use IterateSelections. It populates a listbox on the form with the cell ranges that are selected in the table.

```
function TForm1.MyIterator(RowNum1 : TRowNum; ColNum1 : TColNum;
                           RowNum2 : TRowNum; ColNum2 : TColNum;
                           ExtraData : pointer) : Boolean;
begin
  Result := True;
  Inc(Integer(ExtraData^));
  ListBox1.Items.Add(Format('(%d,%d) .. (%d,%d)',
                            [RowNum1, ColNum1, RowNum2,
                             ColNum2]));
end;

procedure TForm1.Button1Click(Sender : TObject);
var
  SelCount : Integer;
begin
  ListBox1.Clear;
  SelCount := 0;
  RatingsGrid.IterateSelections(MyIterator, @SelCount);
  if SelCount = 0 then
    ListBox1.Items.Add('(none)');
end;
```

LeftCol property

```
property LeftCol : TColNum
```

Default: 1

↳ The number of the column on the left edge of the main table area.

The column number must be between LockedCols and ColCount-1. If it is not, LeftCol is set to LockedCols.

The table actually tries to ensure that its right-most page is as full as possible (i.e. that the unused area shown on that page is as small as possible). This in turn means that not all columns can become the left-most column: if you try and set LeftCol for the far right-most columns in the table it calculates the last column that satisfies the above criterion, and sets LeftCol to that instead.

If you specify a hidden column, the next visible column is selected instead. If there is no next visible column, the previous visible column is selected.

See also: SetTopLeftCell, TopRow

LockedCols property

```
property LockedCols : TColNum
```

Default: 1

↳ The number of fixed columns on the left side of the table.

The first LockedCols columns (from column 0 to LockedCols-1) are fixed; but the actual visible number of locked columns depends on whether any of these columns are hidden. If the value used to set LockedCols is not in the range 0 to ColCount-1, LockedCols is not changed.

See also: LockedRows

LockedRows property

```
property LockedRows : TRowNum
```

Default: 1

↳ The number of fixed rows at the top of the table.

The first LockedRows rows (from row 0 to LockedRows-1) are fixed; but the actual visible number of locked rows depends on whether any of these rows are hidden. If the value used to set LockedRows is not in the range 0 to RowLimit-1, LockedRows is not changed.

See also: LockedCols

LockedRowsCell property

```
property LockedRowsCell : TOvcBaseTableCell
```

Default: none

↳ The cell component used for displaying locked rows.

Generally the columns in a table are different cell types. However, there is usually a locked row at the top of all the columns for headings, and these cells should be displayed with a different cell component. The LockedRowsCell property is a simple way to declare the cell component for the column headings.

You can declare individual locked cells to have different cell components. For information, see “How do I set the attributes for a block of cells?” on page 1031.

```
procedure MoveActiveCell(Command : Word);
```

Moves the active cell to the specified position.

The active cell is moved according to the Command parameter. MoveActiveCell forces the active cell to be visible, so the table is scrolled if required so that the new position of the active cell is visible.

The following lists the commands that are processed by this method. All other commands are ignored and no exception is raised.

Command	**Description**
ccBotOfPage	Move the active cell to the row at the bottom of the page (the column remains unchanged). The active cell is moved to the last fully visible row on the main part of the page. If there is only one row visible on the page, this method does nothing.
ccBotRightCell	Move the active cell to the bottom right corner of the table. The active cell is usually moved to RowLimit-1, ColCount-1 but the location is adjusted to select the cell that's in the lowest unhidden row and in the rightmost unhidden column. The table is scrolled if necessary to make the active cell visible.
ccDown	Move the active cell down one row. The active cell is moved to the next row that is unhidden. If there are no more unhidden rows, the method does nothing. The table is scrolled if necessary to make the active cell visible.
ccHome	Move the active cell to the first unlocked column in the current row. The active cell is usually moved to the first column after the locked columns (the column specified by LockedCols), but it is adjusted to select the first unhidden column. The table is scrolled if necessary to make the active cell visible.
ccFirstPage	Move the active cell to the first unlocked row in the current column. Works like ccHome, but applies to rows.
ccEnd	Move the active cell to the last unhidden column in the current row. The active cell is usually moved to the column specified by ColCount-1, but if this column is hidden, the column number is adjusted to select the last unhidden column. The table is scrolled if necessary to make the active cell visible.

Command	Description
ccLastPage	Move the active cell to the last unhidden row in the current column. Works like ccEnd, but applies to rows.
ccLeft	Move the active cell left one column. The active cell is moved to the first previous column that is unhidden. If there are no previous unhidden columns, the method does nothing. The table is scrolled if necessary to make the active cell visible.
ccNextPage	Move the active cell down one page. The current partially (or fully) displayed row at the bottom of the page is made the top row (scrolling the table as required). If the last row of the table is already fully shown, it is made the top row. If the top row does not change, the active cell is already at the bottom of the table, and this method will do nothing. The active cell is positioned in roughly the same position on the new page as it was on the old page. If this is impossible (for example if there are not enough rows on the new page), then the last row on the page is selected.
ccPageLeft	Move the active cell left one page. The previous page is calculated so that the column that is currently displayed at the left of the table is displayed as far right as possible and still (partially) shown. If this does not cause the table to scroll (i.e., the left-most column of the page is the first unlocked, visible column of the table), the active column is set to the left-most column. The active cell is positioned in roughly the same position on the new page as it was on the old page. If this is impossible (for example if there are not enough columns on the new page), then the rightmost column on the new page is selected.
ccPageRight	Move the active cell right one page. Works like ccPageLeft, but in the opposite direction.
ccPrevPage	Move the active cell up one page. Works like ccNextPage, but in the opposite direction.
ccRight	Move the active cell right one column. Works like ccLeft, but in the opposite direction.

Command	Description
ccTopLeftCell	Move the active cell to the top left corner of the table. The active cell is usually moved to LockedRows, LockedCols but the location is adjusted to select the cell that's in the highest visible row and in the left-most unhidden column. The table is scrolled if necessary to make the active cell visible.
ccTopOfPage	Move the active cell to the row at the top of the page (the column remains unchanged). Works like ccBotOfPage, but in the opposite direction.
ccUp	Move the active cell up one row. Works like ccDown, but in the opposite direction.

OnActiveCellChanged event

```
property OnActiveCellChanged : TCellNotifyEvent
```

```
TCellNotifyEvent = procedure(Sender : TObject;
  RowNum : TRowNum; ColNum : TColNum) of object;
```

Defines an event handler that is called after the active cell changes.

The method assigned to the OnActiveCellChanged event is called with the new active column and row. For example, this event gives you an opportunity to update a status line.

It is a programming error to change the active cell within your event handler.

The following example displays the active cell in a label control in the form Rn:Cm where n is the row number and m the column number.

```
  procedure TForm1.OvcTable1ActiveCellChanged(
    Sender : TObject; RowNum : LongInt; ColNum : Integer);
  begin
    with Label1 do begin
      Caption := Format('R%d:C%d', [RowNum, ColNum]);
      Update;
    end;
  end;
```

```
property OnActiveCellMoving : TCellMoveNotifyEvent
```

```
TCellMoveNotifyEvent = procedure(
  Sender : TObject; Command : Word; var RowNum : TRowNum;
  var ColNum : TColNum) of object;
```

↳ Defines an event handler that is called just before the active cell is moved by user interaction.

The method assigned to the OnActiveCellMoving event is triggered by any user interaction that would move the active cell (for example pressing the up arrow on the keyboard or clicking on a cell with the mouse). The table calculates the new position of the active cell, and then triggers the event, passing the new address in the RowNum and ColNum parameters. This gives you an opportunity to change the new address of the active cell by altering the values of the RowNum and ColNum parameters (do not change the values of ActiveCol and ActiveRow to achieve this). When the event is triggered, ActiveCol and ActiveRow contain the old address of the active cell (where the active cell is moving from).

The Command parameter defines the Orpheus command that triggered the event. Its value is one of the ccXxx values defined in the OvcConst unit. For a mouse click or movement (when selecting), the value is ccMouse.

The following example makes column 5 unselectable, any attempt to move the active cell into column 5 will cause column 4 or 6 to be chosen instead. If the active cell is moving left, it is forced to go to column 4. If the active cell is moving right, it is forced to go to column 6. If the active cell is moved by the mouse, it is forced to column 6.

```
procedure TForm1.OvcTable1ActiveCellMoving(
  Sender : TObject; Command : Word;
  var RowNum : LongInt; var ColNum : Integer);
begin
  if (ColNum = 5) then
    case Command of
      ccLeft, ccPageLeft : ColNum := 4;
      ccRight, ccPageRight : ColNum := 6;
      ccMouse : ColNum := 6;
    end;{case}
end;
```

See also: OnTopLeftCellChanged, OnTopLeftCellChanging

OnBeginEdit event

```
property OnBeginEdit : TCellBeginEditNotifyEvent
```

```
TCellBeginEditNotifyEvent = procedure(
  Sender : TObject; RowNum : TRowNum; ColNum : TColNum;
  var AllowIt : Boolean) of object;
```

↳ Defines an event handler that is called just before the table switches into edit mode for the active cell.

The event is triggered for confirmation purposes: you are being asked to confirm that the active cell can be edited. Sender is generally the table control itself. The RowNum and ColNum parameters identify the cell that is about to be edited (they will equal ActiveRow and ActiveCol respectively).

The AllowIt parameter is True when the event handler is called. You can set it False if you do not want the active cell to be edited for any reason.

See also: Access, OnDoneEdit, OnEndEdit

OnClipboardCopy event

```
property OnClipboardCopy : TNotifyEvent
```

↳ Defines an event handler that is called when the user has requested a copy-to-clipboard operation.

Since the table knows nothing about the data in the table (it does not store either the data or the displayable representation of the data), it cannot provide any meaningful action for a copy-to- clipboard request.

When the user issues a copy-to-clipboard request (either from a menu or from the keyboard), the table triggers the OnClipboardCopy event. Your event handler is responsible for working out the selected cells, the data in those cells, a clipboard format for those cells, and possibly a displayable representation of those cells.

TNotifyEvent is defined in the VCL's Classes unit.

OnClipboardCut event

```
property OnClipboardCut : TNotifyEvent
```

↳ Defines an event handler that is called when the user requests a cut-to- clipboard operation.

Since the table knows nothing about the data in the table (it does not store either the data or the displayable representation of the data), it cannot provide any meaningful action for a cut-to- clipboard request.

When the user issues a cut-to-clipboard request (either from a menu or from the keyboard), the table triggers the OnClipboardCut event. Your event handler is responsible for working out the selected cells, the data in those cells, a clipboard format for those cells, and possibly a displayable representation of those cells. You are also responsible for performing the 'delete' part of the cut request (actually removing the data from your internal data structure).

TNotifyEvent is defined in the VCL's Classes unit.

OnClipboardPaste event

```
property OnClipboardPaste : TNotifyEvent
```

↳ Defines an event handler that is called when the user requests a paste- from-clipboard operation.

Since the table knows nothing about the data in the table (it does not store either the data or the displayable representation of the data), it cannot provide any meaningful action for a paste- from-clipboard request. When the user issues a paste-from-clipboard request (either from a menu or from the keyboard), the table triggers the OnClipboardPaste event. Your event handler is responsible for determining the selected cells, the data in those cells, and the clipboard format for the pasted data.

TNotifyEvent is defined in the VCL's Classes unit.

OnColumnsChanged event

```
property OnColumnsChanged : TColChangeNotifyEvent

TColChangeNotifyEvent = procedure (
  Sender : TObject; ColNum1, ColNum2 : TColNum;
  Action : TOvcTblActions) of object;

TOvcTblActions = (taGeneral, taSingle,
  taAll, taInsert, taDelete, taExchange);
```

Defines an event handler that is called when the user changes the columns of the table.

The method assigned to the OnColumnsChanged event is called when the user changes the configuration of the columns in the table by inserting a column, deleting a column, or swapping two columns. Action defines what happened. If Action equals taInsert or taDelete, the column number is in ColNum1. If Action equals taExchange, the two columns are in ColNum1 and ColNum2. No other TOvcTblActions values are used.

This event gives you an opportunity to change your data structure (or to update a mapping) to reflect the insertion, deletion, or exchange. See the EXTBL02U example program for an example of using this event.

OnDoneEdit event

```
property OnDoneEdit : TCellNotifyEvent

TCellNotifyEvent = procedure(Sender : TObject;
  RowNum : TRowNum; ColNum : TColNum) of object;
```

Defines an event handler that is called just after the data has been saved from the edited cell.

OnDoneEdit enables you to make any automatic saves to your database. At the time the method assigned to the OnDoneEdit event is triggered, the StopEditing method of the underlying cell component has already been called and the data saved.

Caution: OnDoneEdit is triggered in the middle of a focus change – you cannot cause another focus change to happen (by displaying a dialog box or other window, for example) during your event handler.

See also: OnBeginEdit, OnEndEdit

OnEndEdit event

```
property OnEndEdit : TCellEndEditNotifyEvent
```

```
TCellEditNotifyEvent = procedure(Sender : TObject;
  Cell : TOvcTableCellAncestor; RowNum : TRowNum;
  ColNum : TColNum; var AllowIt : Boolean) of object;
```

Defines an event handler that is called just before the data in the editing control has been saved.

The method assigned to the OnEndEdit event is triggered for confirmation purposes: you are being asked to confirm that the newly edited data can be saved to the active cell. Sender is generally the table control itself. Cell is the cell component that is being used for the cell (its CellEditor property is the control that is doing the actual editing). The RowNum and ColNum parameters identify the cell that is has been edited (they will equal ActiveRow and ActiveCol respectively). The AllowIt parameter is True when the event handler is called. You can set it False if you want the active cell to continue to be edited for any reason. The event is triggered after the CanSaveEditedData method for the underlying cell component has been called (and if this returns False, the event is not triggered) but before the StopEditing method.

Caution: OnEndEdit is triggered in the middle of a focus change; you cannot cause another focus change to happen (by displaying a dialog box or other window, for example) during your event handler.

See also: OnBeginEdit, OnDoneEdit

OnEnteringColumn event

```
property OnEnteringColumn : TColNotifyEvent
```

```
TColNotifyEvent = procedure(
  Sender : TObject; ColNum : TColNum) of object;
```

Defines an event handler that is called just before the table requests data for a new column.

When the table repaints itself, it displays cells in row order, and within each row in reverse column order. For each new column, the table triggers the OnEnteringColumn event just prior to triggering the OnGetCellData event to warn you that it is about to request data for the new column. This is a good time to make sure that the data is in the cache before the request to read it occurs.

In the default table model, rows are records and columns are fields and it is assumed that entire records are read at once. Therefore, the OnEnteringColumn event is not that important (presumably the fields for the current record are already in the cache).

This event handler must not change any table properties that would cause the table to be repainted or change the active cell.

See also: OnEnteringRow, OnLeavingColumn

OnEnteringRow event

```
property OnEnteringRow : TRowNotifyEvent
```

```
TRowNotifyEvent = procedure(
  Sender : TObject; RowNum : TRowNum) of object;
```

Defines an event handler that is called just before the table requests data for a new row.

When the table repaints itself, it displays cells in row order, and within each row in reverse column order. For each new row, the table triggers the OnEnteringRow event just prior to triggering the first OnGetCellData event to warn you that it is about to request data for the new row. This is a good time to make sure that the data is in the cache before the request to read it occurs.

In the default table model, rows are records and columns are fields and it is assumed that entire records are read at once. Therefore, the OnEnteringRow event is important in that it allows you to make sure that the entire record is in memory or in the cache.

This event handler must not change any table properties that would cause the table to be repainted or change the active cell.

The following example uses the OnEnteringRow event handler to read a record from a flat file. This example is meant for illustrative purposes only. You should cache a set of records in memory rather than read the record from the file every time.

```
  procedure TForm1.OvcTable1EnteringRow(Sender : TObject; var
  RowNum : LongInt);
  begin
    if (RowNum >= OvcTable1.LockedRows) then begin
      Seek(F, RowNum - OvcTable1.LockedRows);
      Read(F, DatabaseRecord);
    end;
  end;
```

F is assumed to be a file of some record type, and DatabaseRecord is a global instance of that record type. The call to the Seek procedure moves the file pointer to the required record. If you have 1 locked row, then you need to read record 0 when the event is triggered for row 1. You don’t need to read a record for row 0, the locked row, because it is presumed to be column headings, for example.

See also: OnEnteringColumn, OnLeavingRow

```
property OnGetCellAttributes : TCellAttrNotifyEvent

TCellAttrNotifyEvent = procedure(
  Sender : TObject; RowNum : TRowNum; ColNum : TColNum;
  var CellAttr : TOvcCellAttributes) of object;

TOvcCellAttributes = record
  caAccess      : TOvcTblAccess;   {access rights}
  caAdjust      : TOvcTblAdjust;   {data adjustment}
  caColor       : TColor;          {background color}
  caFont        : TFont;           {text font}
  caFontColor   : TColor;          {text color}
  caFontHiColor : TColor;          {text highlight color}
  caTextStyle   : TOvcTextStyle;   {text style}
end;
```

Defines an event handler that is called after the table calculates the default attributes for a cell.

When the table repaints itself, it displays cells in row order, and within each row in reverse column order. For each cell, the table triggers the OnGetCellAttributes event just after calculating the default attributes for the cell, and just prior to calling the Paint method of the

cell component that is in charge of repainting that cell. This event gives you the opportunity to make final adjustments to the attributes for the cell. The changes made in this event handler override all other attributes.

The cell's address is given by RowNum and ColNum, and the default set of attributes is in the passed CellAttr record. Change the values of any of the fields in the CellAttr record to change the way the cell is displayed.

The font object in the CellAttr parameter was specially created for the TOvcCellAttributes record (and is deleted after the cell is painted). Changing it does not change any of the other fonts in the table and its internal classes. Therefore, you can change the font properties singly, or you can assign another font's properties in their entirety to this font by using the Assign method. This event handler must not change any table properties that would cause the table to be repainted.

The following example ensures that all the cells on the main diagonal of the table (the row number equals the column number) are painted with a lime background.

```
procedure TForm1.OvcTable1GetCellAttributes(
  Sender : TObject; RowNum : LongInt; ColNum : Integer;
  var CellAttr : TOvcCellAttributes);
  begin
    if (Row = Col) then
      CellAttr.caColor := clLime;
  end;
```

See also: Access, Adjust, Color, Font

OnGetCellData event

```
property OnGetCellData : TCellDataNotifyEvent
```

```
TCellDataNotifyEvent = procedure(Sender : TObject;
  RowNum : TRowNum; ColNum : TColNum; var Data : Pointer;
  Purpose : TOvcCellDataPurpose) of object;
```

```
TOvcCellDataPurpose = (cdpForPaint, cdpForEdit, cdpForSave);
```

↳ Defines an event handler that is called when the table needs the data for a cell.

The method assigned to the OnGetCellData event is called just before a cell is painted, just before a cell is edited, and just after the cell is edited (when the data needs to be saved). The Purpose parameter defines each of these three occasions.

When the table repaints itself, it displays cells in row order, and within each row in reverse column order. For each cell, the table triggers the OnGetCellData event just prior to calling the Paint method of the cell component that is in charge of repainting that cell (Purpose = cdpForPaint). The event handler must return a pointer to the data in the Data parameter. The cell address is given by the RowNum and ColNum parameters. The table's painting code passes the returned Data pointer directly to the cell component's Paint method.

When the user edits a cell, the table triggers an OnGetCellData event just prior to calling the StartEditing method of the cell component that is in charge of editing the cell (Purpose = cdpForEdit). The event handler returns a pointer to the data and the cell address is given by RowNum and ColNum.

When the user finishes editing a cell, the table triggers an OnGetCellData event just prior to calling the StopEditing method of the cell component that is in charge of editing the cell (Purpose = cdpForSave). The event handler returns a pointer to a variable where the newly edited data can be saved. The cell address is given by RowNum and ColNum.

The cell component for the cell defines the type of data that is displayed in that cell. The returned Data parameter must point to a variable of the correct type for the cell component; no type checking occurs. For example, a TOvcTCString cell component requires a pointer to a string. If you pass a pointer to anything else, you will get some weird effects (since the cell component assumes that the pointer points to a string).

The Purpose parameter enables you to provide a different variable address for Data at the three different times OnGetCellData can be called. For example, when saving you might want to provide the address of a temporary variable where the data is to be saved, and then actually save the data to your 'database' during an OnDoneEdit method.

If you do not provide an OnGetCellData event handler, no data will ever be displayed in the table at run time. All cells will be blank. You must provide an OnGetCellData event handler for every table.

This event handler must not change any table properties that would cause the table to be repainted.

For an example of the OnGetCellData event, see the example on page 1070.

OnLeavingColumn event

```
property OnLeavingColumn : TColNotifyEvent
```

```
TColNotifyEvent = procedure(
  Sender : TObject; ColNum : TColNum) of object;
```

Defines an event handler that is called just before the active column changes.

The method assigned to the OnLeavingColumn event is called just before the column number for the active cell changes. This gives you an opportunity to save data from the cache to disk or to send a message to say the active cell changed (if the current active cell was edited).

In the default table model, rows are records and columns are fields and it is assumed that entire records are being processed at once. Therefore, the OnLeavingColumn event is not that important (presumably changes will be saved when the active row changes, not every time the active column changes within the row).

When the active column is changed, the OnLeavingColumn event is triggered for the old column number before the OnEnteringColumn event is triggered for the new column number.

This event handler must not change any table properties that would cause the table to be repainted or change the active cell.

See also: OnEnteringColumn, OnLeavingRow

```
property OnLeavingRow : TRowNotifyEvent
```

```
TRowNotifyEvent = procedure(
  Sender : TObject; RowNum : TRowNum) of object;
```

Defines an event handler that is called just before the active row changes.

The method assigned to the OnLeavingRow event is called just before the row number for the active cell changes. This gives you an opportunity to save data from the cache to disk or send a message to say the active cell changed (if the current active cell was edited).

In the default table model, rows are records and columns are fields and it is assumed that entire records are being processed at once. Therefore, the OnLeavingRow event gives you an opportunity to save the record data back to disk.

When the active row is changed, the OnLeavingRow event is triggered for the old row number before the OnEnteringRow event is triggered for the new row number.

The event handler must not change any table properties that would cause the table to be repainted or change the active cell.

The following example uses the OnLeavingRow event handler to save the record to a flat file. F is assumed to be a file of some record type, and DatabaseRecord is a global instance of that record type. The call to the Seek procedure moves the file pointer to the required record. If you have 1 locked row, then you need to read record 0 when the event is triggered for row 1. You don't need to read a record for row 0, the locked row, because it is presumed to be column headings, for example. RowHasChanged is a routine that you must write to determine whether the record in row RowNum has changed. You could save a copy of the original version of the record in the OnEnteringRow event and compare the saved version to the one in Database Record.

```
procedure TForm1.OvcTable1LeavingRow
  (Sender : TObject; var RowNum : LongInt);
begin
 if (RowNum >= OvcTable1.LockedRows) and RowHasChanged then begin
   Seek(F, RowNum - OvcTable1.LockedRows);
   Write(F, DatabaseRecord);
 end;
end;
```

See also: OnEnteringRow, OnLeavingColumn

OnLockedCellClick event

```
property OnLockedCellClick : TCellNotifyEvent
```

```
TCellNotifyEvent = procedure(Sender : TObject;
  RowNum : TRowNum; ColNum : TColNum) of object;
```

Defines an event handler that is called when a locked cell is clicked.

One use for this event is to change the active cell so that the column matches the ColNum parameter or so that the row matches the RowNum parameter. This would mean that clicking on the column heading for column 5 would move the active cell to column 5.

The following example changes the color of the cells in a column when the locked cell at the top of the column is clicked. The locked columns are ignored. The color change is only applied if the default cell for the column is descended from TOvcTCBaseString.

```
             procedure TForm1.OvcTable1LockedCellClick
    (Sender : TObject; RowNum : LongInt; ColNum : Integer);
  var
    CellComponent : TOvcBaseTableCell;
  begin
    with OvcTable1 do begin
      if (ColNum >= LockedCols) then begin
        AllowRedraw := false;
        CellComponent := Columns.DefaultCell[ColNum];
        if (CellComponent is TOvcTCBaseString) then
          with TOvcTCString(CellComponent) do
            if (Color = clLime) then
              Color := clSilver
            else
              Color := clLime;
        InvalidateColumn(ColNum);
        AllowRedraw := True;
      end;
    end;
  end;
```

OnPaintUnusedArea event

```
property OnPaintUnusedArea : TNotifyEvent
```

Defines an event handler that is called when the table's unused area needs to be painted.

If no method is assigned to the OnPaintUnusedArea event, the default action is to fill the unused area with the background color for the table (the value of the Color property). If a method is assigned to the OnPaintUnusedArea event, it is responsible for painting the unused area. The default action will not occur.

The unused part of the table consists of any area to the right of the last column and below the last row. At any one time it is either not visible at all, only visible as a rectangle to the right of the displayed cells, only visible as a rectangle under the displayed cells, or visible both to the right and under the displayed cells (a mirror image L). By calling GetDisplayedColNums, GetDisplayedRowNums, ColOffset, and RowOffset you can determine the extent of the unused area.

Your event handler can display anything within the unused area. For example, you could hatch it or tile it with a bitmap (e.g., a logo).

TNotifyEvent is defined in the VCL's Classes unit.

OnRowsChanged event

```
property OnRowsChanged : TRowChangeNotifyEvent
```

```
TRowChangeNotifyEvent = procedure (
  Sender : TObject; RowNum1, RowNum2 : TRowNum;
  Action : TOvcTblActions) of object;
```

```
TOvcTblActions = (taGeneral, taSingle,
  taAll, taInsert, taDelete, taExchange);
```

↳ Defines an event handler that is called when the user changes the rows of the table.

The method assigned to the OnRowsChanged event is called when the user changes the configuration of the rows in the table by inserting a row, deleting a row, or swapping two rows. Action defines what happened. If Action equals taInsert or taDelete, the row number is in RowNum1. If Action equals taExchange, the two rows are in RowNum1 and RowNum2. No other TOvcTblActions values are used.

This event gives you an opportunity to change your data structure (or to update a mapping) to reflect the insertion, deletion, or exchange. See the EXTBL02U example program for an example of using this event.

OnSizeCellEditor — event

```
property OnSizeCellEditor : TSizeCellEditorNotifyEvent
```

```
TSizeCellEditorNotifyEvent = procedure (Sender : TObject;
  RowNum : TRowNum; ColNum : TColNum; var CellRect : TRect;
  var CellStyle : TOvcTblEditorStyle) of object;
```

```
TOvcTblEditorStyle = (tesNormal, tesBorder, tes3D);
```

↳ Defines an event handler that is called before a cell is edited, to define the size of the window for the cell editor.

This event is triggered before the table displays an editor for the active cell. It is called with the address of the active cell in RowNum and ColNum. CellRect is the position and size of the active cell. You can modify the rectangle's value to make the editor a different size or in a different position. Use CellStyle to choose the border style for the cell editor (no border at all, a single line black border, or a 3D-look border). The border style you specify is used if possible. The ability to display a border is a function of the underlying control. For example, a combo box does not have a BorderStyle property at all and so will ignore a CellStyle of tesBorder.

The coordinates in CellRect are relative to the table's window. The cell editor is created as a child window of the table. Therefore you must ensure that the returned CellRect defines a rectangle within the visible part of the table (otherwise the editor will not be visible). OnSizeCellEditor is also called when the table is scrolled in editing mode.

See the EXTBL02U example project for an example of using this event.

OnTopLeftCellChanged — event

```
property OnTopLeftCellChanged : TCellNotifyEvent
```

```
TCellNotifyEvent = procedure(Sender : TObject;
  RowNum : TRowNum; ColNum : TColNum) of object;
```

↳ Defines an event handler that is called when the top left cell of the table changes.

The method assigned to the OnTopLeftCellChanged event is called with the new top column and row values. For example, this event gives you an opportunity to update a status line.

The top left cell is defined as the cell at the top left hand corner of the normal area of the table. In other words, it is the cell just below any locked rows and just to the right of any locked columns.

The event handler must not change any table properties that would cause the table to be repainted or change the active or top left cell.

The following example displays the top left cell in a label control in the form Rn:Cm where n is the row number and m the column number.

```
  procedure TForm1.OvcTable1TopLeftCellChanged(
    Sender : TObject; RowNum : LongInt; ColNum : Integer);
  begin
    with Label2 do begin
      Caption := Format('R%d:C%d', [RowNum, ColNum]);
      Update;
    end;
  end;
```

See also: OnActiveCellMoving, OnTopLeftCellChanging

OnTopLeftCellChanging event

```
property OnTopLeftCellChanging : TCellChangeNotifyEvent
```

```
TCellChangeNotifyEvent = procedure(Sender : TObject;
  var RowNum : TRowNum; var ColNum : TColNum) of object;
```

Defines an event handler that is called just before the top left cell is changed.

The method assigned to the OnTopLeftCellChanging event is triggered just before the TopRow and/or LeftCol properties are to be changed. The new values of these two properties have been calculated and are passed as the RowNum and ColNum parameters respectively. You have the opportunity (as they are var parameters) to change the values of the new top row and left column. Do NOT change TopRow or LeftCol in your event handler (this will result in an infinite loop), instead change the values of RowNum and ColNum.

Note that the top left cell is generally changed to ensure that the active cell is visible. It is possible to change the values of RowNum and ColNum so that the active cell is no longer visible: this can be disconcerting to the user, and the table does NOT test for or correct this particular condition (as it has already tested for it when it calculated the new address for the top left cell).

If you alter the values of RowNum and ColNum, be aware that the table will ensure that the returned values are within the main part of the table and that the cell they reference is not hidden.

Note also that the table tries to minimize the amount of unused area that is visible and hence your selection for the address of the top left cell may be rejected by the table. It will however try and set the top left cell to the nearest cell and still have the table display satisfy the above condition.

The following example shows a perverse example of an OnTopLeftCellChanging event handler: the top left cell is always forced to row 1, column 1 (assuming that there is one locked row and one locked column).

```
procedure TForm1.OvcTable1TopLeftCellChanging(
 Sender : TObject; var RowNum : LongInt; var ColNum : Integer);
begin
  RowNum := OvcTable1.LockedRows;
  ColNum := OvcTable1.LockedCols;
end;
```

See also: OnActiveCellMoving, OnTopLeftCellChanged

OnUserCommand event

```
property OnUserCommand : TUserCommandEvent
```

```
TUserCommandEvent = procedure(
  Sender : TObject; Command : Word) of object;
```

↳ Defines an event handler that is called when a user-defined command is entered.

The method assigned to the OnUserCommand event is called when any user-defined commands that are defined for the table are detected.

This event handler can make any desired changes to the table (e.g., change the top left cell, change the active cell, insert or delete columns or rows).

30

```
property Options : TOvcTblOptionSet
```

```
TOvcTblOption = (otoBrowseRow, otoNoRowResizing,
  otoNoColResizing, otoTabToArrow, otoEnterToArrow,
  otoAlwaysEditing, otoNoSelection);
```

```
TOvcTblOptionSet = set of TOvcTblOption;
```

Default: []

Is a set of flags that determine the behavior of the table.

The possible values for Options are:

Value	Description
otoBrowseRow	The entire active row is highlighted (and selected) when viewing the table.
otoNoRowResizing	Rows cannot be resized at run time.
otoNoColResizing	Columns cannot be resized at run time.
otoTabToArrow	<Tab> is equivalent to the right arrow and <ShiftTab> is equivalent to the left arrow.
otoEnterToArrow	<Enter> is equivalent to the right arrow.
otoAlwaysEditing	The active cell is always forced to edit mode.
otoNoSelection	Cells cannot be selected at run time with the mouse or keyboard.
otoMouseDragSelect	Cells can be selected with a click+drag (or Ctrl+click+drag) operation.
otoRowSelection	Rows can be selected by clicking on the locked cell(s) at the left of a row.
otoColSelection	Columns can be selected by clicking on the locked cell(s) at the top of a column.
otoThumbTrack	The table scrolls at the same time as the scrollbar thumb is moved.
otoAllowColMoves	Allow columns to be moved by click+drag on the locked cell(s) at the top of a column.
otoAllowRowMoves	Allow rows to be moved by click+drag on the locked cell(s) at the left of a row.

Please note that some of these options are mutually exclusive. If otoAlwaysEditing is on, otoNoRowResizing, otoNoColResizing, and otoNoSelection are forced on and otoAllowColMoves and otoAllowRowMoves are forced off. If otoNoSelection is on, otoMouseDragSelect, otoRowSelection, and otoColSelection are all forced off. If otoRowSelection is on, otoAllowRowMoves is forced off. If otoColSelection is on, otoAllowColMoves is forced off.

ProcessScrollBarClick virtual method

```
procedure ProcessScrollBarClick(
  ScrollBar : TOvcScrollBar; ScrollCode : TScrollCode);
```

```
TOvcScrollBar = (otsbVertical, otsbHorizontal);
```

↳ Processes a click on one of the scrollbars.

The ScrollBar parameter identifies which scrollbar was clicked (vertical or horizontal) and the ScrollCode parameter identifies the region of the scrollbar. The TScrollCode type is defined in the VCL documentation.

The actions taken by each combination of ScrollBar and ScrollCode are described in the following table. Any combination that is not in the table is ignored (for example, the scEndScroll scroll code is ignored in any combination). The active cell is not changed by any of these actions.

Combination	**Action**
otsbHorizontal, scLineUp	Scroll the table right one cell. This action is performed when the left arrow on the horizontal scrollbar is clicked. It scrolls the table right one cell, so that the left-most column becomes the second column from the left.
otsbHorizontal, scLineDown	Scroll the table left one cell. This action is performed when the right arrow on the horizontal scrollbar is clicked. It scrolls the table left one cell, so that the second leftmost column becomes the leftmost column. If there are no new columns to display, the unused area gets larger.
otsbHorizontal, scPageUp	Scroll the table right one page. This action is performed when the left paging area on the horizontal scrollbar is clicked. It scrolls the table right one page, so that the leftmost column becomes the rightmost column (if possible).
otsbHorizontal, scPageDown	Scroll the table left one page. This action is performed when the right paging area on the horizontal scrollbar is clicked. It scrolls the table left one page, so that the rightmost column becomes the leftmost column. If there are no new columns to display, the unused area gets larger.
otsbVertical, scLineUp	Scroll the table down one cell. This action is performed when the up arrow on the vertical scrollbar is clicked. It scrolls the table down one cell, so that the top row becomes the second row from the top.

Combination	Action
otsbHorizontal, scLineUp	Scroll the table right one cell. This action is performed when the left arrow on the horizontal scrollbar is clicked. It scrolls the table right one cell, so that the left-most column becomes the second column from the left.
otsbVertical, scLineDown	Scroll the table up one cell. This action is performed when the down arrow on the vertical scrollbar is clicked. It scrolls the table up one cell, to show a new row (or rows) at the bottom of the table. If there are no new rows, the unused area gets larger.
otsbVertical, scPageUp	Scroll the table down one page. This action is performed when the upper paging area on the vertical scrollbar is clicked. It scrolls the table down one page, so that the top row becomes the lowest row (if possible).
otsbVertical, scPageDown	Scroll the table up one page. This action is performed when the lower paging area on the vertical scrollbar is clicked. It scrolls the table up one page, so that the lowest row becomes the topmost row. If there are no new rows to display, the unused area gets larger.

ResolveCellAttributes virtual method

```
procedure
  ResolveCellAttributes(RowNum : TRowNum;
  ColNum : TColNum; var CellAttr : TOvcCellAttributes);
```

```
TOvcCellAttributes = record
  caAccess      : TOvcTblAccess;   {access rights}
  caAdjust      : TOvcTblAdjust;   {data adjustment}
  caColor       : TColor;          {background color}
  caFont        : TFont;           {text font}
  caFontColor   : TColor;          {text color}
  caFontHiColor : TColor;          {text highlight color}
  caTextStyle   : TOvcTextStyle;   {text style}
end;
```

Calculates the display attributes of the specified cell.

This method is called from the cell component as part of its ResolveAttributes method. When a cell needs to be painted (or edited), the table zeroes out a TOvcCellAttributes record structure and then calls the cell component's ResolveAttributes method. ResolveAttributes sets up the default values for the cell component in the TOvcCellAttributes record and then calls ResolveCellAttributes. The ResolveCellAttributes method calculates any special attributes for the cell (for example, if it is locked, selected, active, or has any special attributes set for it through the BlockXxx properties) and changes the record accordingly. Just before returning, it triggers an OnGetCellAttributes event, passing the record. ResolveCellAttributes should never need to be overridden, but is documented here just in case you need to change the default processing.

See also: OnGetCellAttributes

RowLimit — property

```
property RowLimit : TRowNum
```

Default: 10

↳ The number of rows allowed in the table.

You can change the value of RowLimit at any time to increase or decrease the number of rows that the table displays.

If you insert a row into the Rows property, RowLimit is incremented. If you delete a row from the Rows property, RowLimit is decremented. If any rows are defined with any special properties and you decrease RowLimit so that those rows are excluded, the special properties are not discarded. Increasing RowLimit again will reveal them.

See also: ColCount

RowOffset — read-only property

```
property RowOffset[RowNum : TRowNum] : Integer
```

↳ An array of the starting offsets of the rows.

The RowOffset array enables you to find the starting offset of each row within the client area of the table. The offset is measured in pixels from the top edge of the client area (the first displayed row in the table has an offset of 0). If a row is not displayed when RowOffset is called, the returned value of the element is -1. To find out which rows are being displayed, use GetDisplayedRowNums.

The ending offset of each displayed row is the starting offset of the next row. The ending offset of the last row is equal to the starting offset plus the row height (use the Height array property of the Rows property).

See also: ColOffset, GetDisplayedRowNums

Rows property

```
property Rows : TOvcTableRows
```

↳ The array of rows for the table.

Rows is an instantiated object of type TOvcTableRows. See “TOvcTableRows Class” on page 903 for a description of the class.

If you set the Rows property, the current internal rows object is freed and the internal rows object is set to point to the new rows object. After the assignment there is only one copy of the row information, and so the source object cannot be disposed without affecting the table.

See also: Columns

SaveEditedData method

```
function SaveEditedData : Boolean;
```

↳ Saves the data being edited in the active cell.

If the active cell is not being edited this method returns immediately with the result True.

If the active cell is being edited, the cell component’s CanSaveEditedData method is called to see if the data can be saved (i.e., is valid). If this returns false, SaveEditedData returns false immediately. If, however, CanSaveEditedData returns True, the data is saved by calling the cell component’s SaveEditedData method. This function then returns True.

ScrollBars property

```
property ScrollBars : TScrollStyle
```

Default: ssBoth

↳ Determines whether the table displays scroll bars.

If the value of ScrollBars is set to ssNone, no scroll bars are shown at all. If it is sbVertical, only a vertical scroll bar is shown. If it is sbHorizontal, only a horizontal scroll bar is shown. If it is sbBoth, both a horizontal and vertical scroll bar are shown.

Note that when a scrollbar is enabled it will still only appear when required (for example if all the rows can be completely seen, no vertical scroll bar will be shown).

The TScrollStyle type is defined in the VCL documentation.

SetActiveCell method

```
procedure SetActiveCell(RowNum : TRowNum; ColNum : TColNum);
```

↳ Sets the active cell.

This method sets the active cell with just one call, rather than two calls to set the ActiveRow and ActiveCol properties.

See also: ActiveCol, ActiveRow

SetTopLeftCell method

```
procedure SetTopLeftCell(RowNum : TRowNum; ColNum : TColNum);
```

↳ Sets the cell at the top left corner of the main area of the table.

This method sets the cell at the top left corner with just one call, rather than two calls to set the TopRow and LeftCol properties.

See also: LeftCol, TopRow

StartEditingState method

```
function StartEditingState : Boolean;
```

↳ Starts editing the active cell.

The method returns True if the table was successfully switched into edit mode (or if it was already in edit mode). It returns False if the table could not be switched into edit mode for any reason.

Calling StartEditingState will perform several actions. If the table is already in editing mode, this method returns immediately with True. Otherwise the OnBeginEdit event is triggered. If OnBeginEdit returns True, the active cell is forced to be visible (if required), the current attributes for the active cell are calculated, and if the Access attribute so calculated is otaNormal the data pointer for the cell is obtained through the OnGetCellData event. The cell component's StartEditing method is then called to prepare the editing control for editing the cell's data. If all these operations succeed, StartEditingState will return True and the table will be in editing mode; if any step fails (e.g., the cell's Access attribute is calculated as otaReadOnly) then False is returned instead and the table remains in browse mode.

See also: InEditingState, OnBeginEdit, OnGetCellData

StopEditingState method

```
function StopEditingState(SaveValue : Boolean) : Boolean;
```

↳ Stops editing the active cell.

The method returns True if the table was switched back into browse (or non-edit mode) or if it was already in browse mode. It returns False if the table could not be switched into browse mode for any reason.

SaveValue is True if the data in the editing control is to be saved, otherwise it is false (and the edited data is discarded).

Calling StopEditingState will perform several actions. If the table is already in a browse state, this method returns immediately with True. Otherwise the CanSaveEditedData method of the active cell's underlying cell component is called. If this returns True, the OnEndEdit event is triggered. If this returns True as well, the actual editing process is terminated. Firstly the data pointer for the active cell is requested through an OnGetCellData event, and this is passed to the cell component's StopEditing method. Finally the OnDoneEdit event is triggered. If any of the steps 'fails' (e.g., CanSaveEditedData returns False), StopEditingState will return immediately with False as the function result, and the table will remain in editing mode. If every step succeeds then this method will return True and the table will be switched to browse mode.

See also: InEditingState, OnDoneEdit, OnEndEdit

TopRow property

```
property TopRow : TRowNum
```

Default: 1

↳ The number of the row at the top edge of the main table area.

The row number must be between LockedRows and RowLimit-1. If it is not, TopRow is set to LockedRows.

The table actually tries to ensure that its bottommost page is as full as possible (i.e. that the unused area shown on that page is as small as possible). This in turn means that not all rows can become the top row: if you try and set TopRow for the bottom rows in the table it calculates the last row that satisfies the above criterion, and sets TopRow to that instead.

If you specify a hidden row, the next visible row is selected instead. If there is no next visible row, the previous visible row is selected.

See also: LeftCol, SetTopLeftCell

TOvcDbTable Component

TOvcDbTable provides a grid-like component that connects to a datasource to provide viewing and editing capabilities for all database file formats supported by the VCL. The TOvcDbTable is visually similar to the TOvcTable component, and where applicable, TOvcTable's methods and properties are supported in TOvcDbTable. Such concepts as column sizing/moving and the use of column and cell components remain. Properties such as the TOvcTable's LockedRows, inserting or deleting columns, and the ability to use discrete cell components do not, however.

The TOvcDbTable consists of one or more columns that are each attached to a database field. A cell component can be attached to a table column to allow editing of the displayed field data. (For information about cell components, see "TOvcBaseTableCell Class" on page 922.) The data-aware table uses the column and cell information to display and edit each field in a one record-per-row format.

The TOvcDbTable can optionally display a header (normally, this is used to display column labels). The header row can either obtain the column display text directly from the datasource (the default), use an internal list of strings that you provide via the header cell component, or can be configured to allow you to handle all drawing in the header region (OwnerDraw).

The TOvcDbTable offers the following features:

- Table columns associate with a database fields by name
- Moveable/sizable columns, both at run time and design time
- Unlimited number of database records
- Validated data entry (provided by the cell components)
- Supports many cell types with the capability to use custom cell types of your design
- A rich set of display options
- Small overhead (table does not store the data it displays)

Basic use the TOvcDbTable component requires the following steps:

1. Place a TOvcDbTable component on the form and set the DataSource property to a TDataSource component.
2. Place cell components of the type you want to attach to the database fields on the form. (Only one of each type is required unless you want different cell behavior for different columns—one cell component can be attached to several columns.)

Warning Since a cell component can be attached to more than one table column, care must be taken to ensure that the data requirements of the columns (the database fields) are the same if one cell component will be attached to more than one column.

When a TOvcDbTable component is initially connected to a datasource, no cell components are attached to the fields in the datasource and editing of the displayed fields is not supported. In this state, the table is simply a browser, i.e., no editing capabilities. To be able to edit the data displayed by the table, cell components must be attached to the table columns. The easiest way to do this is to use the Columns Property Editor, which can be invoked from the Columns property in the Object Inspector or from the table's local menu.

3. Invoke the Columns editor and select a cell component for each column that will allow editing. Set the Hidden flag to True for columns that should not be displayed (all columns are visible by default).

Table commands

The following commands are available in the Orpheus data-aware table component. The table uses its own special command table, the Grid command table. See "TOvcCommandProcessor Class" on page 50 for information on how to customize the command tables.

Table 30.4: *Commands and their functions*

Command	**Grid command table**	**Description**
ccBotOfPage	<CtrlPgDn>	Move the active cell to the bottom of the page (the column remain unchanged).
ccBotRightCell	<CtrlDown>	Move the focus to the cell at the bottom-right of the table.
ccDown	<Down>	Move the focus to the next row.
ccEnd	<End>	Move the focus to the last column.
ccExtendDown	<ShiftDown>	Extend the selection down to the next row.
ccExtendEnd	<ShiftEnd>	Extend the selection to the last column.
ccExtendHome	<ShiftHome>	Extend the selection to the first column.
ccExtendLeft	<ShiftLeft>	Extend the selection left one column.

Table 30.4: *Commands and their functions (continued)*

Command	Grid command table	Description
ccExtendPgDn	<ShiftPgDn>	Extend the selection down one page.
ccExtendPgUp	<ShiftPgUp>	Extend the selection up one page.
ccExtendRight	<ShiftRight>	Extend the selection to the right by one column.
ccExtendUp	<ShiftUp>	Extend the selection up one row.
ccExtFirstPage	<CtrlShiftHome>	Extend the selection to the first row (the column remains unchanged).
ccExtLastPage	<CtrlShiftEnd>	Extend the selection to the last row (the column remains unchanged).
ccExtWordLeft	<CtrlShiftLeft>	Extend the selection to the left by one page.
ccExtWordRight	<CtrlShiftRight>	Extend the selection to the right by one page.
ccFirstPage	<CtrlHome>	Move the focus to the first cell in the table.
ccHome	<Home>	Move the active cell to the first column.
ccLastPage	<CtrlEnd>	Move the active cell to the last cell in the table.
ccLeft	<Left>	Move the active cell left to the previous cell.
ccNextPage	<PgDn>	Move the active cell to the page following the current page.
ccPageLeft	<CtrlLeft>	Move the active cell left a page (one window width).
ccPageRight	<CtrlRight>	Move the active cell right a page (one window width).
ccPrevPage	<PgUp>	Move the active cell to the previous page.
ccRight	<Right>	Move the active cell right to the next cell.
ccTableEdit	<F2>	Enter / exit table edit mode

Table 30.4: *Commands and their functions (continued)*

Command	**Grid command table**	**Description**
`ccTopLeftCell`	`<CtrlUp>`	Move the active cell to the top left cell in a table.
`ccTopOfPage`	`<CtrlPgUp>`	Move the active cell to the top of the page (the column remains unchanged).
`ccUp`	`<Up>`	Move the active cell up to the previous row.

Hierarchy

TCustomControl (VCL)

❶TOvcCustomControl (OvcBase) 16

❷TOvcCustomControlEx (OvcBase) 19

TOvcTableAncestor (OvcTCmmn) 1021

TOvcCustomTable (OvcTable). 1022

TOvcDbTable (OvcDbTbl)

Properties

❶	About	❷	Controller		LockedColumns
	Access		DataSource		Options
	ActiveColumn		DefaultMargin		RowHeight
	ActiveRow		FieldCount		RowIndicatorWidth
	Adjust		Fields		ScrollBars
❶	AttachedLabel		GridPenSet		SelectedField
	BorderStyle		HeaderCell		TextStyle
	Colors		HeaderHeight		VisibleColumnCount
	ColorUnused		HeaderOptions		VisibleRowCount
	ColumnCount	❶	LabelInfo		
	Columns		LeftColumn		

Methods

CalcRowCol	InvalidateHeader	SaveEditedData
FilterKey	InvalidateIndicators	Scroll
InEditingState	InvalidateRow	SetActiveCell
InvalidateCell	MakeColumnVisible	StartEditing
InvalidateColumn	MoveActiveCell	StopEditing

Events

❶	AfterEnter		OnIndicatorClick		OnPaintUnusedArea
❶	AfterExit		OnLockedCellClick		OnUnusedAreaClick
	OnActiveCellChanged	❶	OnMouseWheel		OnUserCommand

Reference Section

Access — run-time property

```
property Access : TOvcTblAccess
```

```
TOvcTblAccess = (
  otxDefault, otxNormal, otxReadOnly, otxInvisible);
```

Default: otxNormal

Determines the cell access rights.

If a cell component in the table has otxDefault as its Access property, then it will use this Access value.

The possible values for Access are:

Value	Description
otxDefault	The same as otxNormal.
otxNormal	The cell can be read and edited.
otxReadOnly	The cell can be read but not edited.
otxInvisible	The cell cannot be read or edited and is invisible (the cell is displayed as a blank rectangle in the background color).

ActiveColumn — run-time property

```
property ActiveColumn : Integer
```

Default: 0

The column number of the active cell.

The column number must be between LockedColumns and ColumnCount-1. If it is not, ActiveColumn is set to LockedColumns. If you specify a hidden column, the next visible column is selected instead. If there is no next visible column, the previous visible column is selected.

See also: ActiveRow, ColumnCount, Columns, LockedColumns, SetActiveCell

ActiveRow run-time property

```
property ActiveRow : Integer
```

Default: 0

↳ The row number of the active cell.

The row number must be between 0 and VisibleRowCount-1. If it is not, it is forced to the closest valid row number.

Adjust property

```
property Adjust : TOvcTblAdjust
```

```
TOvcTblAdjust = (otaDefault, otaTopLeft, otaTopCenter,
  otaTopRight, otaCenterLeft, otaCenter, otaCenterRight,
  otaBottomLeft, otaBottomCenter, otaBottomRight);
```

Default: otaCenterLeft

↳ Defines where data is displayed in the cell.

If a cell component in the table has otaDefault as its Adjust property, then it uses this Adjust value.

The possible values for Adjust are:

Value	Description
otaDefault	The same as otaCenterLeft.
otaTopLeft	The data is displayed in the top left corner of the cell.
otaTopCenter	The data is centered horizontally at the top of the cell.
otaTopRight	The data is displayed in the top right corner of the cell.
otaCenterLeft	The data is centered vertically at the left of the cell.
otaCenter	The data is centered vertically and horizontally in the cell.
otaCenterRight	The data is centered vertically at the right of the cell.
otaBottomLeft	The data is displayed in the bottom left corner of the cell.
otaBottomCenter	The data is centered horizontally at the bottom of the cell.
otaBottomRight	The data is displayed in the bottom right corner of the cell.

See also: Access

BorderStyle property

```
property BorderStyle : TBorderStyle
```

Default: bsSingle

Determines the style used when drawing the border.

The possible values are bsSingle (a single line border is drawn around the component) or bsNone (no border line).

The TBorderStyle type is defined in the VCL's Controls unit.

```
function CalcRowCol(X, Y : Integer;
  var RowNum, ColNum : Integer) : TOvcTblRegion;
```

```
TOvcTblRegion = (
  otrInMain, otrInLocked, otrInUnused, otrOutside);
```

Returns the cell address for an X,Y coordinate.

Given an X,Y coordinate (for example from a mouse click), this method calculates the address of the cell and the region of the table corresponding to the coordinate. The X,Y coordinate is assumed to be client relative, which means that the top left pixel of the table is assumed to be at (0,0).

The following diagram shows the regions in and around a table. A table is made up of normal cells (region 6) and locked cells (region 5). The large box is the table's client rectangle.

For an X,Y coordinate in each of the various regions, the table below shows the values for the function result and the returned row and column numbers. In this table, "row number" means that the actual row number is returned and "column number" means that the actual column number is returned. In some cases, the row number and column number are obtained by assuming that the rows and columns extend out indefinitely from the used part of the table.

Region	Function result	Row	Column
1	otrOutside	CRCFXY_RowAbove	CRCFXY_ColToLeft
2	otrOutside	CRCFXY_RowAbove	column number
3	otrOutside	CRCFXY_RowAbove	CRCFXY_ColToRight
4	otrOutside	row number	CRCFXY_ColToLeft
5	otrInLocked	row number	column number
6	otrInMain	row number	column number
7	otrInUnused	row number	CRCFXY_ColToRight
8	otrOutside	row number	CRCFXY_ColToRight
9	otrOutside	CRCFXY_RowBelow	CRCFXY_ColToLeft
10	otrInUnused	CRCFXY_RowBelow	column number
11	otrInUnused	CRCFXY_RowBelow	CRCFXY_ColToRight
12	otrOutside	CRCFXY_RowBelow	column number
13	otrOutside	CRCFXY_RowBelow	CRCFXY_ColToRight

Colors — property

```
property Colors : TOvcTableColors
```

↳ The set of colors used to paint the table cells.

See “TOvcTableColors Class” on page 882 for details on the colors used by a table.

ColorUnused — property

```
property ColorUnused : TColor
```

Default: clWindow

↳ The color, which is used to paint the unused area of the table.

See also: Colors

ColumnCount — run-time, read-only property

```
property ColumnCount : Integer
```

↳ The number of columns managed by the Columns object.

Columns — property

```
property Columns : TOvcTableColumns
```

↳ The array of columns for the table.

Columns is an instantiated object of type TOvcTableColumns. See “TOvcTableColumns Class” on page 885 for a description of this object.

DataSource — property

```
property DataSource : TDataSource
```

↳ Specifies the data source component where the entry field obtains the data to display.

DefaultMargin — property

```
property DefaultMargin : Integer
```

Default: 4

↳ Determines the column margin when no cell is assigned.

DefaultMargin controls the left indent in pixels for columns that do not have a cell component assigned.

FieldCount run-time, read-only property

```
property FieldCount : Integer
```

↳ Returns the number of associated TField objects.

See also: Fields

Fields run-time, read-only property

```
property Fields[Index : Integer] : TField
```

↳ Provides access to the TField objects associated with the table.

This indexed property provides access to the TField objects associated to the table columns. The valid range for Index is 0 to FieldCount-1.

See also: FieldCount

FilterKey method

```
function FilterKey(var Msg : TWMKey) : TOvcTblKeyNeeds;
```

```
TOvcTblKeyNeeds = (otkDontCare, otkWouldLike, otkMustHave);
```

↳ Is called from a cell to filter a keystroke.

For every keystroke entered into an editing control, its cell component calls this method of its owning table so that the table can filter the keystroke. The table returns the category of the keystroke as a value of type TOvcTblKeyNeeds.

If the return value is otkDontCare, the table is informing the cell (and its editing control) that it does not want to use this keystroke. The editing control can take any action on the key. If the return value is otkWouldLike, the table would like this keystroke, but it is leaving the decision to the cell and the editing control. Keystrokes in this category are the arrow keys and the active cell movement keys. If, for example, the editing cell does not use the <Up> keystroke, it can let the table use it instead.

If the return value is otkMustHave, the table absolutely must have the keystroke. There are only two keystrokes in this category and both are used to terminate the cell editing session. <Esc> aborts the editing and <F2> saves the new data. If the return value is otkDontCare or otkWouldLike, the cell component uses SendMessage to send a WM_KEYDOWN message containing this keystroke directly to the table.

GridPenSet property

```
property GridPenSet : TOvcGridPenSet
```

↳ The set of grid pens used to draw the grid lines.

See “TOvcGridPenSet Class” on page 891 for details about the grid lines and how they are used by the table.

HeaderCell property

```
property HeaderCell : TOvcBaseTableCell
```

Default: none

↳ The cell component used for displaying the column header.

Generally the columns in a table are different cell types. However, there is usually a header row at the top of all the columns to provide static headings. This property allows you to assign the cell component used to display the column title. TOvcTCString and TOvcTCColHead cell components can be assigned to this property. All others are rejected.

See also: HeaderOptions

HeaderHeight property

```
property HeaderHeight : Integer
```

Default: 21

↳ Determines the height (in pixels) of the header row.

If the hoShowHeader option is enabled, the header row is displayed using the height specified by HeaderHeight.

See also: HeaderCell, HeaderOptions

HeaderOptions property

```
property HeaderOptions : TOvcHeaderOptionSet
```

```
TOvcHeaderOptionSet = set of TOvcHeaderOptions;
```

```
TOvcHeaderOptions = (hoShowHeader,
  hoUseHeaderCell, hoUseLetters, hoUseStrings);
```

Default: [hoShowHeader]

Determines if the header row is displayed and how the displayed strings are obtained.

The possible values for HeaderOptions are:

Value	Description
hoShowHeader	If this option is enabled, the first row of the table represents column headings. The strings that are displayed in the header row are determined by the remaining options.
hoUseHeaderCell	The header cell assigned to the HeaderCell property handles painting of the column labels. If this option is disabled, hoUseLetters and hoUseStrings are disabled.
hoUseLetters	The header cell displays standard column lettering as the column labels. This option can be enabled only if the header cell is a TOvcTCColHead.
hoUseStrings	The header cell uses its internal list of strings as the column labels. This option can be enabled only if the header cell is a TOvcTCColHead.

See also: HeaderCell, HeaderHeight

InEditingState method

```
function InEditingState : Boolean;
```

Returns True if the active cell is being edited.

This function provides a simple test to find out if the table is currently in edit mode, i.e. that the active cell is being edited.

See also: StartEditing, StopEditing

InvalidateCell method

```
procedure InvalidateCell(RowNum, ColNum : Integer);
```

Marks the cell as requiring repainting.

See also: InvalidateColumn, InvalidateHeader, InvalidateIndicators, InvalidateRow

InvalidateColumn method

```
procedure InvalidateColumn(ColNum : Integer);
```

Marks the specified column as requiring repainting.

See also: InvalidateCell, InvalidateHeader, InvalidateIndicators, InvalidateRow

InvalidateHeader method

```
procedure InvalidateHeader;
```

Marks the header row as requiring repainting.

See also: InvalidateCell, InvalidateColumn, InvalidateIndicators, InvalidateRow

InvalidateIndicators method

```
procedure InvalidateIndicators;
```

Marks the indicator column as requiring repainting.

See also: InvalidateCell, InvalidateColumn, InvalidateHeader, InvalidateRow

InvalidateRow method

```
procedure InvalidateRow(RowNum : Integer);
```

Marks the specified row as requiring repainting.

See also: InvalidateCell, InvalidateColumn, InvalidateHeader, InvalidateIndicators

LeftColumn property

```
property LeftColumn : Integer
```

Default: 0

↳ The number of the column on the left edge of the main table area.

The column number must be between LockedColumns and ColumnCount-1. If it is not, LeftColumn is set to LockedColumns. If you specify a hidden column, the next visible column is selected instead. If there is no next visible column, the previous visible column is selected.

See also: ColumnCount, LockedColumns

LockedColumns property

```
property LockedColumns : Integer
```

Default: 0

↳ The number of fixed columns on the left side of the table.

The first LockedColumns columns (from column 0 to LockedColumns-1) are fixed; but the actual visible number of locked columns depends on whether any of these columns are hidden. If the value used to set LockedColumns is not in the range 0 to ColumnCount-1, LockedColumns is not changed.

See also: ColumnCount

MakeColumnVisible method

```
procedure MakeColumnVisible(ColNum : Integer);
```

↳ Makes the specified column within the visible portion of the table.

If the specified column is already visible, this method does nothing. If the specified column is not currently visible and does not have its Hidden property set, MakeColumnVisible alters the LeftColumn property so that the specified column appears within the visible area of the table.

See also: LeftColumn

```
procedure MoveActiveCell(Command : Word);
```

Moves the active cell to the specified position.

The active cell is moved according to the Command parameter. The following lists the commands that are processed by this method. All other commands are ignored and no exception is raised.

Command	Action
ccBotOfPage	Move the active cell to the row at the bottom of the page (the column remains unchanged). The active cell is moved to the last fully visible row on the main part of the page. If there is only one row visible on the page, this method does nothing.
ccBotRightCell	Move the active cell to the bottom right corner of the table. The active cell is moved to the last record and last visible column. The table is scrolled if necessary to make the active cell visible.
ccDown	Move the active cell down one record. The active cell is moved to the next record. If there are no more records, the method does nothing. The table is scrolled if necessary to make the active cell visible.
ccHome	Move the active cell to the first unlocked column in the current row. The active cell is usually moved to the first column after the locked columns (the column specified by LockedCols), but it is adjusted to select the first unhidden column. The table is scrolled if necessary to make the active cell visible.
ccFirstPage	Move the active cell to the first unlocked row in the current column. Works like ccHome, but applies to rows.
ccEnd	Move the active cell to the last unhidden column in the current row. The active cell is usually moved to the column specified by ColCount-1, but if this column is hidden, the column number is adjusted to select the last unhidden column. The table is scrolled if necessary to make the active cell visible.
ccLastPage	Move the active cell to the last record. Works like ccEnd, but applies to rows.
ccLeft	Move the active cell left one column. The active cell is moved to the first previous column that is unhidden. If there are no previous unhidden columns, the method does nothing. The table is scrolled if necessary to make the active cell visible.

Command	Action
ccBotOfPage	Move the active cell to the row at the bottom of the page (the column remains unchanged). The active cell is moved to the last fully visible row on the main part of the page. If there is only one row visible on the page, this method does nothing.
ccNextPage	Move the active cell down one page. The current partially (or fully) displayed record at the bottom of the page is made the top row (scrolling the table as required). If the last record is already fully shown, it is made the top row. If the top row does not change, the active cell is already at the bottom of the table, and this method will do nothing. The active cell is positioned in roughly the same position on the new page as it was on the old page. If this is impossible (for example if there are not enough rows on the new page), then the last row on the page is selected.
ccPageLeft	Move the active cell left one page. The previous page is calculated so that the column that is currently displayed at the left of the table is displayed as far right as possible and still (partially) shown. If this does not cause the table to scroll (i.e., the leftmost column of the page is the first unlocked, visible column of the table), the active column is set to the leftmost column. The active cell is positioned in roughly the same position on the new page as it was on the old page. If this is impossible (for example if there are not enough columns on the new page), then the rightmost column on the new page is selected.
ccPageRight	Move the active cell right one page. Works like ccPageLeft, but in the opposite direction.
ccPrevPage	Move the active cell up one page. Works like ccNextPage, but in the opposite direction.
ccRight	Move the active cell right one column. Works like ccLeft, but in the opposite direction.
ccTopLeftCell	Move the active cell to the top left corner of the table. The active cell is moved to the first record and the first visible unlocked column. The table is scrolled if necessary to make the active cell visible.
ccTopOfPage	Move the active cell to the record at the top of the page (the column remains unchanged). Works like ccBotOfPage, but in the opposite direction.
ccUp	Move the active cell up one record. Works like ccDown, but in the opposite direction.

OnActiveCellChanged event

```
property OnActiveCellChanged : TCellNotifyEvent
```

```
TCellNotifyEvent = procedure(Sender : TObject;
  RowNum : TRowNum; ColNum : TColNum) of object;
```

↳ Defines an event handler that is called after the active cell changes.

The method assigned to the OnActiveCellChanged event is called with the new active column and row. For example, this event gives you an opportunity to update a status line. It is a programming error to change the active cell within your event handler.

OnIndicatorClick event

```
property OnIndicatorClick : TCellNotifyEvent
```

```
TCellNotifyEvent = procedure(Sender : TObject;
  RowNum : TRowNum; ColNum : TColNum) of object;
```

↳ Defines an event handler that is called when the indicator column is clicked.

OnLockedCellClick event

```
property OnLockedCellClick : TCellNotifyEvent
```

```
TCellNotifyEvent = procedure(Sender : TObject;
  RowNum : TRowNum; ColNum : TColNum) of object;
```

↳ Defines an event handler that is called when a locked cell is clicked.

One use for this event is to change the active cell so that the column matches the ColNum parameter or so that the row matches the RowNum parameter. This would mean that clicking on the column heading for column 5 would move the active cell to column 5.

OnPaintUnusedArea event

```
property OnPaintUnusedArea : TNotifyEvent
```

↳ Defines an event handler that is called when the table's unused area needs to be painted.

If no method is assigned to the OnPaintUnusedArea event, the default action is to fill the unused area with the background color for the table (the value of the Color property). If a method is assigned to the OnPaintUnusedArea event, it is responsible for painting the unused area. The default painting will not occur.

The unused part of the table is any area to the right of the last column and below the last row. It is either not visible at all, visible as a rectangle to the right of the displayed cells, visible as a rectangle under the displayed cells, or visible both to the right and under the displayed cells (a mirror image L). Your event handler can display anything in the unused area. For example, you could hatch it or tile it with a bitmap (e.g., a logo).

TNotifyEvent is defined in the VCL's Classes unit.

OnUnusedAreaClick event

```
property OnUnusedAreaClick : TCellNotifyEvent
```

```
TCellNotifyEvent = procedure(Sender : TObject;
  RowNum : TRowNum; ColNum : TColNum) of object;
```

↳ Defines an event handler that is called when a click occurs in the unused area of the table.

OnUserCommand event

```
property OnUserCommand : TUserCommandEvent
```

```
TUserCommandEvent = procedure(
  Sender : TObject; Command : Word) of object;
```

↳ Defines an event handler that is called when a user-defined command is entered.

The method assigned to the OnUserCommand event is called when any user-defined commands that are defined for the table are detected.

This event handler can make any desired changes to the table (e.g., change the top left cell, change the active cell, insert or delete columns or rows).

```
property Options : TOvcDbTableOptionSet
```

```
TOvcDbTableOptions = (dtoCellsPaintText,
  dtoAllowColumnMove, dtoAllowColumnSize, dtoAlwaysEditing,
  dtoHighlightActiveRow, dtoIntegralHeight, dtoEnterToTab,
  dtoPageScroll, dtoShowIndicators, dtoShowPictures,
  dtoStretch, dtoWantTabs, dtoWrapAtEnds);
```

```
TOvcDbTableOptionSet = set of TOvcDbTableOptions;
```

Default: [dtoShowIndicators]

Is a set of flags that determine the behavior of the table.

The possible values for Options are:

Value	**Description**
dtoCellsPaintText	Cell components handle painting text on the table Canvas rather than the table. The cell components use the properties of the cell rather than the display string returned from the TField object.
dtoAllowColumnMove	Allow columns to be moved by clicking and dragging the header cell(s) at the top of a column.
dtoAllowColumnSize	Columns can be resized at run time.
dtoAlwaysEditing	The active cell is always in edit mode unless the cell or database field is read-only.
dtoHighlightActiveRow	The entire active row is selected when viewing the table.
dtoIntegralHeight	The table is resized so that no blank area is shown below the last visible row.
dtoEnterToTab	<Enter> is equivalent to <Tab> or the right arrow and moves the active cell one column to the right.
dtoPageScroll	If this option is disabled, the active cell moves incrementally in the direction of the scroll until the last visible row is reached. All visible cells are then scrolled. If this option is enabled, the active cell jumps by one page in the direction of the scroll before scrolling all cells. This is consistent with the scrolling behavior of VCL’s grids.
dtoShowIndicators	A column indicating which row is active is displayed at the left of the table.

Value	Description
dtoShowPictures	The table displays graphic images associated with the table column. If disabled, graphic images are displayed by the TOvcTCBitmap cell component if the cell is the active cell.
dtoStretch	The graphic image is scaled to fit the size of the table cell.
dtoWantTabs	<Tab> is equivalent to the right arrow and <ShiftTab> is equivalent to the left arrow.
dtoWrapAtEnds	Attempts to move past the last column result in the first cell of the next row being selected and attempts to move prior to the first column result in the last cell of the previous row being selected.

Some of these options are mutually exclusive. For example, if dtoAlwaysEditing is on, dtoAllowColumnMove and dtoAllowColumnSize are forced off.

RowHeight property

```
property RowHeight : Integer
```

Default: 21

Determines the height of the table rows.

The RowHeight property specifies the height (in pixels) of all table rows except the header row.

See also: HeaderHeight

RowIndicatorWidth property

```
property RowIndicatorWidth : Integer
```

Default: 11

Determines the width (in pixels) of the indicator column.

If the dtoShowIndicators option is enabled, the indicator column is displayed using this value as the width of the indicator column.

See also: Options

SaveEditedData method

```
function SaveEditedData : Boolean;
```

↳ Forces the cell to save its edited data to its associated TField object.

This method can be used to force the active cell to save its contents prior to the cell loosing the focus. The function result is True if there is no error.

Scroll method

```
procedure Scroll(Delta : Integer);
```

↳ Scrolls the datasource.

This method scrolls the datasource by Delta records (signed). If scrolling Delta records would result in a new record prior to the first or after the last record, the first or last record is made the active record and no error is generated.

See also: DataSource

ScrollBars property

```
property ScrollBars : TScrollStyle
```

Default: ssBoth

↳ Determines whether vertical and horizontal scroll bars are displayed.

The TScrollStyle enumerated type is defined in the VCL's Controls unit. Both vertical and horizontal scroll bars are displayed by default.

SelectedField run-time property

```
property SelectedField : TField
```

↳ Determines the TField object associated with the active cell.

Assigning one of the TField objects associated with the table to this property forces the corresponding cell to become the active cell.

See also: Fields

SetActiveCell method

```
procedure SetActiveCell(RowNum, ColNum : Integer);
```

Sets the active cell.

This method sets the active cell with just one call, rather than two calls to set the ActiveRow and ActiveColumn properties.

StartEditing method

```
function StartEditing : Boolean;
```

Starts editing the active cell.

StartEditing returns True if the table was successfully switched into edit mode (or if it was already in edit mode). It returns False if the table could not be switched into edit mode.

StopEditing method

```
function StopEditing(SaveValue : Boolean) : Boolean;
```

Stops editing the active cell.

StopEditing returns True if the table was switched back into browse (or non-edit mode) or if it was already in browse mode. It returns False if the table could not be switched into browse mode for any reason.

SaveValue is True if the data in the editing control is to be saved, otherwise it is False (and the edited data is discarded).

See also: InEditingState

TextStyle property

```
property TextStyle : TOvcTextStyle
```

Default: tsFlat

The appearance of the text drawn by the table component.

Text can appear in three different styles. tsFlat is the normal appearance (it looks “flat” on the screen). tsRaised makes the text appear embossed in a raised fashion. tsLowered makes the text appear embossed in a lowered fashion. For the 3D styles, the highlighted color is Colors.TextHiColor.

See also: Colors

VisibleColumnCount read-only property

```
property VisibleColumnCount : Integer
```

Returns the number columns that are currently visible within the table's client window.

See also: VisibleRowCount

VisibleRowCount read-only property

```
property VisibleRowCount : Integer
```

Returns the number rows that are currently visible within the table's client window.

See also: VisibleColumnCount

Chapter 31: Inspector Grid

This chapter deals with the InspectorGrid component. The idea behind the InspectorGrid is to provide a customized grid that emulates the Delphi Object Inspector. It is a two-column grid. The left column is read-only but it has the ability to nest items within other items. Each cell in the right-column has the ability to display an edit control with or without an attached button, or a combo box.

TO32CustomInspectorGrid Class

The TO32CustomInspectorGrid class is the direct descendant of the TO32InspectorGrid component. It provides all of the functionality and interface of the TO32InspectorGrid except it does not publish any properties.

Hierarchy

TCustomControl(VCL)

    TO32CustomControl(OvcBase) 25

        TO32CustomInspectorGrid(O32IGrid)

TO32InspectorGrid Component

Hierarchy

TCustomControl(VCL)

  TO32CustomControl(OvcBase) 25

    TO32CustomInspectorGrid(O32IGrid) 1112

      TO32InspectorGrid(O32IGrid)

Properties

	About	ChildIndentation		ItemCount
	ActiveItem	Editing		Items
	ActiveRow	ExpandGlyph	❶	LabelInfo
	Add	Font		ReadOnly
❶	AttachedLabel	Images		Selected
	AutoExpand	ItemCollection		Sorted

Methods

Add	Delete	ExpandGlyph
BeginUpdate	EndUpdate	GetItemAt
CollapseAll	ExpandAll	Insert

Events

	AfterEdit	OnCellPaint		OnExpand
❶	AfterEnter	OnCollapse		OnItemChange
❶	AfterExit	OnDelete		OnLeavingCell
	BeforeEdit	OnEnteringCell	❶	OnMouseWheel

Reference Section

About — read-only property

```
property About : string
```

Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

ActiveItem — property

```
property ActiveItem : Word
```

The item index of the currently active item.

Use ActiveItem at run time in conjunction with the indexed Items property to access the active item object.

The following example changes the caption property of the active item to the value contained in Edit1.Text:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Edit1.Text <> '' then
    O32InspectorGrid1.Items[O32InspectorGrid.ActiveItem].Caption
      := Edit1.Text;
end;
```

See also: Selected, Items

ActiveRow — property

```
property ActiveRow : Word
```

The index of the currently selected row.

Examine ActiveRow at run time to determine which row is currently being examined or edited.

```
function Add(Type : TO32IGridItemType;
  Parent : TO32InspectorItem) : TO32InspectorItem;
```

Adds a new InspectorGrid item to the list and returns a reference to the new item.

If Parent is nil then the item is added at the root level, otherwise it is added as a child of the specified parent.

Note: Children items can only belong to items of type itParent or itSet.

You may use Add or ItemCollection.Add interchangeably.

The following example uses Add to add a new list based item to the collection and sets the item index to the item corresponding to “Com 3”:

```
  procedure TForm1.Button1Click(Sender: TObject);
  var
    NewItem: TO32InspectorItem;
  begin
    NewItem := O32InspectorGrid1.Add('Port', itList);
    NewItem.StringList.Add('Com 1');
    NewItem.StringList.Add('Com 2');
    NewItem.StringList.Add('Com 3');
    NewItem.StringList.Add('Com 4');
    NewItem.ItemIndex := NewItem.StringList.IndexOf('Com 3');
  end;
```

See also: Insert, TO32InspectorItem.Add

AfterEdit — event

```
property AfterEdit : TO32IGridItemEvent
```

```
TO32IGridItemEvent = procedure(
  Sender: TObject; Item: Word) of object;
```

AfterEdit defines an event that is generated when the user finishes editing a value in an InspectorGrid item.

AutoExpand — property

```
property AutoExpand : Boolean
```

Default: False

Determines whether nested items will be automatically expanded when the grid paints itself.

Set AutoExpand to True if you want the nested items to open automatically, or to False if you want to force the user to click the +/- glyph to expand each item individually.

BeforeEdit — event

```
property BeforeEdit : TO32IGridItemEvent
```

```
To32IGridItemEvent = procedure(
  Sender: TObject; Item: Word) of object;
```

BeforeEdit defines an event that is generated immediately before the item’s edit control is enabled for editing.

BeginUpdate — method

```
procedure BeginUpdate;
```

Prevents repainting of the control during extensive update operations.

Calling BeginUpdate before executing any extensive updates to the InspectorGrid prevents screen flicker caused by rapid successive repainting of the control.

Note: Calls to BeginUpdate are cumulative. Each call to BeginUpdate must be matched with a call to EndUpdate or the control will never repaint itself.

See also: EndUpdate

ChildIndentation property

```
property ChildIndentation : Word
```

Default: 5

Determines the amount of space in pixels, that child item captions will be indented from their parents.

All items display their captions in the left column. Indenting the child items a little bit gives a visual cue to the fact that the item is a child of the less indented item.

Note: Children items can only belong to items of type itParent or itSet.

CollapseAll method

```
procedure CollapseAll;
```

Collapses all open branches.

Call CollapseAll to programmatically force all open branches to collapse.

See also: ExpandAll

Delete method

```
procedure Delete(Index: Word);
```

Deletes the item specified by Index.

You may use Delete(i) or Items[i].Delete interchangeably.

Delete will collapse the item if it is open, and destroy all child items.

See also: Add, TO32InspectorItems.Delete

Editing run-time, read-only property

```
property Editing : Boolean
```

Determines if a cell in the grid is currently being edited.

Use Editing in conjunction with ActiveItem to determine which item is being edited.

See also: ActiveItem, AfterEdit, BeforeEdit, OnEnteringCell, OnItemChange, OnLeavingCell

EndUpdate method

```
procedure EndUpdate;
```

Re-enables painting of the control after extensive update operations.

Calling BeginUpdate before executing any extensive updates to the InspectorGrid prevents screen flicker caused by rapid successive repainting of the control. Calling EndUpdate restores the control's ability to repaint itself.

Note: Calls to EndUpdate are cumulative. Each call to EndUpdate must be matched with a previous call to BeginUpdate.

See also: BeginUpdate

ExpandAll method

```
procedure ExpandAll;
```

Forces all expandable items to expand their branches.

ExpandAll is recursive. Calling ExpandAll will expand all expandable items at all levels.

See also: CollapseAll

ExpandGlyph property

```
property ExpandGlyph : TBitmap
```

Default: nil

Contains the image that is displayed to the left of the caption on items that contain children.

The ExpandGlyph is drawn on the left side of the item's caption area (the left column).

If ExpandGlyph is nil then standard + and – characters in a small white box will be drawn. ExpandGlyph defines a 3-part, nx3n, 16 color bitmap which holds the three different states of the ExpandGlyph. Each state is nxn pixels. The first and second sections represent the enabled and disabled states of the '+' image, while the third section represents the '-' image. There is no disabled state for the '-' image as the item cannot expand if it is disabled. The default ExpandGlyph that ships with Orpheus is a 10x30 bitmap with each state being represented by a 10x10 image. It is named ExpandGlyph and can be found in the Bonus directory off of the Orpheus root directory.

Font property

```
property Font : TFont
```

Contains the font that will be used by default in the InspectorGrid.

Each item in the grid will use the font specified here by default. There is a Font property in each item that can be set, which will override the font selection for that item.

See also: TO32InspectorItem.Font

GetItemAt method

```
function GetItemAt(x, y : Integer) : TO32InspectorItem;
```

GetItemAt returns the item associated with the cell located at the provided screen coordinates.

Use GetItemAt in conjunction with the mouse to gain easy access to the item under the mouse cursor.

The following example uses GetItemAt in the OnMouseDown event to change the caption of the item which was right-clicked:

```
procedure TForm1.O32InspectorGrid1MouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  Item: TO32InspectorItem;
begin
  if Button = mbRight then begin
    Item := O32InspectorGrid1.GetItemAt(x, y);
    if Item <> nil then
      Item.Caption := 'Right Clicked';
  end;
end;
```

Images property

```
property Images : TImageList
```

Default: nil

Specifies the ImageList component that contains the images used in the InspectorGrid.

Each item in the InspectorGrid can display an image in its cell. The images for these items is stored in the ImageList associated with the Images property.

Insert method

```
function Insert(Name : String; Type : TO32IGridItemType;
  Parent : TO32InspectorItem; Index : Word) : TO32InspectorItem;
```

Creates a new InspectorGridItem, inserts it in the proper place in the list, and returns a reference to the new item.

Use Insert instead of Add if you want to specify where the item is inserted in the list. If Parent is nil then the item is inserted at the root level, otherwise, it is inserted as a child of the specified parent item.

Note: Child items can only belong to items of type itParent or itSet.

You may use Insert or ItemCollection.Insert interchangeably.

Note: If Sorted is True, then their placement in the list is determined by the item's caption and not their index.

See also: Add, ItemCollection.Insert

ItemCollection property

```
property ItemCollection : TO32InspectorGridItems
```

Provides access to the InspectorGrid's item collection.

See also: Items

ItemCount run-time, read-only property

```
property ItemCount : Word
```

Returns the number of items in the InspectorGrid.

ItemCount returns the number of all items in the list, including all child items. ItemCount is not the same as using Items.Count. Items.Count will return the number of level 1 items but does not take into account the number of child items that may or may not be contained in each parent item.

Note: ItemCount does not return the total number of possible rows as some items such as the itSet type, are expandable items that contain a separate row for each member of the set. Children items can only belong to items of type itParent or itSet.

Items property

```
property Items[Index : Integer] : TO32InspectorItem
```

↳ Is an indexed property containing the items in the TO32InspectorGrid.

Use Items at run time to access a particular item by its item index or to enumerate the items. The following example changes the caption of the second item in the list:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  O32InspectorGrid1.Items[1].Caption := 'Address';
end;
```

See also: ItemCollection, ItemCount

OnItemChange event

```
property OnItemChange : TNotifyEvent
```

↳ Defines an event handler that is generated when an item's contents are changed.

Define an OnItemChanged event handler and use the Selected property to gain access to the item which has changed.

OnCellPaint event

```
property OnCellPaint : TO32IGridCellPaintEvent
```

```
TO32IGridCellPaintEvent = procedure(
  Sender : TObject; Rect : TRect; Canvas : TCanvas; Item : Word;
  Cell : TO32IGridCell; var Default : Boolean) of object;
```

↳ Defines an event handler that is generated when the individual cells are painted.

Create an OnCellPaint event to override painting of individual cells. Examine the contents of Item and Cell to determine whether you want to handle painting of the individual cell. Set Default to True to handle painting yourself or False to allow the cell to paint itself.

OnDelete event

```
property OnDelete : TO32IGridDeleteItemEvent
```

```
To32IGridDeleteItemEvent = procedure(
  Sender : TObject; Item : Word; var AllowIt : Boolean) of object;
```

↳ Defines an event handler that is generated when an item is deleted.

Examine the value of Item to determine which item is being deleted. Set the value of Allow it to False to prevent the item from being deleted.

OnEnteringCell event

```
property OnEnteringCell :  TO32IGridCellEvent
```

```
TO32IGridCellEvent = procedure(
  Sender : TObject; Item; Word; Cell : TO32IGridCell) of object;
```

```
TO32IGridCell = (icLeft, icRight);
```

↳ Defines an event handler that is generated when a cell is entered.

Item contains the index of the item being entered. Cell indicates which cell is being entered.

OnExpand event

```
property OnExpand : TO32IGridItemEvent
```

```
TO32IGridGridItemEvent = procedure(
  Sender : TObject; Item : Word) of object;
```

↳ OnExpand Defines an event handler that is generated when an InspectorItem with children is expanded.

Examine the value of Item to determine which item is being expanded. Item contains the absolute index of the item being expanded.

See also: TO32InspectorItem.AbsoluteIndex

OnLeavingCell event

```
property OnLeavingCell : TO32IGridCellEvent
```

```
TO32IGridCellEvent = procedure(
  Sender : TObject; Item; Word; Cell : TO32IGridCell) of object;
```

```
TO32IGridCell = (icLeft, icRight);
```

↳ Defines an event handler that is generated when a cell is exited.

Item contains the index of the item being exited. Cell indicates which cell is being exited.

ReadOnly property

```
property ReadOnly : Boolean
```

Default: False

↳ Sets or disabled the InspectorGrid’s ReadOnly flag.

Set ReadOnly to prevent user’s from being able to edit the contents of the grid.

Selected property

```
property Selected : TO32InspectorItem
```

↳ Provides direct access to the currently selected item.

This is a shortcut to using the Items property in conjunction with ActiveItem.

See also: ActiveItem, Items

Sorted property

```
property Sorted : Boolean
```

Default: True

↳ Determines whether the grid will automatically sort items alphabetically by their caption.

If Sorted is True the InspectorGrid will sort items alphabetically by their caption. If Sorted is False then the InspectorGrid will display items by the order in which they were added.

The InspectorGrid will always sort its items. Sorted simply tells it whether to use the Item's captions or their Index.

TO32InspectorItems Class

The TO32InspectorItems class maintains the TO32InspectorGrid's list of items.

Hierarchy

TCollection(VCL)

❶ TO32Collection(OvcBase) 45

TO32InspectorItems(O32IGrid)

Properties

About	❶ ItemByName	InspectorGrid
❶ Count	❶ ItemClass	❶ Owner
❶ Item	Items	❶ ReadOnly

Methods

❶ Add	LoadFromFile	LoadFromStream
Delete	❶ ParentForm	SaveToStream
Insert	SaveToFile	

Events

❶ OnChanged

Reference Section

About — read-only property

```
property About : string
```

↳ Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

Add — method

```
function Add(Name : String; Type : TO32IGridItemType;
  Parent : TO32InspectorItem) : TO32InspectorItem;
```

↳ Adds a new InspectorGrid item to the list and returns a reference to the new item.

If Parent is nil then the item is added at the root level; otherwise, it is added as a child of the specified parent.

Note: Children items can only belong to items of type itParent or itSet.

You may use Add or TO32InspectorGrid.Add interchangeably.

The following example uses Add to add a new list based item to the collection and sets the item index to the item corresponding to “Com 3”:

```
  procedure TForm1.Button1Click(Sender: TObject);
  var
    NewItem: TO32InspectorItem;
  begin
    NewItem := O32InspectorGrid1.ItemCollection.Add('Port',
      itList);
    NewItem.StringList.Add('Com 1');
    NewItem.StringList.Add('Com 2');
    NewItem.StringList.Add('Com 3');
    NewItem.StringList.Add('Com 4');
    NewItem.ItemIndex := NewItem.StringList.IndexOf('Com 3');
  end;
```

See also: Insert, TO32InspectorGrid.Add

Delete — method

```
procedure Delete(Index : Word);
```

Deletes the item specified by Index.

You may use Delete(i) or Items[i].Delete interchangeably.

Delete will collapse the item if it is open, and destroy all child items.

See also: Add, TO32InspectorGrid.Delete

Insert — method

```
function Insert(Name : String;
  Type : TO32IGridItemType; Parent : TO32InspectorItem;
  Index : Word) : TO32InspectorItem;
```

Creates a new InspectorGridItem, inserts it in the proper place in the list, and returns a reference to the new item.

Use Insert instead of Add if you want to specify where the item is inserted in the list. If Parent is nil then the item is inserted at the root level, otherwise, it is inserted as a child of the specified parent item.

Note: Children items can only belong to items of type itParent or itSet.

You may use Insert or TO32InspectorGrid.Insert interchangeably.

Note: If Sorted is True, then their placement in the list is determined by the item's caption and not their index.

See also: Add, TO32InspectorGrid.Insert

InspectorGrid — run-time, read-only property

```
property InspectorGrid : TO32CustomInspectorGrid
```

Returns a reference to the TO32InspectorGrid that owns the collection.

InspectorGrid is used internally but is surfaced as public in case you need to acces the owner through its items collection.

Items property

```
property Items[Index : Integer] : TO32InspectorItem
```

↳ An indexed property containing the items in the collection.

Use Items at run time to access a particular item by its item index or to enumerate the items. You may use Items or the TO32InspectorGrid.Items property interchangeably.

The following example changes the caption of the second item in the list:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  O32InspectorGrid1.ItemCollection.Items[1].Caption
    := 'Address';
end;
```

See also: TO32InspectorGrid.Items, Count

LoadFromFile method

```
procedure LoadFromFile(FileName : String);
```

↳ Loads the InspectorGrid's ItemCollection from the file specified by FileName.

LoadFromStream method

```
procedure LoadFromStream(Stream : TStream);
```

↳ Loads the InspectorGrid's ItemCollection from the specified stream.

SaveToFile method

```
procedure SaveToFile(FileName : String);
```

↳ Saves the InspectorGrid's ItemCollection to the file specified by FileName.

See also: LoadFromFile

SaveToStream method

```
procedure SaveToStream(Stream : TStream);
```

↳ Saves the InspectorGrid's ItemCollection to the specified stream.

TO32InspectorItem Class

The TO32InspectorItem class models the items contained in the TO32InspectorGrid. The items can be of the following types:

Table 31.1: *TO32InspectorItem class item types*

Type	Description
itParent	Item can server as a parent to other items for the purpose of nesting like items together. This item is like an itString item except its edit cell is read-only and it has children.
itSet	Item maintains a set of itString items. Each of the itString items is maintained as a child. This item's edit cell is read-only and the item has a child item for each member of the set. The child items are of type itLogical and determine whether the individual element is included or excluded in the set.
itList	Item maintains a mutually exclusive list of strings which are selectable from a combo box type control.
itString	Item contains standard string type data. The string can be displayed or edited in the associated cell.
itInteger	Item contains an integer based value. The number can be displayed or edited in the associated cell.
itFloat	Item contains a floating point based value. The number can be displayed or edited in the associated cell.
itCurrency	Item contains a currency based value. The number can be displayed or edited in the associated cell.
itDate	Item contains the date portion of a TDateTime based value. The date can be edited directly in the associated cell, or via a pop-up calendar.

Table 31.1: *TO32InspectorItem class item types*

Type	Description
itColor	Item contains a TColor based value. The color displayed or edited in the associated cell, or via a pop-up color dialog. The cell contains a small square on the left side that is filled with the selected color.
itLogical	Item contains a boolean based value. The value is displayed in a combo box type edit cell. It can be toggled by double-clicking on the edit cell, or by selecting “true” or “false” from the associated drop-down list.
itFont	Item contains a TFont value. The value can be edited directly or via an associated pop-up Font dialog.

Hierarchy

TCollectionItem(VCL)

❶ TO32CollectionItem(OvcBase). 45

TO32InspectorItem(O32IGrid)

Properties

About	Editing	Level
AbsoluteIndex	ImageIndexDefault	❶ Name
Child	ImageIndexOverlay	Owner
ChildCount	ImageIndexSelected	Parent
Children	InspectorGrid	ReadOnly
Deleting	IsExpanded	Value
❶ DisplayText	IsFocused	Visible

Reference Section

About — read-only property

```
property About : string
```

Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

AbsoluteIndex — property

```
property AbsoluteIndex : Word
```

Returns the absolute index of the item.

Absolute index refers to the item's placement in regard to all other items contained by the InspectorGrid. For instance if the InspectorGrid has 2 main items and the first item is of type itParent and has 3 children, the second item's Index will be 1 (Item indexes are zero-based) and its AbsoluteIndex will be 4.

Child — property

```
property Child[Index : Integer] : TO32InspectorItem
```

Is an indexed property containing the children owned by this item.

Use Child at run time to access a child by its item index or to enumerate this item's children.

See also: Children, ChildCount

ChildCount — property

```
property ChildCount : Word
```

↳ Returns the number of children that the item owns.

You may use ChildCount and Children.Count interchangeably.

Note: Only items of type itParent or itSet can have children.

Children — property

```
property Children : TO32InspectorGridItems
```

↳ Provides access to the current item's collection of children.

See also: Child, ChildCount

Deleting — run-time, read-only property

```
property Deleting : Boolean
```

↳ Determines whether the item is in the process of being deleted.

Editing — run-time, read-only property

```
property Editing : Boolean
```

↳ Determines whether the item is currently being edited by the user.

See also: IsFocused

ImageIndexDefault — property

```
property ImageIndexDefault : Integer
```

Default: -1

↳ Contains the index of the image in the TO32InspectorGrid's ImageList that is used as the default image.

See also: ImageIndexSelected, ImageIndexOverlay

ImageIndexOverlay property

```
property ImageIndexOverlay : Integer
```

Default: -1

Contains the index of the image in the TO32InspectorGrid's ImageList that is used for as an image overlay.

See also: ImageIndexSelected, ImageIndexDefault

ImageIndexSelected property

```
property ImageIndexSelected : Integer
```

Default: -1

Contains the index of the image in the TO32InspectorGrid's ImageList that is used when the item is selected.

See also: ImageIndexOverlay, ImageIndexDefault

InspectorGrid run-time, read-only property

```
property InspectorGrid : TO32CustomInspectorGrid
```

Returns a reference to the TO32InspectorGrid that owns the item.

InspectorGrid is used internally but is surfaced as public in case you need to access the InspectorGrid through any of its items.

IsExpanded run-time, read-only property

```
property IsExpanded : Boolean
```

Determines whether the item is expanded.

Only items of type itParent or itSet can have children. As such, only those item types can be expanded.

IsFocused run-time, read-only property

```
property IsFocused : Boolean
```

Determines whether the item is currently focused.

See also: Editing,

Level — property

```
property Level : Word
```

↳ Returns the item's level.

Items at the root level are at level 1. 1st level children are at level 2, etc.

Owner — property

```
property Owner : TO32CustomInspectorItems
```

↳ Returns a reference to the TO32InspectorItems collection that owns the item.

Owner is used internally but is surfaced as public in case you need to access the owner through its items collection.

Parent — run-time, read-only property

```
property Parent : TO32InspectorItem
```

↳ Returns a reference to the item's parent.

Note: Only items of type itParent or itSet can have children.

ReadOnly — property

```
property ReadOnly : Boolean
```

Default: False

↳ Determines whether this item can be edited at run time.

Items of type itParent and itSet are ReadOnly by default and the ReadOnly property cannot be changed for these types. Other item types can be toggled from ReadOnly to editable.

Value — property

```
property Value : Variant
```

↳ Read Value to access the item's value.

Visible — property

```
property Visible : Boolean
```

↳ Determines whether the item is visible in the InspectorGrid.

Set Visible to True to enable the item to be displayed in the grid, and False to disable display.

31

Chapter 32: Miscellaneous Components and Classes

This chapter describes the Orpheus components that don't fit neatly into any other category and several classes used by other Orpheus components.

The following components are described:

- TOvcDbSearchEdit provides a database search mechanism similar to the search capability in the Windows online help Index.
- TOvcDbIndexSelect provides a drop-down list of the indexes available in an attached dataset.

The following classes are described:

- TOvcEdPopup is a base edit class for use as an ancestor for edit fields that have the capability to pop-up a window or some other control (such as a pop-up calendar.)
- TOvcCaretType encapsulates an editing caret and the methods necessary to manipulate it.
- TOvcColors encapsulates both a text and a background color.
- TOvcFixedFont is equivalent to the normal TFont class except that it limits the selectable font to fixed pitch only.
- TOvcUserData provides substitution character and user-defined character sets for use by entry fields and components that use entry fields.

TOvcBaseISE Class

The TOvcBaseISE class is an abstract class derived from TCustomEdit. It provides the core functions for the higher-level components that implement incremental search capabilities. Functions provided by this class implement text handling and searching facilities used by its descendants.

Hierarchy

TCustomEdit (VCL)

  TOvcBaseISE (OvcISEB)

Properties

AutoSearch	KeyDelay
CaseSensitive	ShowResults

Reference Section

AutoSearch — property

```
property AutoSearch : Boolean
```

↳ Determines whether the search is performed automatically.

If AutoSearch is True and a search has not been done at the time the search edit loses the focus, one is performed automatically. If AutoSearch is False, you must call PerformSearch to do the search.

CaseSensitive — property

```
property CaseSensitive : Boolean
```

↳ Determines whether the search is case sensitive.

KeyDelay — property

```
property KeyDelay : Integer
```

Default: 500

↳ The number of milliseconds to wait after keyboard entry before performing the database search.

This property is used only if AutoSearch is True.

See also: AutoSearch

PerformSearch — abstract method

```
procedure PerformSearch; abstract;
```

↳ An abstract method for use by descendants to implement the database search logic.

Descendants must override this method and implement search logic for the contents of the Text property.

See also: AutoSearch

ShowResults property

```
property ShowResults : Boolean
```

Default: True

Determines whether the edit display is updated with the result of the search.

If ShowResults is True and the search was successful, the edit display is updated with the result of the search. The portion of the text that is appended to the end of the search criteria (the entered character) is selected. This makes it easy to continue searching (the next entered character replaces the selected text.)

TOvcDbSearchEdit Component

The TOvcDbSearchEdit component is a descendant of the TOvcBaseISE class and provides all of the properties and methods of that class. In addition to the standard edit control functions, the Incremental Search Edit (ISE) component provides an easy-to-implement database search mechanism. By setting the DataSource property to a properly configured TDataSource component, the ISE component can perform database searches as the search criteria (a string, number, date, etc.) is being entered into the edit control. For example, entering the following characters into an ISE control could produce results something like that shown in Table 32.1.

Table 32.1: *ISE control example*

Entered Characters	**Display String**
`O`	`O`**`atmeal`**
`Or`	`Or`**`ange`**
`Ord`	`Ord`**`er`**
`Ordi`	`Ordi`**`nary`**

The bold characters represent the highlighted portion of the ISE's display string. By default, the portion of the ISE control's display text that is appended to the end of the search criteria (the entered characters) is highlighted. This makes it easy to continue searching (the next entered character replaces the highlighted text) and to see exactly what the current match is.

Hierarchy

TCustomEdit (VCL)

- ❶ TOvcBaseISE (OvcISEB). 1136
  - TOvcDbSearchEdit(OvcDbISE)

Properties

About	❶ CaseSensitive	❶ KeyDelay
AttachedLabel	DataSource	❶ ShowResults
❶ AutoSearch	DbEngineHelper	

Methods

❶ PerformSearch

Reference Section

About read-only property

```
property About : string
```

↳ Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

AttachedLabel property

```
property AttachedLabel : TOvcAttachedLabel
```

↳ Provides access to the TOvcAttachedLabel component.

The TOvcAttachedLabel is normally accessed only at design time, but this property makes it possible to alter the attached label at run time if necessary.

See “TOvcAttachedLabel Component” on page 762 for more information.

DataSource property

```
property DataSource : TDataSource
```

↳ Specifies the data source component that contains the data to display.

DbEngineHelper property

```
property DbEngineHelper : TOvcDbEngineHelperBase
```

↳ Identifies a component that acts as source for information specific to different database systems.

This component uses the component assigned to DbEngineHelper to assist in obtaining information from the database system in a generic way. Its primary purpose is to allow this and other data-aware Orpheus components to work with third-part database systems.

If you are using the BDE, place a TOvcDbBDEHelper component on the form and assign its name to this property. Additional “Helper” components designed to support other database systems are available. See the DBHELPER.HLP in the main Orpheus directory for additional information.

A value must be assigned to this property.

TOvcDbIndexSelect Component

TOvcIndexSelect is a descendant of the TCustomComboBox component and provides a drop- down list of the indexes available in the attached dataset. The list of indexes can be configured to display the index name in several ways to include a user-defined string for each index.

Selecting one of the index names from the drop-down list changes the dataset's active index so that future operations on that dataset use the newly selected index.

Hierarchy

TCustomComboBox (VCL)

TOvcDbIndexSelect (OvcDbIdx)

Properties

About	DbEngineHelper	MonitorIndexChanges
AttachedLabel	DisplayMode	
DataSource	LabelInfo	

Methods

RefreshList	RefreshNow	SetRefreshPendingFlag

Events

OnGetDisplayLabel

Reference Section

About — read-only property

```
property About : string
```

↳ Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

AttachedLabel — property

```
property AttachedLabel : TOvcAttachedLabel
```

↳ Provides access to the TOvcAttachedLabel component.

The TOvcAttachedLabel is normally accessed only at design time, but this property makes it possible to alter the attached label at run time if necessary.

See “TOvcAttachedLabel Component” on page 762 for more information.

DataSource — property

```
property DataSource : TDataSource
```

↳ Specifies the data source component that contains the data to display.

DbEngineHelper — property

```
property DbEngineHelper : TOvcDbEngineHelperBase
```

↳ Identifies a component that acts as source for information specific to different database systems.

This component uses the component assigned to DbEngineHelper to assist in obtaining information from the database system in a generic way. Its primary purpose is to allow this and other data-aware Orpheus components to work with third-part database systems.

If you are using the BDE, place a TOvcDbBDEHelper component on the form and assign its name to this property. Additional “Helper” components designed to support other database systems are available. See the DBHELPER.HLP in the main Orpheus directory for additional information.

A value must be assigned to this property.

DisplayMode property

```
property DisplayMode : TDisplayMode
```

```
TDisplayMode = (dmFieldLabel, dmIndexName,
  dmFieldNames, dmFieldNumbers, dmUserDefined);
```

Default: dmFieldLabel

↳ Determines the text that is displayed in the drop-down list.

The following is a list of possible values for DisplayMode and their meanings:

Value	**Meaning**
dmFieldLabel	Displays the DisplayLabel of the primary field in the index.
dmIndexName	Displays the actual index name ("Default" for Paradox's primary index).
dmFieldNames	Displays the list of field names that make up the index.
dmFieldNumbers	Displays the list of field numbers that make up the index.
dmUserDefined	Displays a string returned from calls to the OnGetDisplayLabel event handler.

See also: OnGetDisplayLabel

LabelInfo property

```
property LabelInfo : TOvcLabelInfo
```

↳ Provides access to the status of the attached label.

TOvcLabelInfo groups the Visible, OffsetX, and OffsetY properties that determine whether the attached label is visible and where it is positioned.

MonitorIndexChanges property

```
property MonitorIndexChanges : Boolean
```

Default: False

↳ Determines whether changes in the active index are automatically detected.

If MonitorIndexChanges is True, changes in the active index of the attached datasource are detected by the TOvcDbIndexSelect component and the display is updated to reflect the new index name. If MonitorIndexChanges is False, the display is not updated.

OnGetDisplayLabel event

```
property OnGetDisplayLabel : TGetDisplayLabelEvent

TGetDisplayLabelEvent = procedure(
  Sender : TOvcDbIndexSelect; const FieldNames,
  IndexName : string; IndexOptions : TIndexOptions;
  var DisplayName : string) of object;
```

↳ Defines an event handler that is called to get the text to display as the index name.

FieldNames contains one or more field names (separated by semicolons) by which the current index is defined. IndexName is the name of the index as provided by the TTable associated with the datasource. IndexOptions contains the options that were used when the index was created and is defined by the VCL. DisplayName is the text that will be displayed as the name of the index. You can assign any string to the DisplayName parameter.

See also: DisplayMode

RefreshList method

```
procedure RefreshList;
```

↳ Updates the list of index names, if necessary.

If the refresh-pending flag is True, RefreshList updates the list of index names. If the refresh-pending flag is False, RefreshList does nothing.

See also: RefreshNow, SetRefreshPendingFlag

RefreshNow method

```
procedure RefreshNow;
```

↳ Forces an update of the list of index names.

RefreshNow sets the refresh-pending flag to True and then calls RefreshList.

SetRefreshPendingFlag method

```
procedure SetRefreshPendingFlag;
```

Sets an internal flag that indicates that the list of index names needs to be updated.

After calling SetRefreshPendingFlag, the list of index names is updated the next time it requires painting.

See also: RefreshList, RefreshNow

TOvcCaret Class

The TOvcCaret class provides a wrapper around the most frequently used Windows caret handling routines. This class provides configuration options for the editing carets used by the Orpheus entry field, editor, and viewer components. It provides properties that allow you to customize the shape, placement, blink rate, and size of the editing caret.

This section describes how to use the properties provided by the TOvcCaret class as it is used with other Orpheus components. It does not explain how to incorporate the TOvcCaret class into your own components. Refer to the entry field, editor, or viewer source code for examples of how to implement this class in a component.

Hierarchy

TPersistent (VCL)

    TOvcCaret (OvcCaret)

Properties

Align	BitmapHotSpotY	CaretWidth
Bitmap	BlinkTime	IsGray
BitmapHotSpotX	CaretHeight	Shape

Reference Section

Align property

```
property Align : TOvcCaretAlign
```

```
TOvcCaretAlign = (caLeft, caTop, caRight, caBottom, caCenter);
```

Default: caLeft

↳ Determines the placement of the caret.

This property determines the placement of the caret shape within a character cell. A character cell is defined by the average width and height of the font being used. Possible align values are:

Value	Meaning
caLeft	The caret is at the left edge of the character cell and centered vertically.
caTop	The caret is at the top of the character cell and cen tered horizontally.
caRight	The caret is at the right edge of the character cell and centered vertically.
caBottom	The caret is at the bottom of the character cell and centered horizontally.
caCenter	The caret is centered horizontally and vertically within the character cell.

Bitmap property

```
property Bitmap : TBitmap
```

↳ Defines the bitmap resource that is used for the caret shape.

If the Shape property is set to csBitmap, the bitmap assigned to Bitmap is used as the caret shape. When the bitmap is displayed, it is XOR’ed with the colors beneath it, so take that into account when you choose the colors for the bitmap.

See also: Shape

BitmapHotSpotX property

```
property BitmapHotSpotX : Integer
```

Default: 0

The X offset (in pixels) for a bitmap caret.

If Shape is set to csBitmap, this property is used when positioning the bitmap within the character cell. For any other value of Shape, BitmapHotSpotX is ignored. The value of this property is best determined by experimentation.

See also: Bitmap, BitmapHotSpotY, Shape

BitmapHotSpotY property

```
property BitmapHotSpotY : Integer
```

Default: 0

The Y offset (in pixels) for a bitmap caret.

If Shape is set to csBitmap, this property is used when positioning the bitmap within the character cell. For any other value of Shape, BitmapHotSpotY is ignored. The value of this property is best determined by experimentation.

See also: Bitmap, BitmapHotSpotX, Shape

BlinkTime property

```
property BlinkTime : Word
```

Default: Current Windows setting

Determines the rate at which the caret blinks.

The caret flashes on or off every BlinkTime milliseconds. One complete flash (off and on) takes BlinkTime*2 milliseconds.

CaretHeight property

```
property CaretHeight : Integer
```

Default: 10

The height of the caret in pixels.

If the Shape property is set to csCustom, CaretHeight determines the height of the caret. If Shape is csBitmap, the height of the caret is determined by the actual height of the bitmap.

See also: CaretWidth, Shape

CaretWidth property

```
property CaretWidth : Integer
```

Default: 2

↳ The width of the caret.

If the Shape property is set to csCustom, CaretWidth determines the width of the caret. If Shape is csBitmap, the width of the caret is determined by the actual width of the bitmap.

See also: CaretHeight, Shape

IsGray property

```
property IsGray : Boolean
```

Default: False

↳ Determines whether the caret is “grayed out.”

Shape property

```
property Shape : TOvcCaretShape
```

```
TOvcCaretShape = (csBlock, csHalfBlock, csVertLine,
  csHorzLine, csCustom, csBitmap);
```

Default: csVertLine

↳ Determines the shape of the caret.

The possible shape values are:

Value	**Meaning**
csBlock	A block caret that covers the entire character cell.
csHalfBlock	A block caret that covers the bottom half of the char acter cell.
csVertLine	A vertical line caret positioned at the left (by default) of the character cell. Use Align to change the position of the caret.
csHorzLine	A horizontal line caret positioned on the bottom (by default) of the character cell. Use Align to change the position of the caret.
csCustom	CaretWidth and CaretHeight determine the shape of the caret.
csBitmap	The caret defined by the Bitmap property.

See also: Align, Bitmap, CaretHeight, CaretWidth

TOvcColors Class

The TOvcColors class is used by many Orpheus components to combine text and background colors in one property. In addition to visually grouping the colors in the Object Inspector, TOvcColors also provides the ability to revert to default text and background colors.

Setting the UseDefault property to True restores the default text and background colors. The default colors are set by passing them as parameters to the constructor when an instance of the TOvcColors class is created.

```
  constructor Create(FG, BG : TColor); virtual;
```

FG and BG are the default text and background colors for this instance of TOvcColors.

This section describes how to use the properties provided by the TOvcColors class as it is used with your own Orpheus components. It does not explain how to incorporate the TOvcColors class into other components. Refer to the entry field or editor source code for examples of how to implement this class in a component.

Hierarchy

TPersistent (VCL)

    TOvcColors (OvcColor)

Properties

BackColor	TextColor	UseDefault

Events

OnColorChange

Reference Section

BackColor property

```
property BackColor : TColor
```

The background color.

See also: TextColor

OnColorChange event

```
property OnColorChange : TNotifyEvent
```

Defines an event handler that is called when one of the colors managed by this object is changed.

TNotifyEvent is defined in the VCL's Classes unit.

TextColor property

```
property TextColor : TColor
```

The text color.

See also: BackColor

UseDefault property

```
property UseDefault : Boolean
```

Default: True

Determines whether the default colors are used.

If UseDefault is True, the default color values are assigned to the TextColor and BackColor properties. If you set the TextColor or BackColor properties, UseDefault is set to False.

See also: BackColor, TextColor

TOvcFixedFont Class

The TOvcFixedFont class is very similar to the standard VCL TFont class. The only difference is that the Name property—which lists the available font names, is limited to displaying names of fixed-pitch fonts only. Several Orpheus components require a fixed-pitch font and the TOvcFixedFont class allows them to present only the fonts that are valid.

The TOvcFixedFont class provides Color, Font, Name, Size, and Style properties, but omits the Pitch property. With the exception of the Name and Font properties, their behavior is identical to the TFont properties of the same names.

The Font property provides access to the TOvcFixedFont's internal TFont object as a read-only property. This might be required if you need to assign the font to another component or class. One such situation occurs when you print the text contained in an Orpheus editor component. You could do something like this to assign the font that the editor is currently using to a printer object:

```
Printer.Canvas.Font := Editor1.FixedFont.Font;
```

To assign a valid TFont (a fixed-pitch font) or another TOvcFixedFont object to an instance of TOvcFixedFont at run time, assign the font object to the TOvcFixedFont object using the Assign method:

```
var
  MyFont      : TFont;
  MyFixedFont : TOvcFixedFont;
...
Editor1.FixedFont.Assign(MyFont);
... {or}
Editor1.FixedFont.Assign(MyFixedFont);
```

Attempting to assign a font that is not a fixed pitch font to a TOvcFixedFont object raises an EInvalidFixedFont exception. Attempting to assign something other than a TFont or TOvcFixedFont using the Assign method raises an EInvalidFontParam exception.

Hierarchy

TPersistent (VCL)

    TOvcFixedFont (OvcFxFnt)

TOvcUserData Class

The TOvcUserData class serves three purposes. It defines the set of valid characters for an entry field's mask character, it defines the case-changing characteristics of the mask character, and it defines the meaning of the substitution characters (Subst1 through Subst8).

For information about how Orpheus entry fields use this class and how you can alter the behavior of the Orpheus entry fields, see "User-defined mask characters" on page 322 and "Substitution characters" on page 323.

A global instance of the TOvcUserData class (named OvcUserData) is created automatically and used as the default UserData property value for all Orpheus components that support a UserData property.

Changing the properties of the global instance of TOvcUserData (OvcUserData), alters the behavior of all components using the global instance. If you want to isolate one or more components from the default global instance of TOvcUserData, you can create a new instance of TOvcUserData, set properties to achieve the desired behavior, and then assign the new instance to the UserData property of the individual components.

Hierarchy

TObject (VCL)

  TOvcUserData (OvcUser)

Properties

ForceCase	SubstChars	UserCharSet

Reference Section

ForceCase property

```
property ForceCase[Index : TForceCaseRange] : TCaseChange;
```

```
TForceCaseRange = pmUser1..pmUser8;
```

```
TCaseChange = (mcNoChange, mcUpperCase, mcLowerCase, mcMixedCase);
```

Default: mcNoChange (for all eight array elements)

Determines whether the case of entered characters is changed.

This property sets the case changing state for the corresponding user-defined picture mask character. Valid values for Index are pmUser1 through pmUser8.

The possible values for ForceCase are:

Value	**Meaning**
mcNoChange	No change to the character entered.
mcUpperCase	Force upper case.
mcLowerCase	Force lower case.
mcMixedCase	Force mixed case (e.g., “john jones” becomes “John Jones”).

The user-defined picture mask characters are:

```
  pmUser1 = '1';
  pmUser2 = '2';
  pmUser3 = '3';
  pmUser4 = '4';
  pmUser5 = '5';
  pmUser6 = '6';
  pmUser7 = '7';
  pmUser8 = '8';
```

SubstChars property

```
property SubstChars[Index : TSubstCharRange] : AnsiChar
```

```
TSubstCharRange = Subst1..Subst8;
```

Default: (no substitution)

↳ Determines what characters are substituted for a user-defined substitution character at run time.

By default, all elements of this array are assigned their corresponding substitution character (SubstChars[Subst1] := Subst1, SubstChars[Subst2] := Subst2, and so on).

The user-defined substitution characters are:

```
  Subst1 = #241;
  Subst2 = #242;
  Subst3 = #243;
  Subst4 = #244;
  Subst5 = #245;
  Subst6 = #246;
  Subst7 = #247;
  Subst8 = #248;
```

UserCharSet property

```
property UserCharSet[Index : TUserSetRange] : TCharSet
```

```
TUserSetRange = pmUser1..pmUser8;
```

```
TCharSet = set of AnsiChar;
```

Default: 0-255 (for all eight array elements)

↳ Determines the characters allowed in the positions that use a user-defined mask character.

By default, all characters are allowed. To limit the allowed characters, assign a different set for one of the user-defined picture mask characters.

LabelInfo property

```
property LabelInfo : TOvcLabelInfo
```

↳ Provides access to the status of the attached label.

TOvcLabelInfo (see page 764) groups the Visible, OffsetX, and OffsetY properties that determine whether the attached label is visible and where it is positioned.

ShowHidden property

```
property ShowHidden : Boolean
```

Default: True

↳ Determines whether hidden TField objects corresponding to indexes are displayed.

If ShowHidden is True, fields that have their Visible property set to False and that are used as one of the index fields are displayed in the list of index names.

Chapter 33: Deprecated Components

The introduction of a new LookoutBar, a new edit control and a new set of validation classes has greatly improved some aspects of the library. The new components represent sweeping changes to existing components or may replace components or groups of components completely. Some of the older, less-robust components have been deprecated to allow Orpheus to evolve while still maintaining a certain level of backward-compatibility. These components are documented in this chapter.

The deprecated components are located on the Orpheus 3.x tab of the component palette. The functionality of these components has been replaced by newer, more robust components. However, given the thousands of applications that have been developed with the older components and an acute awareness of the lack of resources in the industry, we understand that not all applications will be immediately re-worked to use the more advanced versions. We have therefore retained the deprecated components in the library and in this documentation.

Keep in mind that as Orpheus moves away from these components and classes, they will be generally treated with a much lower priority than the rest of the library. Furthermore, deprecated components may be removed from the library at some point in the future. With this in mind we recommend that you refrain from using the deprecated components in any new development and that you endeavor to replace them with the newer components when time permits.

Note: The deprecated components still descend from the Orpheus base classes, which no longer support Delphi 1 and 2. If you wish to continue to use Delphi 1 or 2 then you will need to keep Orpheus 3.x around.

The following components and classes are described:

- TOvcLookoutBar: The TOvcLookOutBar component emulates the control you see on the left side of Microsoft Outlook ™. Orpheus 4 includes a re-worked version of the LookoutBar. Since the changes to the LookoutBar impacted its interface, the decision was made to deprecate the older version as it is, and create a brand new component under a new name. For more information see “TO32LookOutBar Component” on page 112.

- TOvcNumberEdit: The TOvcNumberEdit introduces an edit control for entry of numeric data.

- TOvcDbNumberEdit: The TOvcDbNumberEdit is identical to the TOvcNumberEdit component with the additional ability of allowing it to be connected to and edit a field in a dataset.

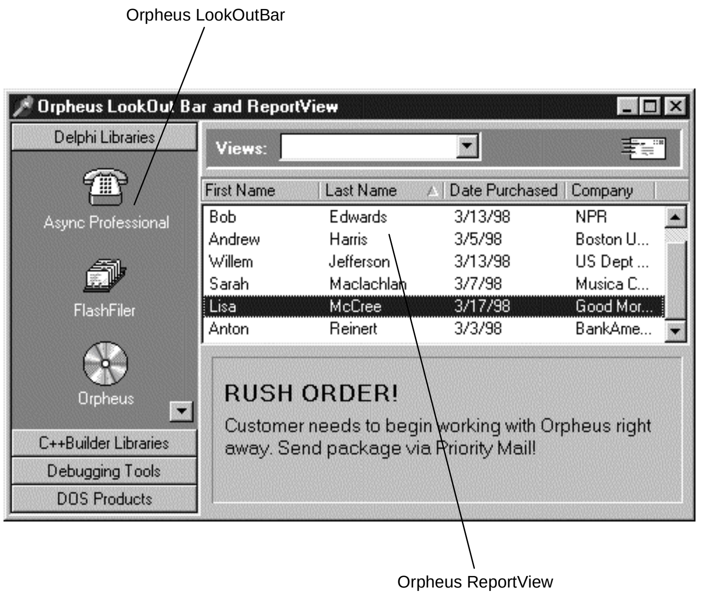
- TOvcDateEdit: TOvcDateEdit is a specialized edit control that provides enhanced date entry capabilities and a pop-up calendar.
- TOvcDbDateEdit: The TOvcDbDateEdit is identical to the TOvcDateEdit component with the additional ability of allowing it to be connected to and edit a field in a dataset.
- TOvcTimeEdit: TOvcTimeEdit does for time and duration entries what TOvcDateEdit does for date entry— greatly simplifies it. With the Time Edit control you can enter something like "2" and have it interpreted as 2:00 PM (or 2:00 AM). You can set up the field to work with duration of time instead of time of day and have "2h 3m 33s" interpreted as 2 hours 3 minutes 33 seconds.
- TOvcDbTimeEdit: The TOvcDbTimeEdit is identical to the TOvcTimeEdit Component with the additional ability of allowing it to be connected to and edit a field in a dataset.
- TOvcSimpleArrayEditor: The TOvcSimpleArrayEditor component is derived from a TOvcBaseArrayEditor. The TOvcSimpleArrayEditor edit cells are just TOvcSimpleFields. The PictureMask, MaxLength, and DecimalPlaces properties behave just like the TOvcSimpleField properties of the same name.
- TOvcPictureArrayEditor: The TOvcPictureArrayEditor component is derived from a TOvcBaseArrayEditor. The TOvcPictureArrayEditor edit cells are just TOvcPictureFields. The PictureMask, MaxLength, and DecimalPlaces properties behave just like the TOvcPictureField properties of the same name.
- TOvcNumericArrayEditor: The TOvcNumericArrayEditor component is derived from a TOvcBaseArrayEditor. The TOvcNumericArrayEditor edit cells are just TOvcNumericFields. The PictureMask property behaves just like the TOvcNumericField property of the same name.
- TOvcDbSimpleArrayEditor: The TOvcDbSimpleArrayEditor component is derived from a TOvcBaseDbArrayEditor. It behaves exactly like the TOvcSimpleArrayEditor with the added ability to connect to a data source.
- TOvcDbPictureArrayEditor: The TOvcDbPictureArrayEditor component is derived from a TOvcBaseDbArrayEditor. The TOvcDbPictureArrayEditor edit cells are just TOvcDbPictureFields. The PictureMask, MaxLength, and DecimalPlaces properties behave just like the TOvcDbPictureField properties of the same name.
- TOvcDbNumericArrayEditor: The TOvcDbNumericArrayEditor component is derived from a TOvcBaseDbArrayEditor. The TOvcDbNumericArrayEditor edit cells are just TOvcDbNumericFields. The PictureMask property behaves just like the TOvcDbNumericField property of the same name.

- TOvcEdit: TOvcEdit is identical to the standard TEdit with the addition of the capability to have an attached label and to provide access to an Orpheus Controller (TOvcController class) for its descendants. See page 796 for additional information about the attached label.
- TOvcEdPopup: TOvcEdPopup implements an edit control with the ability to display a button embedded within the client area of the control. The action performed when the button is pressed is not defined in this class. As is indicated by the name of the class, TOvcEdPopup was created to provide the capability to “pop up” some other control. For example, TOvcEdPopup is used as the ancestor for a pop-up calendar control and a pop-up calculator control.

OvcLookOutBar

The TOvcLookoutBar component has been replaced with the TO32LookoutBar component. The new component has all of the functionality of the deprecated version, plus a slew of new functionality including standardized streaming, the ability to embed components into the folders and greater flexibility in customizing it's appearance. A number of bugs have been fixed in the new version as well. As the modifications required changes to the LookoutBar's interface, the older version has been deprecated and the new version is re-named TO32LookoutBar. It is documented in "Chapter 3: LookOutBar Component" on page 109.

The TOvcLookOutBar component emulates the control you see on the left side of Microsoft Outlook ™. The contents of the component are arranged by folders and items. Each TOvcLookOutBar contains one or more folders. Within each folder are one or more folder items. Folders are depicted as buttons on the TOvcLookOutBar. Clicking on a folder button makes that folder the active folder and displays its contents (the items in the folder). Clicking on a folder item generates an event that can be used to perform some action in the program. A typical use for an TOvcLookOutBar is to place the bar in the left side of a form with different item views displayed to the right of the bar. As shown below, using an TOvcLookOutBar and TOvcReportView together is a powerful combination.



First Name	Last Name	Date Purchased	Company
Bob	Edwards	3/13/98	NPR
Andrew	Harris	3/5/98	Boston U...
Willem	Jefferson	3/13/98	US Dept ...
Sarah	Maclachlan	3/7/98	Musica C...
Lisa	McCree	3/17/98	Good Mor...
Anton	Reinert	3/3/98	BankAme...

RUSH ORDER!

Customer needs to begin working with Orpheus right away. Send package via Priority Mail!

Figure 33.1: A form with the LookOut bar and Report view

Only one folder's contents can be visible at a given time. If the folder contains more items than can be displayed in the client area of the folder, scroll buttons will be displayed. Clicking the scroll buttons allows you see additional folder items.

Folder items can be displayed in one of two ways. When large icons are used (images greater than 16 by 16 pixels), the image is centered in the client area of the TOvcLookOutBar with the item's text below the image. When small icons are used, the icon image appears near the left edge of the TOvcLookOutBar with text to the right of the image. Icon is a generic term; the image list can contain either icons or bitmaps. Use a VCL TImageList component to store the item images.

The TOvcLookOutBar supports in-place editing for folders and folder items at run time. Calling the RenameFolder method invokes in-place editing of folder captions. In-place editing of folder item captions is invoked by calling RenameItem. Users can then directly type a new caption for the folder or item. Pressing the Enter key on the keyboard accepts the new caption and the updates TOvcLookOutBar. Pressing the ESC key abandons edits and the previous caption is restored. Persistence for folder and folder item captions can be implemented by using the Orpheus component state components. See page 257 for more information.

Two primary events are supplied by TOvcLookOutBar. The OnFolderClick event notifies you when a folder changes. The OnItemClick event notifies you when a folder item is clicked. You can use this event to change the view associated with the item clicked.

Folders can be added at design time using the Folder Editor. To invoke the Folder Editor double-click the TOvcLookOutBar on your form. Alternatively you can double-click the value column next to the FolderCollection property in the Object Inspector. The Folder Editor contains buttons to add, remove, or reorder folders. Click on a folder in the Folder Editor to display that folder's properties in the Object Inspector. Folder items are added at design time via the Folder Item Editor. This property editor is displayed when you double-click next to a folder's ItemCollection property in the Object Inspector. As with the Folder Editor, clicking a folder item in the Folder Items Editor displays that item's properties in the Object Inspector.

TOvcLookOutBar Component

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase). 16

❷ TOvcCustomControlEx (OvcBase) 19

TOvcLookOutBar (OvcLkOut)

Properties

❶	About		ButtonHeight	❶	LabelInfo
	ActiveFolder	❷	Controller		PreviousFolder
	ActiveItem		DrawingStyle		PreviousItem
	AllowRearrange		FolderCollection		PlaySounds
❶	AttachedLabel		FolderCount		ScrollDelta
	BackgroundColor		Folders		SoundAlias
	BackgroundImage		Images		
	BackgroundMethod		ItemFont		

Methods

BeginUpdate	InsertFolder	RemoveItem
EndUpdate	InsertItem	RenameFolder
GetFolderAt	InvalidateItem	RenameItem
GetItemAt	RemoveFolder	

Events

❶	AfterEnter		OnFolderChange	❶	OnMouseOverItem
❶	AfterExit		OnFolderChanged		OnMouseWheel
	OnDragDrop		OnFolderClick		
	OnDragOver		OnItemClick		

Reference Section

ActiveFolder — property

```
property ActiveFolder : Integer
```

↳ The index number of the active folder.

Read ActiveFolder to determine the index number of the active folder. Set ActiveFolder to programmatically set the active folder. Folders are 0-based with the first folder at index 0, the second at index 1, and so on.

See also: FolderCount, InsertFolder, RemoveFolder, RenameFolder

ActiveItem — run-time, read-only property

```
property ActiveItem : Integer
```

↳ The index number of the active folder item.

See also: ActiveFolder, TOvcLookOutFolder.Items

AllowRearrange — property

```
property AllowRearrange : Boolean
```

Default: True

↳ Determines whether or not the items within a folder can be rearranged by dragging.

When AllowRearrange is True, items can be moved via drag and drop. When AllowRearrange is False, items cannot be rearranged. Folder items can be dragged within a folder or between folders. When dragging between folders, the target folder will be expanded when the mouse cursor is dragged over a folder button.

See also: Items

BackgroundColor — property

```
property BackgroundColor : TColor
```

Default: clInactiveCaption

↳ The background color of the LookOut Bar.

BackgroundColor is applied to the background when the BackgroundMethod property is set to bmNone.

See also: BackgroundImage, BackgroundMethod

BackgroundImage property

```
property BackgroundImage : TBitmap
```

↳ The image that is displayed in the LookOut Bar's background.

By default BackgroundImage is nil and the background color is determined by BackgroundColor. Set BackgroundImage to display a bitmap on the background of the component. The background fill method (none, normal, or stretch) is determined by the BackgroundMethod property.

See also: BackgroundColor, BackgroundMethod

BackgroundMethod property

```
property BackgroundMethod : TOvcBackgroundMethod
```

Default: bmNormal

↳ The painting method to paint the background.

The possible values for BackgroundMethod are:

Value	**Description**
bmNone	No background bitmap is applied. The background is filled using the color specified in BackgroundColor.
bmNormal	The bitmap specified in BackgroundImage is used to paint the background. The image is not sized to fit the component.
bmStretch	The background is filled with the bitmap specified in BackgroundImage. The image is stretched to fit the size of the component.

Setting BackgroundMethod to bmNormal or bmStretch has no effect if BackgroundImage is not assigned

See also: BackgroundColor, BackgroundImage

BeginUpdate method

```
procedure BeginUpdate;
```

↳ Temporarily disables repainting of the control.

Call BeginUpdate to disable repainting of the component's items if you are programmatically adding many items to a folder. Call EndUpdate after adding items to force the component to be repainted. Each call to BeginUpdate must have a corresponding EndUpdate call. The following example disables painting of the component, adds items to the first folder in the folder list, and then re-enables painting:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I  : Integer;
begin
  OvcLookOutBar1.BeginUpdate;
  for I := 0 to 19 do
    OvcLookOutBar1.InsertItem('Item ' + IntToStr(I + 1),
                              0, I, I);
  OvcLookOutBar1.EndUpdate;
end;
```

See also: EndUpdate

ButtonHeight property

```
property ButtonHeight : Integer
```

Default: 20

↳ The height of the LookOut Bar's folder button.

DrawingStyle property

```
property DrawingStyle : TOvcFolderDrawingStyle
```

```
TOvcFolderDrawingStyle = (dsDefault, dsEtched);
```

↳ The style used to draw the LookOut Bar.

When DrawingStyle is set to dsDefault, the border and folder buttons are drawn using the Microsoft Outlook97™ style. When DrawingStyle is set to dsEtched, the component is drawn using the Outlook98™ style.

EndUpdate method

```
pocedure EndUpdate;
```

↳ Repaints the component after adding items with painting disabled.

Call EndUpdate to repaint the component after adding items when repainting has been disabled (by calling BeginUpdate). Each call to BeginUpdate must have a corresponding call to EndUpdate. See BeginUpdate for an example.

See also: BeginUpdate

FolderCollection property

```
property FolderCollection : TOvcCollection
```

↳ The list of folders for the component.

A TOvcLookOutBar contains one or more folders. FolderCollection contains the list of folders. Each folder's characteristics are set at design time via the FolderCollection property editor or at run time through code. To enumerate a TOvcLookOutBar's folders, use the Folders property. To add folders at run time, call the InsertFolder method. To remove folders at run time, call the RemoveFolder method.

See also: FolderCount, Folders, InsertFolder, RemoveFolder

FolderCount run-time, read-only property

```
property FolderCount : Integer
```

↳ The number of folders in the TOvcLookOutBar.

See also: Folders, FolderCollection

Folders run-time, read-only property

```
property Folders[Index : Integer] : TOvcLookOutFolder
```

↳ An indexed property containing the folders in the TOvcLookOutBar.

Use Folders at run time to access a particular folder by its folder index or to enumerate the folders. The following example changes the caption of the second folder in the list:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  OvcLookOutBar1.Folders[1].Caption := 'Inbox';
end;
```

See also: FolderCollection, FolderCount

GetFolderAt method

```
function GetFolderAt(X, Y : Integer) : Integer;
```

Returns the folder index of the folder at the given coordinates.

See also: Folders, GetItemAt

GetItemAt method

```
function GetItemAt(X, Y : Integer) : Integer;
```

Returns the item index of the folder item at the given coordinates.

See also: GetFolderAt, TOvcLookOutFolder.Items

Images property

```
property Images : TImageList
```

The image list that contains the images used to display folder items.

Set Images to the name of the image list that contains the folder item images. Set each folder item's IconIndex property to the index corresponding to the image to display for that item. The folder items' display characteristics are determined by the folder's IconSize property. For large icons 32 by 32 pixels is the recommended size. For small icons, 16 by 16 pixel images are recommended.

Use a TImageList component to store the item images.

See also: TOvcLookOutItem.IconIndex, TOvcLookOutFolder.IconSize

InsertFolder method

```
procedure InsertFolder(
  const ACaption : string; AFolderIndex : Integer);
```

Adds a folder to the OvcLookOutBar.

ACaption is the caption for the new Folder. AFolderIndex is the position in the list where the new item will be inserted. The following example adds a new folder at the bottom of a TOvcLookOutBar:

```
  procedure TForm1.Button1Click(Sender : TObject);
  begin
    OvcLookOutBar1.InsertFolder('Outbox',
  OvcLookOutBar1.FolderCount);
  end;
```

See also: InsertItem, FolderCollection, Folders, RemoveFolder

InsertItem method

```
procedure InsertItem(const ACaption : string;
  AFolderIndex, AItemIndex, AIconIndex : Integer);
```

↳ Adds a folder item into a folder.

Call InsertItem to programmatically add an item to a folder at run time. ACaption is the caption for the new item. AFolderIndex is the index number of the parent folder for the new item. AItemIndex is the position within the folder where the new item will be added. AIconIndex is the index of the image within the image list that will be used to display the new item.

The following example inserts a new item at the top of the first folder in a TOvcLookOutBar (folder index 0, item index 0). The image index is set to the sixth image in the image list (image index 5).

```
  procedure TForm1.Button1Click(Sender: TObject);
  begin
    OvcLookOutBar1.InsertItem('New Item', 0, 0, 5);
  end;
```

See also: InsertFolder, RemoveItem

InvalidateItem method

```
procedure InvalidateItem(FolderIndex, ItemIndex : Integer);
```

↳ Forces a redraw of the specified item.

ItemFont property

```
property ItemFont : TFont
```

↳ The font that is used to draw the text associated with a folder item. If ItemFont is unassigned, the OvcLookOutBar's font will be used.

OnDragDrop event

```
property OnDragDrop : TOvcLOBDragDropEvent

TOvcLOBDragDropEvent = procedure(Sender, Source: TObject;
  X, Y: Integer; FolderIndex, ItemIndex : Integer)of object;
```

↳ Defines an event handler that is generated when a drop occurs from another component.

Source is the component that initiated the drag event. X is the cursor x position and Y is the cursor y position. State is the drag state (see the VCL help for an explanation of TDragState). FolderIndex is the index of the folder on which the drop occurred. ItemIndex is the item index position of the drop within the folder. The value of ItemIndex varies depending on whether you are allowing a drop on folders, on items, or both. If you are allowing drop on folders, ItemIndex is the index where a new item will be inserted (when you call InsertItem, for example). If you are allowing drop on items within a folder, ItemIndex is the index number of the item that was under the cursor when the drop occurred.

See also: OnDragOver

OnDragOver event

```
property OnDragOver : TOvcLOBDragOverEvent

TOvcLOBDragOverEvent = procedure(Sender, Source : TObject;
  X, Y : Integer; State : TDragState;
  var AcceptFolder, AcceptItem : Boolean) of object;
```

↳ Defines an event handler that is generated when an item is being ragged from another component.

Source is the component that initiated the drag event. X is the cursor x position and Y is the cursor y position. State is the drag state (see the VCL help for an explanation of TDragState). AcceptFolder controls whether or not a folder can accept a drop. When AcceptFolder is True a line is drawn on the folder at the cursor position. An item can be dropped on the folder at the indicated position. AcceptItem controls whether or not a folder item can accept a drop. When AcceptItem is True, the folder item under the mouse cursor is highlighted to indicate that a drop on the folder item is valid. If neither AcceptFolder nor AcceptItem are True then the no drop cursor is displayed and the OnDragDrop event is not generated. AcceptFolder does not have to be True to accept drops on folder items.

See also: OnDragDrop

OnFolderClick event

```
property OnFolderClick : TFolderClickEvent
```

```
TFolderClickEvent = procedure(
  Sender : TObject; Button : TMouseButton;
  Shift : TShiftState; Index : Integer) of object;
```

↳ Defines an event handler that is generated when a folder button is clicked.

Button is the mouse button that was use to click the folder. Shift is the keyboard state when the folder was clicked. Index is the folder index of the folder that was clicked. Provide an event handler for OnFolderClick to determine when a folder button is clicked.

See also: OnItemClick

OnFolderChange event

```
property OnFolderChange : TFolderChangeEvent
```

```
TFolderChangeEvent = procedure(
  Sender : TObject; Index : Integer;
  var AllowChange : Boolean; Dragging : Boolean) of object;
```

↳ Defines an event handler that is generated when the ctive folder is about to change.

Index is the index number of the folder that is about to become active. AllowChange determines whether or not the folder can change. By default AllowChange is True. Setting AllowChange to False within your event handler prevents the folder change from taking place. Dragging is a flag that indicates whether a folder item is being dragged. You may want to allow a particular folder to change if the user is attempting to drag an item from one folder to another, but disallow the folder change during normal operation.

See also: OnFolderChanged

OnFolderChanged event

```
property OnFolderChanged : TFolderChangedEvent
```

```
TFolderChangedEvent = procedure(
  Sender : TObject; Index : Integer) of object;
```

↳ Defines an event handler that is generated when the active folder changes.

Index is the index number of the new active folder. OnFolderChange is a notification event. As such, it is generated after the active folder changes. To detect a folder state change before the folder changes, use the OnFolderChange event.

See also: OnFolderChange

OnItemClick event

```
property OnItemClick : TItemClickEvent
```

```
TItemClickEvent = procedure(
  Sender : TObject; Button : TMouseButton;
  Shift : TShiftState; Index : Integer) of object;
```

Defines an event handler that is generated when a folder item is clicked.

Button is the mouse button that was use to click the folder. Shift is the keyboard state when the item was clicked. Index is the item index of the item that was clicked. Provide an event handler for OnFolderClick to determine when a folder button is clicked. Read the ActiveFolder property to determine the folder that the item clicked belongs to.

The following example changes pages in a notebook based on the item clicked. This example assumes the OvcLookOutBar contains a single folder:

```
  procedure TForm1.OvcLookOutBar1ItemClick(
    Sender : TObject; Button : TMouseButton;
    Shift : TShiftState; Index : Integer);
  begin
    NoteBook1.PageIndex := Index;
  end;
```

See also: OnFolderClick

OnMouseOverItem event

```
property OnMouseOverItem : TOvcMouseOverItemEvent
```

```
TOvcMouseOverItemEvent = procedure(
  Sender : TObject; Item : TOvcLookOutItem) of object;
```

Defines an event handler that is fired when the mouse cursor moves over a TOvcLookOutBar item.

Sender is the object that generated the event. Item is a pointer to the item that the cursor is currently over. The following example shows how to display each item’s description in a status bar:

```
  procedure TForm1.OvcLookOutBar1MouseOverItem(
    Sender : TObject; Item : TOvcLookOutItem);
  begin
    StatusBar1.SimpleText := Item.Description;
  end;
```

PlaySounds property

```
property PlaySounds : Boolean
```

Default: False

Determines whether or not a sound is played when the active folder changes.

Set PlaySounds to True to enable playing of a sound when the active folder changes. The sound played is determined by the SoundAlias property.

See also: SoundAlias

PreviousFolder run-time, read-only property

```
property PreviousFolder : Integer
```

The index of the folder that was previously selected.

Read PreviousFolder to determine the index of the folder that was previously selected. If no previous folder was selected PreviousFolder contains –1.

See also: ActiveFolder, Folders, FolderCollection, OnFolderClick, PreviousItem

PreviousItem run-time, read-only property

```
property PreviousItem : Integer
```

The index of the folder item that was previously clicked.

Read PreviousItem to determine the index number of the item that was previously clicked. If no item was previously clicked, PreviousItem contains –1. To determine the folder containing the previously clicked item, read the PreviousFolder or ActiveFolder properties.

See also: OnItemClicked, PreviousFolder, TOvcLookOutFolder.Items

RemoveFolder method

```
procedure RemoveFolder(AFolderIndex : Integer);
```

Removes a folder from a TOvcLookOutBar.

Call RemoveFolder to delete a folder from the folder list. AFolderIndex is the index number of the folder to remove.

See also: Folders, FolderCollection, InsertFolder, RemoveItem

RemoveItem method

```
procedure RemoveItem(AFolderIndex, AItemIndex : Integer);
```

Removes a folder item from a TOvcLookOutBar.

Call RemoveItem to remove an item from a folder. AFolderIndex is the index number of the folder containing the item to remove. AItemIndex is the index number of the item to remove. The following example removes the first item in the second folder of a TOvcLookOutBar:

```
procedure TForm1.Button3Click(Sender : TObject);
begin
  OvcLookOutBar1.RemoveItem(1, 0);
end;
```

See also: InsertItem, TOvcLookOutFolder.Items, TOvcLookOutFolder.ItemCollection

RenameFolder method

```
procedure RenameFolder(AFolderIndex : Integer);
```

Enables in-place editing of a folder's caption.

Call RenameFolder to invoke the OvcLookOutBar's in-place editor. Users can then type a new caption for the folder. Pressing <Esc> abandons the operation and returns the folder caption to its original state. Pressing the <Enter> changes the folder's caption to the new text. The Orpheus state components can be used to make folder caption changes persistent.

See also: InsertFolder, RemoveFolder, RenameItem

RenameItem method

```
procedure RenameItem(AFolderIndex, AItemIndex : Integer);
```

Enables in-place editing of an item's caption.

Call RenameItem to invoke the OvcLookOutBar's in-place editor. Users can then type a new name for the item. Pressing the <Esc> abandons the operation and returns the item caption to its original state. Pressing the <Enter> changes the item's caption to the new text. The Orpheus state components can be used to make folder item caption changes persistent.

See also: InsertItem, RemoveItem, RenameFolder

ScrollDelta property

```
property ScrollDelta : Integer
```

Default: 2

Determines the scrolling granularity when a folder changes.

When a folder button is clicked, the new folder scrolls into view. A lower ScrollDelta results in finer (and slower) scrolling. Set ScrollDelta to a higher number to increase the scroll speed.

SoundAlias property

```
property SoundAlias : string
```

The sound that will be played when the active folder changes.

SoundAlias can be either a filename or the name of a system sound. System sounds include Minimize, Maximize, MailBeep, and so on. System sound names are listed in either the registry (system sound names can be found under the HKEY_CURRENT_USER\AppEvents\Schemes registry key). Using system sounds is advantageous in that it allows you to incorporate the user's current sound scheme. You may, for example, want to associate a folder change with Windows' minimize sound. If the specified sound cannot be found, the default Windows sound is played (usually the "ding" sound). SoundAlias is ignored if the PlaySounds property is set to *False.*

See also: PlaySounds

TOvcLookOutFolder Class

Hierarchy

TCustomControl (VCL)

❶ TOvcComponent (OvcBase) 21

❷ TOvcCollectible (OvcBase) 41

TOvcLookFolder (OvcLkOut)

Properties

❶	About		IconSize		Items
	Caption	❷	Index	❷	Name
❷	Collection		ItemCount		
❷	DisplayText		ItemCollection		

33

Reference Section

Caption property

```
property Caption : string
```

↳ The text displayed on a folder's button.

A folder's caption can be changed at run time by assigning a new value to Caption. Additionally, you can call TOvcLookOutBar.RenameFolder to invoke the OvcLookOutBar's in-place editor.

See also: TOvcLookOutBar.RenameFolder

IconSize property

```
property IconSize : TOvcIconSize
```

```
TOvcIconSize = (isLarge, isSmall);
```

Default: isLarge

↳ Determines the display characteristics of folder items.

IconSize can be either isLarge or isSmall. When IconSize is set to isLarge, images are displayed centered on the OvcLookOutBar with the text below the image. When IconSize is set to isSmall the image is drawn on the left side of the OvcLookOutBar with the text to the right of the image. For large icons an image size of 32 by 32 pixels is recommended. For small icons a 16 by 16 pixel image is recommended. Using images larger than 16 by 16 with the isSmall setting may result in the item's text overlapping the image.

See also: TOvcLookOutBar.Images

ItemCollection property

```
property ItemCollection : TOvcCollection
```

↳ The list of items in a folder.

A TOvcLookOutBar contains one or more folders with one or more items in each folder. ItemCollection contains the list of items for a folder. Each item's characteristics are set at design time via the ItemCollection property editor or by run-time through code. To enumerate a folder's items, use the Items property. To add items at run time call the OvcLookOutBar's InsertItem method. To remove items at run time, call the RemoveItem method.

See also: Items, ItemCount, TOvcLookOutBar.InsertItem, TOvcLookOutBar.RemoveItem

ItemCount run-time, read-only property

```
property ItemCount : Integer
```

The number of items in the folder.

See also: Items, ItemCollection

Items run-time, read-only property

```
property Items[Index : Integer] : TOvcLookOutItem
```

An indexed property that allows access to the items in a folder.

Use Items to enumerate the items in a folder, or to access a particular item by index number. The following example changes the caption of the second item in the first folder of a OvcLookOutBar:

```
  procedure TForm1.Button3Click(Sender: TObject);
  begin
    OvcLookOutBar1.Folders[0].Items[1].Caption := 'Outbox';
  end;
```

See also: ItemCount, ItemCollection

TOvcLookOutItem Class

Hierarchy

TCustomControl (VCL)

❶ TOvcComponent (OvcBase) 21

❷TOvcCollectible (OvcBase) 41

Tovclookoutitem (Ovclkout)

Properties

❶	About	❷	DisplayText	❷	Index
	Caption		IconIndex		LabelRect
❷	Collection		IconRect	❷	Name

33

Reference Section

Caption — property

```
property Caption : string
```

The display text for a folder item.

An item’s caption can be changed at run time by assigning a new value to Caption. Additionally, you can call TOvcLookOutBar.RenameItem to invoke the OvcLookOutBar’s in- place editor.

See also: TOvcLookOutBar.RenameItem

Description — property

```
property Description : string
```

Specifies a text description of a TOvcLookOutBar item.

Description can be used to store any additional text associated with a TOvcLookOutBar item. For example, you can use Description to display hint text for a particular item in your application’s status bar. See TOvcLookOutBar.OnMouseOverItem for an example.

See also: TOvcLookOutBar.OnMouseOverItem

IconIndex — property

```
property IconIndex : Integer
```

The index in the image list used to draw a folder item.

Set each folder item’s IconIndex property to the index corresponding to the image to display for that item. Images are contained in the OvcLookOutBar’s Images property.

See also: TOvcLookOutBar.Images

IconRect — run-time, read-only property

```
property IconRect : TRect
```

Contains the size and location of a folder item’s display icon.

See also: LabelRect

LabelRect — run-time, read-only property

```
property LabelRect : TRect
```

Contains the size and location of a folder’s label.

See also: IconRect

OvcNumberEdit and OvcDbNumberEdit

The edit components described in this chapter share a common ancestor, TOvcCustomEdit. TOvcCustomEdit (see page 1272) provides a common foundation and implements several key capabilities such as the ability to create and use a TOvcController component (see page 28) and the ability to create and associate a special label to itself.

- TOvcNumberEdit allows entry of numeric data and provides a pop-up calculator.
- TOvcDbNumberEdit is a number edit that can be attached to a field in a dataset.

TOvcCustomNumberEdit Class

The TOvcCustomNumberEdit class is the immediate ancestor of the TOvcNumberEdit component. It implements all of the methods and properties used by the TOvcNumberEdit component and is identical to TOvcNumberEdit except that no properties are published.

TOvcCustomNumberEdit is provided to facilitate creation of descendent table components. For property and method descriptions, see "TOvcNumberEdit Component" on page 1185.

Hierarchy

TCustomEdit (VCL)

  TOvcCustomEdit (OvcEditF) 1272

    TOvcEdPopup (OvcEdPop) 1275

      TOvcCustomNumberEdit (OvcEdClc)

33

TOvcNumberEdit Component

TOvcNumberEdit introduces an edit control for entry of numeric data. Additionally, a pop-up calculator (TOvcCalculator) can be displayed by pressing a button positioned at the right edge of the control. Several properties allow you to access the value displayed in the control in different ways — as an integer, as a floating-point value, or as a string.

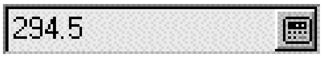


Figure 33.2: The NumberEdit component

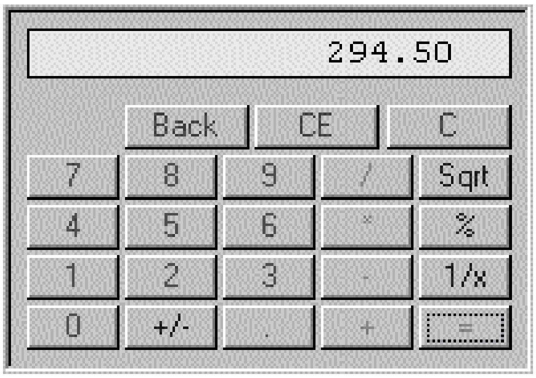


Figure 33.3: The NumberEdit component displaying the pop-up calculator

Several other properties allow control over how the pop-up calculator appears.

Caution: Because of the nature of the window used to display the pop-up, you must insure that the pop-up window is closed prior to closing the form. Failure to do so will result in run-time errors. The most common situation where this can occur is when the form has a default button that acts in some way to close the form. (Since it is the default button, pressing <ENTER> triggers the button's Click method without changing the focus.) Since changes in the focus cause the pop-up window to close, the easiest way to avoid the issue is to set the focus to the button from inside the button's OnClick event handler. Use the button's SetFocus method to do this.

Hierarchy

TCustomEdit (VCL)

 ❶TOvcCustomEdit (OvcEditF). 1272

  ❷TOvcEdPopup (OvcEdPop). 1275

   TOvcCustomNumberEdit (OvcEdClc) 1184

    TOvcNumberEdit (OvcEdClc)

❶These properties are documented in the TOvcEdit section on page 1273.

Properties

❶	About	❷	ButtonGlyph		PopupDecimals
	AllowIncDec		Calculator		PopupFont
	AsFloat	❶	Controller		PopupHeight
	AsInteger	❶	LabelInfo		PopupWidth
	AsString	❷	PopupAnchor		ReadOnly
❶	AttachedLabel		PopupColors	❷	ShowButton

Methods

❷	PopupClose	❷	PopupOpen

Reference Section

AllowIncDec property

```
property AllowIncDec : Boolean
```

Default: True

↳ Determines if pressing the plus or minus keys will increment and decrement the field value.

If AllowIncDec is True, pressing the plus key will increase the field value by one and pressing the minus key will reduce the field value by one. If AllowIncDec is False, pressing the plus and minus keys is ignored.

AsFloat run-time property

```
property AsFloat : Double
```

↳ Determines the field value as a floating-point value.

AsInteger run-time property

```
property AsInteger : LongInt
```

↳ Determines the field value as an integer value.

AsString run-time property

```
property AsString : string
```

↳ Determines the field value as text.

Values assigned using this property must be valid integer or floating-point numbers.

Calculator run-time, read-only property

```
property Calculator : TOvcCalculator
```

↳ Provides direct access to the properties, methods, and events of the popup calendar.

Warning: Care should be taken when altering any of the Calculator properties to insure that the new settings do not interfere with the “normal” operation of the popup calculator.

PopupClose virtual method

```
procedure PopupClose(Sender : TObject);
```

↳ Closes the popup calculator.

Calling PopupClose hides the popup calendar. If the calendar is not visible, no action is taken.

See also: PopupOpen

PopupColors property

```
property PopupColors : TOvcCalcColors
```

↳ Determines the colors used when displaying the popup calculator.

See “TOvcCalcColors Class” on page 509 for additional information about the individual color properties that can be set.

See also: TOvcCalculator.Colors

PopupDecimals property

```
property PopupDecimals : Integer
```

Default: 2

↳ Determines the number of decimal places the popup calculator displays.

PopupFont property

```
property PopupFont : TFont
```

↳ Determines the font used by the popup calculator.

PopupHeight property

```
property PopupHeight : Integer
```

Default: 140

↳ Determines the height of the popup calculator.

PopupOpen virtual method

```
procedure PopupOpen;
```

Displays the popup calendar.

See also: Popup Close

PopupWidth property

```
property PopupWidth : Integer
```

Default: 200

Determines the width of the popup calculator.

ReadOnly property

```
property ReadOnly : Boolean
```

Default: False

Determines if the number displayed in the edit control can be changed.

If ReadOnly is True, the displayed number cannot be edited. Otherwise, changes to the value are allowed.

TOvcCustomDbNumberEdit Class

The TOvcCustomDbNumberEdit class is the immediate ancestor of the TOvcDbNumberEdit component. It implements all of the methods and properties used by the TOvcDbNumberEdit component and is identical to TOvcDbNumberEdit except that no properties are published.

TOvcCustomTable is provided to facilitate creation of descendent data-aware number edit components. For property and method descriptions, see “TOvcDbNumberEdit Component” on page 1191.

Hierarchy

TTCustomEdit (VCL)

  TOvcCustomEdit (OvcEditF) 1272

    TOvcEdPopup (OvcEdPop) 1275

      TOvcCustomNumberEdit (OvcEdClc) 1184

        TOvcCustomDbNumberEdit (OvcDbNum)

TOvcDbNumberEdit Component

The TOvcDbNumberEdit is identical to the TOvcNumberEdit component with the additional ability of allowing it to be connected to and edit a field in a dataset.

Hierarchy

TCustomEdit (VCL)

TOvcCustomEdit (OvcEditF) 1272

❶TOvcEdPopup (OvcEdPop). 1275

❷TOvcCustomNumberEdit (OvcEdClc). 1184

TOvcDbNumberEdit (OvcDbNum)

❷These properties are documented in the TOvcNumberEdit section on page 1191

Properties

❷ AllowIncDec	❷ Controller	❷ PopupDecimals
❷ AsFloat	DataField	❷ PopupFont
❷ AsInteger	DataSource	❷ PopupHeight
❷ AsString	Field	❷ PopupWidth
AutoUpdate	❷ LabelInfo	❷ ReadOnly
❶ ButtonGlyph	❶ PopupAnchor	❶ ShowButton
❷ Calculator	❷ PopupColors	

Methods

❷ PopupClose	❷ PopupOpen

Reference Section

AutoUpdate property

```
property AutoUpdate : Boolean
```

Default: True

↳ Determines if the underlying data field is updated with the new value when the control loses the focus.

If AutoUpdate is True and the field contents contain a valid value, the associated dataset field is updated with the new value when the control loses the focus. If False, no action is taken.

DataField property

```
property DataField : string
```

↳ Identifies the field from which the component displays data.

See also: DataSource

DataSource property

```
property DataSource : TDataSource
```

↳ Specifies the data source component where the component obtains the data to display.

See also: DataField

Field read-only property

```
property Field : TField
```

↳ Returns the TField object to which the entry field component is linked.

Use the Field object to change the value of the data in the field programmatically.

Dates and Times:

This section provides a reference for many date and time routines used throughout Orpheus.

- TOvcDateEdit allows free-form date entry and provides a popup calendar.
- TOvcDbDateEdit is a date edit that can be attached to a field in a dataset.
- TOvcTimeEdit allows entry of time and duration values.
- TOvcDbTimeEdit is a time edit that can be attached to a field in a dataset.

The StDate unit provides routines to store dates and times in compact formats, to convert them to other forms, and to perform date and time arithmetic. StDate is a unit from TurboPower's SysTools library: a collection of essential classes, system routines, and components that take care of the low-level work behind application development.

TOvcCustomDateEdit Class

The TOvcCustomDateEdit class is the immediate ancestor of the TOvcDateEdit component. It implements all of the methods and properties used by the TOvcDateEdit component and is identical to TOvcDateEdit except that no properties are published.

TOvcCustomDateEdit is provided to facilitate your creation of descendent DateEdit components. For property and method descriptions, see “TOvcDateEdit Component” on page 1195.

Hierarchy

TCustomEdit (VCL)

 TOvcCustomEdit (OvcEditF) 1272

  TOvcEdPopup (OvcEdPop) 1275

   TOvcCustomDateEdit (OvcEdCal)

TOvcDateEdit Component

TOvcDateEdit is a specialized edit control that provides enhanced date entry capabilities and a popup calendar.

Date entry has notoriously been error prone. Areas of concern include international issues, year 2000 worries, or decisions on what to do with partial or erroneous entries. With the TOvcDateEdit you can be assured that your applications will suffer none of these ailments. Each date entry is checked for validity based on the current international settings. Partial dates are allowed and expanded to valid dates following the rules you choose as shown in Figure 33.4.

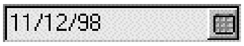


Figure 33.4: Partial dates

Your users will be able to enter free-form ambiguous dates like “7-7” and “5” and produce intuitive results. Your users can also enter text to specify a date relative to the current date, such as: “tomorrow”, “next Tuesday”, “yesterday”, “second Tuesday”, “2nd”, etc. All of this plus the ability to popup a monthly calendar! See “TOvcCalendar Component” on page 481 for details about the Orpheus calendar.

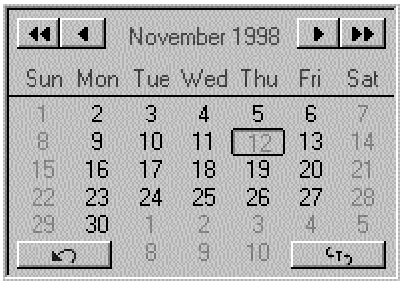


Figure 33.5: Date edit demo

The following table provides is a complete list of strings the TOvcDateEdit component understands:

Table 7.2: *TOvcDateEdit component recognized strings*

Command	**Notes**	**Action**
Next [day of week]	If no day of week specIfied, current day of week is assumed.	Displays the date for the next <day of week> day.
<day of week>	Next is assumed; may be abbreviated 1st 3 chars.	Displays the date for the next <day of week> day.
Last | Prev (Previous) [day of week]	If no day of week specIfied, current day of week is assumed.	Displays the date for the previous <day of week>.
First | 1st [day of week]	If no day of week specIfied, current day of week is assumed.	Displays the date for the first occurrence of the <day of week>.
Second | 2nd [day of week]	If no day of week specIfied, current day of week is assumed.	Displays the date for the second occurrence of the <day of week>.
Third | 3rd [day of week]	If no day of week specIfied, current day of week is assumed.	Displays the date for the third occurrence of the <day of week>.
Fourth | 4th [day of week]	If no day of week specIfied, current day of week is assumed.	Displays the date for the forth occurrence of the <day of week>.
Final [day of week]	If no day of week specIfied, current day of week is assumed.	Displays the date for the final occurrence of the <day of week>.
BOM | Begin		Displays the date for the first day of the current month.
EOM | End		Displays the date for the last day of the current month.
Yesterday		Displays yesterday's date.
Today		Displays today's date.
Tomorrow		Displays tomorrow's date.

Items in square brackets [] are optional. The pipe character (|) indicates an alternate command string.

Caution: Because of the nature of the window used to display the popup calendar, you must insure that the popup window is closed prior to closing the form. Failure to do so will result in run-time errors. The most common situation where this can occur is when the form has a default button that acts in some way to close the form. (Pressing <ENTER> invokes the default button's Click method without changing the focus.) Since changes in the focus cause the popup window to close, the easiest way to avoid the issue is to set the focus to the default button from inside the button's OnClick event handler. Use the button's SetFocus method to programatically set focus to the default button.

Hierarchy

TCustomEdit (VCL)

❶TOvcCustomEdit (OvcEditF) 1272

❷TOvcEdPopup (OvcEdPop). 1275

TOvcCustomDateEdit (OvcEdCal). 1194

TOvcDateEdit (OvcEdCal)

❶These properties are documented in the TOvcEdit section on page 1273.

❷These properties are documented in the TOvcNumberEdit section on page 1191

Properties

❶	About	❶	LabelInfo		PopupWeekStarts
	AllowIncDec	❷	PopupAnchor		PopupWidth
❶	AttachedLabel		PopupColors		ReadOnly
	Calendar		PopupDateFormat		RequiredFields
❶	Controller		PopupDayNameWidth	❷	ShowButton
	Date		PopupFont		TodayString
	Epoch		PopupHeight		
	ForceCentury		PopupOptions		

Methods

DateString	PopupClose
FormatDate	PopupOpen

Events

OnGetDate	OnPreParseDate	OnSetDate

Reference Section

AllowIncDec property

```
property AllowIncDec : Boolean
```

Default: True

↳ Determines if pressing the plus or minus keys will increment and decrement the field value.

If AllowIncDec is True, pressing the plus key will increase the field value by one day and pressing the minus key will reduce the field value by one day. If AllowIncDec is False, pressing the plus and minus keys has no effect.

Calendar run-time, read-only property

```
property Calendar : TOvcCalendar
```

↳ Provides direct access to the properties, methods, and events of the popup calendar.

Warning. Care should be taken when altering any of the Calendar properties to insure that the new settings do not interfere with the “normal” operation of the popup calendar.

Date run-time property

```
property Date : TDateTime
```

↳ Determines the value stored and edited in the control.

DateString method

```
function DateString (const Mask : string) : string;
```

↳ Returns the currently displayed date value as a formatted string.

DateString calls TOvcIntlSup.DateToDateString, passing Mask and the current date value as the parameters. See “TOvcIntlSup Class” on page 344 for information about the DateToDateString method.

See also: FormatDate

Epoch property

```
property Epoch : Integer
```

Default: 0

↳ Specifies the starting year of a 100-year period.

The default of zero means that the value specified in Controller.Epoch is used. See “The Epoch property” on page 321 for more information.

See also: TOvcController.Epoch

ForceCentury property

```
property ForceCentury : Boolean
```

Default: False

↳ Determines if the date mask is forced to display four digit years.

If ForceCentury is True, the date mask obtained from Windows is modified to produce a date that includes display of a four digit year. If False, the date mask obtained from Windows is used without modification.

FormatDate virtual method

```
function FormatDate (Value : TDateTime) : string;
```

↳ Responsible for correctly formatting a date into text suitable for display.

FormatDate uses TOvcIntlSup.InternationalDate and TOvcIntlSup.DateToDateString to format the date value. See “TOvcIntlSup Class” on page 344 for information about the InternationalDate and DateToDateString methods.

See also: DateString

OnGetDate event

```
property OnGetDate : TOvcGetDateEvent
```

```
TOvcGetDateEvent = procedure(
  Sender : TObject; var Value : string) of object;
```

↳ Defines an event handler that allows you to modify the date that is displayed in the control.

Sender is the instance of the component and Value is the text value of the date after it has already been parsed and expanded as necessary. You can change Value to represent any valid date. The format of the date must match the format used by the control. That is, it must use the Windows date mask and date separator characters and must honor the setting of the ForceCentury property.

See also: ForceCentury, OnPreParseDate

OnPreParseDate event

```
property OnPreParseDate : TOvcPreParseDateEvent
```

```
TOvcPreParseDateEvent =  procedure(
  Sender : TObject; var Value : string) of object;
```

↳ Defines an event handler that allows you to intercept and interpret the text the user has entered into the field.

Sender is the instance of the component and Value is the text the user has entered into the field. The text has not been processed to replace strings such as “today” or “yesterday”.

This event is provided primarily so that you can enhance the date edit’s capabilities. You can use this event to make this component “understand” additional text strings. For example, you might want to provide support for entering strings like “next monday” or “jane’s birthday”, etc.

See also: OnGetDate

OnSetDate event

```
property OnSetDate : TNotifyEvent
```

↳ Defines an event handler that is called each time the value of the displayed date changes.

PopupClose virtual method

```
procedure PopupClose (Sender : TObject);
```

↳ Closes the popup calendar.

Calling PopupClose hides the popup calendar. If the calendar is not visible, no action is taken.

See also: PopupOpen

PopupColors property

```
property PopupColors : TOvcCalColors
```

↳ Determines the colors used when displaying the popup calendar.

See “TOvcColors Class” on page 1186 for additional information about the individual color properties that can be set.

See also: TOvcCalendar.Colors

PopupDateFormat property

```
property PopupDateFormat : TOvcDateFormat
```

```
TOvcDateFormat = (dfShort, dfLong);
```

Default: dfLong

↳ Determines whether the short date format (dfShort) or the long date format (dfLong) is used.

PopupDayNameWidth property

```
property PopupDayNameWidth : TOvcDayNameWidth
```

```
TOvcDayNameWidth = 1..3;
```

Default: 3

↳ Determines the width in characters of the name of the day displayed in the popup calendar.

PopupFont property

```
property PopupFont : TFont
```

↳ Determines the font used for the popup calendar.

See also: TOvcCalendar.Font

PopupHeight property

```
property PopupHeight : Integer
```

↳ Determines the height of the popup calendar.

PopupOpen virtual method

```
procedure PopupOpen;
```

↳ Displays the popup calendar.

See also: Popup Close

PopupOptions property

```
property PopupOptions : TOvcCalDisplayOptions
```

```
TOvcCalDisplayOptions = set of TOvcCalDisplayOption;
```

```
TOvcCalDisplayOption = (cdoShortNames, cdoShowYear,
  cdoShowInactive, cdoShowRevert, cdoShowToday );
```

Default: [cdoShortNames, cdoShowYear, cdoShowInactive, cdoShowRevert,

cdoShowToday];

↳ Determines which options the popup calendar uses.

For additional information about the calendar options, see “TOvcCalendar Component” on page 481.

See also: TOvcCalendar.Options

PopupWeekStarts property

```
property PopupWeekStarts : TOvcDayType
```

```
TOvcDayType = (dtSunday, dtMonday, dtTuesday,
  dtWednesday, dtThursday, dtFriday, dtSaturday);
```

Default: dtSunday

↳ Determines which day of the week the popup calendar displays as the first day of the week.

See also: TOvcCalendar.WeekStarts

PopupWidth property

```
property PopupWidth : Integer
```

↳ Determines the width of the popup calendar.

ReadOnly property

```
property ReadOnly : Boolean
```

Default: False

↳ Determines if the date displayed in the edit field can be changed.

If ReadOnly is True, the displayed date cannot be edited. Otherwise, changes to the date are allowed.

RequiredFields property

```
property RequiredFields : TOvcRequiredDateFields
```

```
TOvcRequiredDateFields = set of TOvcRequiredDateField;
```

```
TOvcRequiredDateField = (rfYear, rfMonth, rfDay);
```

Default: [rfMonth, rfDay]

↳ Determines which of the three possible date fields the user must enter.

By default, only the month and day fields are required in which case the year is assumed to be the current year. In general, if a field isn't required, the current date is used to determine the value for the missing portion.

TodayString property

```
property TodayString : string
```

Default: (the Windows date separator)

↳ Determines the text that is interpreted as meaning, “enter the current date here”.

By default, TodayString is the character used to separate the individual date fields (day, month, and year). Entering the TodayString and exiting the field will cause the current date to be automatically entered. For example, entering “/” and leaving the field will display the current date.

TOvcCustomDbDateEdit Class

The TOvcCustomDbDateEdit class is the immediate ancestor of the TOvcDbDateEdit component. It implements all of the methods and properties used by the TOvcDbDateEdit component and is identical to TOvcDbDateEdit except that no properties are published.

TOvcCustomDbDateEdit is provided to facilitate creation of descendent table components. For property and method descriptions, see “TOvcDateEdit Component” on page 1195.

TCustomEdit (VCL)

❶TOvcCustomEdit (OvcEditF) 1272

❷TOvcEdPopup (OvcEdPop). 1275

TOvcCustomDateEdit (OvcEdCal). 1194

TOvcDateEdit (OvcEdCal). 1195

TOvcCustomDbDateEdit (OvcDbDat)

TOvcDbDateEdit Component

The TOvcDbDateEdit is identical to the TOvcDateEdit component with the additional ability of allowing it to be connected to and edit a field in a dataset.

Hierarchy

TCustomEdit (VCL)

❶TOvcCustomEdit (OvcEditF). 1272

❷TOvcEdPopup (OvcEdPop). 1275

TOvcCustomDateEdit (OvcEdCal) 1194

❸ TOvcDateEdit (OvcEdCal). 1195

TOvcCustomDbDateEdit (OvcDbDat). 1205

TOvcDbDateEdit (OvcDbDat)

❶These properties are documented in the TOvcEdit section on page 1273.

❷These properties are documented in the TOvcNumberEdit section on page 1191

Properties

❶	About	❸	Epoch	❸	PopupHeight
❸	AllowIncDec		Field	❸	PopupOptions
❶	AttachedLabel	❸	ForceCentury	❸	PopupWeekStarts
	AutoUpdate	❶	LabelInfo	❸	PopupWidth
❸	Calendar	❷	PopupAnchor	❸	ReadOnly
❶	Controller	❸	PopupColors	❸	RequiredFields
	DataField	❸	PopupDateFormat	❷	ShowButton
	DataSource	❸	PopupDayNameWidth	❸	TodayString
❸	Date	❸	PopupFont		

Methods

❸	DateString	❸	FormatDate

Reference Section

AutoUpdate — property

```
property AutoUpdate : Boolean
```

Default: True

↳ Determines if the underlying data field is updated with the new date when the control loses the focus.

If AutoUpdate is True and the field contents contain a valid date, the associated dataset field is updated with the new value when the control loses the focus. If False, no action is taken.

DataField — property

```
property DataField : string
```

↳ Identifies the field from which the component displays data.

See also: DataSource

DataSource — property

```
property DataSource : TDataSource
```

↳ Specifies the data source component where the component obtains the data to display.

See also: DataField

Field — run-time, read-only property

```
property Field : TField
```

↳ Returns the TField object to which the entry field component is linked.

Use Field to change the value of the data in the field programmatically.

TOvcCustomTimeEdit Class

The TOvcCustomTimeEdit class is the immediate ancestor of the TOvcTimeEdit component. It implements all of the methods and properties used by the TOvcTimeEdit component and is identical to TOvcTimeEdit except that no properties are published.

TOvcCustomTimeEdit is provided to facilitate creation of descendent components. For property and method descriptions, see “TOvcTimeEdit Component” on page 1209.

Hierarchy

TCustomEdit (VCL)

  TOvcCustomEdit (OvcEditF) 1272

    TOvcCustomTimeEdit (OvcEdTim)

TOvcTimeEdit Component

TOvcTimeEdit does for time and duration entries what TOvcDateEdit does for date entry—greatly simplifies it. With the Time Edit control you can enter something like “2” and have it interpreted as 2:00 PM (or 2:00 AM). You can set up the field to work with duration of time instead of time of day and have “2h 3m 33s” interpreted as 2 hours 3 minutes 33 seconds.

TOvcTimeEdit's relaxed time entry capabilities allow your users to concentrate on their business rather than remembering difficult data entry rules.

The ExTime example project demonstrates many of TOvcTimeEdit's features.

Hierarchy

TCustomEdit (VCL)

    ❶ TOvcCustomEdit (OvcEditF) 1272

        TOvcCustomTimeEdit (OvcEdTim)

            TOvcTimeEdit (OvcEdTim)

❶These properties are documented in the TOvcEdit section on page 1273.

Properties

❶ About	❶ Controller	ShowSeconds
AsDateTime	DefaultToPM	ShowUnits
AsHours	DurationDisplay	TimeMode
AsMinutes	❶ LabelInfo	UnitsLength
AsSeconds	NowString	
❶ AttachedLabel	PrimaryField	

Methods

FormatTime

Events

OnGetTime	OnPreParseTime	OnSetTime

Reference Section

AsDateTime run-time property

```
property AsDateTime : TDateTime
```

Determines the field value as a VCL TDateTime type.

If the field is operating in duration mode, AsDateTime can contain values greater than 24 hours. This can be accomplished by using the number of days and the EncodeTime routine. For example, to set a duration value of 76 hours and 10 minutes:

```
  OvcTimeEdit1.AsDateTime :=
    EncodeDate(0, 0, 3) + EncodeTime(4, 10, 0);
```

AsHours run-time property

```
property AsHours : Integer
```

Determines the field value in hours.

AsMinutes run-time property

```
property AsMinutes : Integer
```

Determines the field value in minutes.

AsSeconds run-time property

```
property AsSeconds : Integer
```

Determines the field value in seconds.

DefaultToPM property

```
property DefaultToPM  : Boolean
```

Default: False

Determines if times as PM entries.

If DefaultToPM is True, times less than 12 with no indication of AM or PM are treated as PM entries. Otherwise, they are treated as AM entries.

DurationDisplay property

```
property DurationDisplay : TOvcDurationDisplay
```

```
TOvcDurationDisplay = (ddHMS, ddHM, ddMS, ddHHH, ddMMM, ddSSS);
```

Default: ddHMS

Determines which format is used to display the value as a time duration value.

DurationDisplay is only used if TimeMode is set to tmDuration.

Possible values for DurationDisplay are:

Value	**Description**
ddHMS	hours, minutes, and seconds
ddHM	hours and minutes
ddMS	minutes and seconds
ddHHH	just hours
ddMMM	just minutes
ddSSS	just seconds

See also: TimeMode

FormatTime virtual method

```
function FormatTime(Value : TDateTime) : string;
```

Default: FormatTime

Responsible for correctly formatting the passed time Value into text for display.

FormatTime uses TOvcIntlSup.InternationalTime and TOvcIntlSup.TimeToTimeString to format the time of day value. See “TOvcIntlSup Class” on page 344 for information about the InternationalTime and TimeToTimeString methods.

NowString property

```
property NowString : string
```

Default: (the Windows time separator)

Determines the text that is interpreted as meaning, “enter the current time here”.

By default, the NowString is the character used to separate the individual time fields (hour, minute, and second). Entering the NowString and exiting the field will cause the current time to be automatically entered. For example, entering “:” and leaving the field will display the current time.

See also: FormatTime

OnGetTime event

```
property OnGetTime : TOvcGetTimeEvent
```

```
TOvcGetTimeEvent = procedure(
  Sender : TObject; var Value : string) of object;
```

OnGetTime defines an event handler that allows you to modify the time that is displayed in the control.

Sender is the instance of the component and Value is the text value of the time after it has already been parsed and expanded as necessary. You can change Value to represent any valid time. The format of the time must be an acceptable format for the field (one that can be converted to a time or duration).

See also: OnPreParseTime

OnPreParseTime event

```
property OnPreParseTime : TOvcPreParseTimeEvent
```

```
TOvcPreParseTimeEvent = procedure(
  Sender : TObject; var Value : string) of object;
```

Defines an event handler that allows you to intercept and interpret the text the user has entered into the field.

Sender is the instance of the component and Value is the text the user has entered into the field. The text has not been processed.

This event is provided mainly so that you can enhance the editor's capabilities. You can use this event to make the component understand additional text strings. For example, you might want to provide support for entering strings like "+ 50 minutes" or "- 10 seconds", etc.

See also: OnGetTime

OnSetTime event

```
property OnSetTime : TNotifyEvent
```

↳ Defines an event handler that is called each time the value of the displayed time changes.

PrimaryField property

```
property PrimaryField : TOvcTimeField
```

```
TOvcTimeField = (tfHours, tfMinutes, tfSeconds);
```

Default: tfHours

↳ Determines how to interpret values that are entered without units (hours, minutes, or seconds).

The primary field value is used to decide what time unit to assign to ambiguous entries. For example, an entry of "5" could mean 5 as an hour, minute, or second. Using PrimaryField, the control can make assumptions as to what the value means.

ShowSeconds property

```
property ShowSeconds : Boolean
```

Default: False

↳ Determines if the displayed time value includes seconds.

ShowUnits property

```
property ShowUnits : Boolean
```

Default: True

↳ Determines if the displayed time or duration value includes time units (hours, minutes, or seconds)

The number of characters displayed for the unit is determined by the UnitsLength property.

See also: UnitsLength

TimeMode property

```
property TimeMode : TOvcTimeMode
```

```
TOvcTimeMode = (tmClock, tmDuration);
```

Default: tmClock

↳ Determines how values are intrepreted and displayed.

If TimeMode is tmClock, all values are assumed to be time-of-day values and are forced to be in the range of 00:00:00 to 23:59:59.

If TimeMode is tmDuration, entry and display allows for hours, minutes, and seconds to be any positive value. For example, entries such as “29 hours” or “433 minutes” are possible.

See also: DurationDisplay

UnitsLength property

```
property UnitsLength : Integer
```

Default: 1

↳ Determines the number of characters to use when displaying the time units.

TOvcCustomDbTimeEdit Class

The TOvcCustomDbTimeEdit class is the immediate ancestor of the TOvcDbTimeEdit component. It implements all of the methods and properties used by the TOvcDbTimeEdit component and is identical to TOvcDbTimeEdit except that no properties are published.

TOvcCustomDbTimeEdit is provided to facilitate creation of descendent time edit components. For property and method descriptions, see “TOvcDbTimeEdit Component” on page 1216.

Hierarchy

TCustomEdit (VCL)

- ❶ TOvcCustomEdit (OvcEditF) 1272
  - ❷TOvcCustomTimeEdit (OvcEdTim) 1208
    - TOvcTimeEdit (OvcEdTim). 1209
      - TOvcCustomDbTimeEdit (OvcDbTim)

33

TOvcDbTimeEdit Component

The TOvcDbTimeEdit is identical to the TOvcTimeEdit Component with the additional ability of allowing it to be connected to and edit a field in a dataset.

Hierarchy

TCustomEdit (VCL)

- ❶ TOvcCustomEdit (OvcEditF) 1272
  - ❷TOvcCustomTimeEdit (OvcEdTim) 1208
    - TOvcTimeEdit (OvcEdTim) 1209
      - TOvcCustomDbTimeEdit (OvcDbTim) .1215
        - TOvcDbTimeEdit (OvcDbTim)

❶These properties are documented in the TOvcEdit section on page 1273.

❷These properties are documented in the TOvcNumberEdit section on page 1191.

Properties

❷ About	❶ Controller	❷ NowString
❷ AsDateTime	DataField	❷ PrimaryField
❷ AsHours	DataSource	❷ ShowSeconds
❷ AsMinutes	❷ DefaultToPM	❷ ShowUnits
❷ AsSeconds	❷ DurationDisplay	❷ TimeMode
❶ AttachedLabel	Field	❷ UnitsLength
AutoUpdate	❶ LabelInfo	

Methods

❷ FormatTime

Events

❷ OnGetTime	❷ OnPreParseTime	❷ OnSetTime

Reference Section

AutoUpdate property

```
property AutoUpdate : Boolean
```

Default: True

↳ Determines if the underlying data field is updated with the new date when the control loses the focus.

If AutoUpdate is True and the field contents contain a valid date, the associated dataset field is updated with the new value when the control loses the focus. If False, no action is taken.

DataField property

```
property DataField : string
```

↳ Identifies the field from which the component displays data.

See also: DataSource

DataSource property

```
property DataSource : TDataSource
```

↳ Specifies the data source component where the component obtains the data to display.

See also: DataField

Field run-time, read-only property

```
property Field : TField
```

↳ Returns the TField object to which the entry field component is linked.

Use the Field object to change the value of the data in the field programmatically.

Array Editors:

The Orpheus array editors allow editing of array-like structures in what looks like a listbox containing editable cells. There are data-aware versions of the array editors that connect to a data source and allow editing of a database field in a columnar fashion. The array editors are very similar to the TOvcVirtualListbox component (see page 689). They allow you to edit the active list item just as you edit the simple, picture, or numeric entry fields (or the respective data-aware versions of the entry fields) described in “Chapter 11: Validated Data Entry Components” on page 361.

The array editors consist of three components derived from a base class:

- TOvcBaseArrayEditor class
- TOvcSimpleArrayEditor component
- TOvcPictureArrayEditor component
- TOvcNumericArrayEditor component

The data-aware array editors also consist of three components derived from a base class:

- TOvcBaseDbArrayEditor class
- TOvcDbSimpleArrayEditor component
- TOvcDbPictureArrayEditor component
- TOvcDbNumericArrayEditor component

The array editors get their in-place editing capabilities from their corresponding entry field components. Any standard data type that can be stored in an array or viewed as an array-like structure (database files, linked lists, etc.) can be edited using the array editors. You can even edit data structures that contain dates and times.

The standard (non data-aware) array editors get their data by calling an event handler (OnGetItem) that you provide. Actually, the array editor obtains a pointer to the data and uses the pointer to update the listbox display and the data itself when it is edited. The array editors do not store a copy of the data. Because of this, there are essentially no capacity limitations imposed by the array editors. The only limit is imposed by the index, which is a long integer. This means that the maximum number of items that can be managed by an array editor is slightly over 2 billion.

When a standard array editor needs to paint a row, it calls the method assigned to the OnGetItem event to get the address of the data. If the cell is modified, this address is used to store the new value just before the focus leaves the edit cell.

After you drop a standard array editor on your form, you must perform the following steps before the array editor can be used:

1. Set the DataType property to match the type of data in your data structure.
2. Set the NumItems property to reflect the number of elements in your data structure.
3. Assign a method to the OnGetItem event that returns the address of the requested list element. The address returned by your method must point to a value that has the same data type as the DataType property.

When your program changes a value in the data structure, the standard array editor does not automatically update its display. The standard array editor has no way of detecting the change, so you must call the Refresh method to force the array editor to redraw the display.

These issues are handled automatically when you use the data-aware array editors.

TOvcBaseArrayEditor Class

The TOvcBaseArrayEditor class is the immediate ancestor of the three array editor components. It provides methods and properties common to all array editors.

Array Editor Commands

The following commands are available in any Orpheus array editor. The commands and key assignments described here are available when you use the “Default” or “WordStar” command tables (several command tables can be used at the same time). See “TOvcCommandProcessor Class” on page 50 for information on how to customize the command tables.

Many of the commands mentioned here work differently when used in a numeric array editor. The reason for the difference in behavior is that there is no caret movement in a numeric edit cell. The caret always remains in a fixed position at the right of the cell contents. The differences are noted in the following descriptions.

Table 33.3: *Commands and their functions*

Command	**Default**	**WordStar**
ccBack	<BkSp>	<CtrlH>
	Delete the character to the left of the caret.	
ccCompleteDate	not assigned	not assigned
	Insert the current day, month, or year (depending on the position of the caret) in an array editor cell.	
ccCompleteTime	not assigned	not assigned
	Insert the current hour, minute, or second (depending on the position of the caret) in an array editor cell.	
ccCopy	<CtrlIns>, <CtrlC>	not assigned
	Copy the selected text to the clipboard.	
ccCtrlChar	not assigned	<CtrlP>
	Treat the next character as a control character.	
ccCut	<ShiftDel>, <CtrlX>	not assigned
	Delete the selected text after copying it to the clipboard.	
ccDec	not assigned	not assigned
	Decrement the current value by one ordinal amount.	
ccDel	<Del>	<CtrlG>

Table 33.3: *Commands and their functions (continued)*

	Delete the current character. Or, if text is selected, delete the selection.	
ccDelBol	not assigned	not assigned
	Delete text from the caret position to the beginning of the edit cell. This command does nothing in a numeric array editor.	
ccDelEol	not assigned	<ShiftCtrlY>, <CtrlQ><Y>
	Delete text from the caret position to the end of the edit cell. This command does nothing in a numeric array editor.	
ccDelLine	not assigned	<CtrlY>
	Delete the edit cell contents.	
ccDelWord	not assigned	<CtrlT>
	Delete the word to the right of the caret. This command does nothing in a numeric array editor.	
ccDown	<Down>	<CtrlX>
	Move the focus down to the next list item.	
ccEnd	<End>	<CtrlQ><D>
	Move the caret to the end of the edit cell. This command does nothing in a numeric array editor.	
ccExtendEnd	<ShiftEnd>	not assigned
	Extend the selection to the end of the edit cell.	
ccExtendHome	<ShiftHome>	not assigned
	Extend the selection to the start of the edit cell.	
ccExtendLeft	<ShiftLeft>	not assigned
	Extend the selection to the left by one character.	
ccExtendRight	<ShiftRight>	not assigned
	Extend the selection to the right by one character.	
ccExtWordLeft	<CtrlShiftLeft>	not assigned
	Extend the selection to the left by one word.	
ccExtWordRight	<CtrlShiftRight>	not assigned
	Extend the selection to the right by one word.	
ccFirstPage	<CtrlHome>	not assigned
	Move the focus to the first list item.	
ccHome	<Home>	<CtrlQ><S>
	Move the caret to the beginning of the edit cell. This command does nothing in a numeric array editor.	

Table 33.3: *Commands and their functions (continued)*

ccInc	not assigned	not assigned
	Increment the value of the edit cell.	
ccIns	<Ins>	<CtrlV>
	Toggle insert mode. The caret reflects the current mode. By default, a solid line indicates insert mode; a block indicates overwrite mode. Although this command is recognized by numeric array editors, it doesn’t do anything except change the caret.	
ccLastPage	<CtrlEnd>	not assigned
	Move the focus to the last list item.	
ccLeft	<Left>	<CtrlS>
	Move the caret left one character. If the caret is already at the beginning of the edit cell (or this is a numeric array editor) and AutoAdvanceLeftRight is *true*, the caret is moved to the previous component, just as it is if <ShiftTab> is pressed.	
ccNextPage	<PgDn>	<CtrlC>
	Scroll the list down one page.	
ccPaste	<ShiftIns>, <CtrlV>	not assigned
	Paste the text from the clipboard.	
ccPrevPage	<PgUp>	<CtrlR>
	Scroll the list up one page.	
ccRestore	<AltBkSp>, <CtrlZ>	<CtrlQ><L>
	Restore the original contents of the edit cell (i.e., the value the edit cell contained when the array editor last had the focus).	
ccRight	<Right>	<CtrlD>
	Move the caret right one character. If the caret is already at the end of the edit cell(or this is a numeric array editor) and the AutoAdvanceLeftRight property is *true*, the caret is moved to the next component, just as it is if <Tab> is pressed.	
ccUp	<Up>	<CtrlE>
	Move the focus to the previous list item.	
ccWordLeft	<CtrlLeft>	<CtrlA>

Table 33.3: *Commands and their functions (continued)*

	Move the caret left one word. This command does nothing in a numeric array editor.	
ccWordRight	<CtrlRight>	<CtrlF>
	Move the caret right one word. This command does nothing in a numeric array editor.	

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomContol (OvcBase) 16

❷ TOvcCustomControlEx (OvcBase) 19

TOvcBaseArrayEditor (OvcAE)

Properties

❶ About	ItemIndex	RangeLo
❶ AttachedLabel	❶ LabelInfo	RowHeight
BorderStyle	LineColor	TextMargin
❷ Controller	NumItems	UseScrollBar
DisabledColors	PadChar	
HighlightColors	RangeHi	

Methods

WriteCellValue

Events

❶ AfterEnter	OnGetItem	OnUserCommand
❶ AfterExit	OnGetItemColor	OnUserValidation
OnChange	❶ OnMouseWheel	
OnError	OnSelect	

Reference Section

BorderStyle property

```
property BorderStyle : TBorderStyle
```

Default: bsSingle

↳ Determines the style used when drawing the border.

The possible values are bsSingle (a single line border is drawn around the component) or bsNone (no border line). The TBorderStyle type is defined in the VCL's Controls unit.

DisabledColors property

```
property DisabledColors : TOvcColors
```

↳ Determines the colors used to display the items in the array when the array editor is disabled.

HighlightColors property

```
property HighlightColors : TOvcColors
```

Default: clHighlightText and clHighlight

↳ Determines the colors used to display the highlighted portion of the cell.

See "TOvcColors Class" on page 1186 for more information. Refer to the VCL's Color property for a list of available values and their meanings.

The following example displays highlighted text in blue with a white background.

```
  AE1.HighlightColors.TextColor := clBlue;
  AE1.HighlightColors.BackColor := clWhite;
```

ItemIndex run-time property

```
property ItemIndex : LongInt
```

↳ Determines which array element is the selected cell.

The ItemIndex property (available only at run time) is used to determine which array element currently has the input focus and to move the focus to a different array element. The array list is scrolled as necessary to make the newly assigned index item visible.

33

The array editor assumes that the array is 0-based; therefore, valid index values are 0 through NumItems-1. If a value outside this range is assigned to ItemIndex, it is ignored.

See also: NumItems

LineColor property

```
property LineColor : TColor
```

Default: clSilver

Determines the color of the row divider lines.

NumItems property

```
property NumItems : LongInt
```

Default: depends on the height of the array editor

The maximum number of items maintained by the array editor.

When scrolling through the array editor's list of values, NumItems determines when the last item has been displayed. NumItems can be changed at any time to reflect additions or deletions in the underlying data structure. This forces the array editor to repaint itself.

The minimum value for NumItems is equivalent to the number of rows that can be visible in the array editor for its current height.

OnChange event

```
property OnChange : TNotifyEvent;
```

Defines an event handler that is called when the contents of the active edit cell are modified.

The method assigned to the OnChange event is called when the active edit cell is modified. Sender is the class instance of the array editor containing the active cell. TNotifyEvent is defined in the VCL's Classes unit.

The following example causes ArrayEditor1Change to be called when the contents of the active cell change. A beep is produced for each change in the active edit cell.

```
procedure Form1.ArrayEditor1Change(Sender : TObject);
begin
  MessageBeep(0);
end;
...
OnChange := Form1.ArrayEditor1Change;
```

OnError event

```
property OnError : TValidationErrorEvent;
```

```
TValidationErrorEvent = procedure(Sender : TObject;
  ErrorCode : Word; ErrorMessage : string) of object;
```

Defines an event handler that is called when an array editor error occurs.

The method assigned to the OnError event is called when an error occurs. If no method is assigned to this event, the event handler assigned to the array editor's controller OnError event is called.

The possible values for ErrorCode are:

Value	**Description**
oeRangeError	The entered value is not within the accepted range.
oeInvalidNumber	An invalid number was entered in a numeric cell.
oeInvalidDate	An invalid date was entered in a date cell.
oeInvalidTime	An invalid time was entered in a time cell.
oeCustomError	The first error code reserved for user applications.

All error values less than oeCustomError are reserved for use by Orpheus.

ErrorMessage is a short description of the error.

See "TOvcController Component" on page 28 for additional information about error handling.

OnGetItem event

```
property OnGetItem : TGetItemEvent;
```

```
TGetItemEvent = procedure(Sender : TObject;
  Index : LongInt; var Value : Pointer) of object;
```

Defines an event handler that is called to get data addresses for the array editor.

The method assigned to the OnGetItem event is called to get a pointer (assigned to Value) to the data corresponding to the Index item of the array. If no method is assigned, the array editor displays sample data in all rows and ignores changes made to the edit cells.

The following example assumes these property settings:

DataType	**sftString**
MaxLength	15
NumItems	6
OnGetItem	GetItem

It returns the address of the Index item in the PizzaToppings string array.

```
var
  PizzaToppings : array[0..5] of string[15];
...
Form1.ArrayEditor1GetItem(Sender : TObject; Index : LongInt;
                          var Value : Pointer);
begin
  Value := @PizzaToppings[Index];
end;
```

OnGetItemColor event

```
property OnGetItemColor : TGetItemColorEvent;
```

```
TGetItemColorEvent = procedure(Sender : TObject;
  Index : LongInt; var FG, BG : TColor) of object;
```

↳ Defines an event handler that is called to get the colors used to draw an array element.

The method assigned to the OnGetItemColor event is called by the array editor to get the colors to use when displaying the item specified by Index. On entry to the event handler, FG and BG contain the default foreground and background colors. Set FG and BG to the desired colors. Sender is the class instance of the array editor.

This event is not called to get colors for the active cell. The colors for the active cell are obtained from the current Font.Color and Color properties.

The following example displays every other array element using red as the text color.

```
procedure
  Form1.ArrayEditor1GetItemColor(
    Sender : TObject; Index : LongInt; var FG, BG : TColor);
begin
  if Odd(Index) then
    FG := clRed;
end;
...
OnGetItemColor := Form1.ArrayEditor1GetItemColor;
```

OnSelect event

```
property OnSelect : TSelectEvent

TSelectEvent = procedure(
  Sender : TObject; NewIndex : LongInt) of object;
```

Defines an event handler that is called to provide notification that a new item was selected.

NewIndex is the index value of the newly selected item. Sender is the class instance of the array editor.

OnUserCommand event

```
property OnUserCommand : TUserCommandEvent

TUserCommandEvent = procedure(
  Sender : TObject; Command : Word) of object;
```

Defines an event handler that is called when a user-defined command is entered.

The method assigned to the OnUserCommand event is called when you enter the key sequence corresponding to one of the user-defined commands, (ccUser1, ccUser2, etc.).

For example, suppose you add the <CtrlD> key sequence to one of the active command tables and assign it to the ccUser1 command. When you press <CtrlD>, the method assigned to the OnUserCommand event is called and ccUser1 is passed in Command.

See “TOvcCommandProcessor Class” on page 50 for additional information about user-defined commands and command tables.

OnUserValidation event

```
property OnUserValidation : TUserValidationEvent

TUserValidationEvent = procedure(
  Sender : TObject; var ErrorCode : Word) of object;
```

Defines an event handler that is called when cell validation is required.

The method assigned to the OnUserValidation event is called after the default validation is performed on the current array editor cell. Validation is performed when the array editor loses the focus.

ErrorCode contains the result of the validation. If it is zero, then the default validation didn’t find any errors and you can perform any additional validation that is required. The error constants are defined in the OnError event.

If the OnUserValidation event handler determines that the cell contents are invalid, assign a value to ErrorCode before exiting the method. The error value should be greater than or equal to oeCustomError (32768).

Caution: The event handler assigned to the OnUserValidation event must not perform any actions that could cause the focus to change. This could cause Windows to lose track of where the focus is and could cause other problems as well. Do not display a form or dialog reporting the error—that should only be done in a method assigned to the components OnError event or the TOvcController's OnError event. The OnUserValidation event should only do what is necessary to confirm that the cell contents are valid.

See "TOvcController Component" on page 28 for additional information about error handling for multiple components.

PadChar property

```
property PadChar : AnsiChar
```

Default: ASCII 32 (space)

The character that is used to fill the blank portion of the display string.

RangeHi property

```
property RangeHi : string
```

Default: depends on the array editor's data type

The upper limit for the range of allowable cell values.

This property is intended primarily for use at design time. However, it can be used at run time by assigning a string version of the desired range to the property. Range settings are ignored for string type array editors.

You can directly enter a range limit in the RangeHi edit field in the Object Inspector or you can invoke the Range Property Editor, which allows entry and validation of the range limit into a cell of the appropriate data type.

The following example sets an upper limit of 200 for the cells.

```
  {AE1 is a simple array editor with a data type of Byte}
  AE1.RangeHi := '200';
```

See also: RangeLo

RangeLo property

```
property RangeLo : string
```

Default: depends on the array editor's data type

↳ The lower limit for the range of allowable cell values.

This property is intended primarily for use at design time. However, it can be used at run time by assigning a string version of the desired range to the property. Range settings are ignored for string type array editors.

You can directly enter a range limit in the RangeLo edit field in the Object Inspector or you can invoke the Range Property Editor, which allows entry and validation of the range limit into a cell of the appropriate data type.

The following example sets a lower limit of 10 for all cells.

```
  {AE1 is a simple array editor with a data type of Byte}
  AE1.RangeLo := '10';
```

See also: RangeHi

RowHeight property

```
property RowHeight : Integer
```

Default: dependent on the parent font

↳ The height of the array editor cells in pixels.

The row height is automatically determined based on the current font. Assigning a new value to the RowHeight property causes the array editor to resize each editing cell. The array editor is also resized if necessary to avoid displaying a partial row.

TextMargin property

```
property TextMargin : Integer
```

Default: 2

↳ Determines the cell's display indent.

The TextMargin property controls the left indent in pixels for the simple and picture array editors. For the numeric array editor, this property is the right margin since the painting is done from right to left. The minimum value for TextMargin is 2.

UseScrollBar property

```
property UseScrollBar : Boolean
```

Default: True

Determines if a vertical scrollbar is displayed.

If UseScrollBar is True, a vertical scrollbar is displayed at the right of the array editor.

WriteCellValue method

```
function WriteCellValue : LongInt;
```

Forces the contents of the current cell to be saved.

TOvcSimpleArrayEditor Component

The TOvcSimpleArrayEditor component is derived from a TOvcBaseArrayEditor. The TOvcSimpleArrayEditor edit cells are just TOvcSimpleFields. The PictureMask, MaxLength, and DecimalPlaces properties behave just like the TOvcSimpleField properties of the same name.

Example

This example shows how you can use the TOvcSimpleArrayEditor to construct and use a simple array editor to allow editing a list of 50 strings.

Create a new project and add the following declaration to the private area of the forms class definition in the source code window:

```
  Names : array[0..49] of string[35];  {50 items in the array}
```

This is the data structure that holds the information entered in the array editor. It could be initialized with valid names, but for this example, leaving it in its default initialized state of empty will suffice.

Add components and set the property values as indicated in the following table:

Component	Property	Value
OvcSimpleArrayEditor	DataType	sftString
	MaxLength	35
	NumItems	50
	PictureMask	'x'

From the Object Inspector events tab for the simple array editor, double-click the OnGetItem event and add the following source code:

```
  Value := @Names[Index];
```

The OnGetItem event handler provides the address of the array element specified by Index. This method is called when the array editor needs to refresh the cell.

Run the project and experiment with the generated program.

Hierarchy

TCustomControl (VCL)

- ❶ TOvcCustomContol (OvcBase) 16
  - ❷ TOvcCustomControlEx (OvcBase) 19
    - ❸ TOvcBaseArrayEditor (OvcAE) 1220
      - TOvcSimpleArrayEditor (OvcAE)

Properties

❶	About	❸	HighlightColors		PictureMask
❶	AttachedLabel	❸	ItemIndex	❸	RangeHi
❸	BorderStyle	❶	LabelInfo	❸	RangeLo
❷	Controller	❸	LineColor	❸	RowHeight
	DataType		MaxLength	❸	TextMargin
	DecimalPlaces	❸	NumItems	❸	UseScrollBar
❸	DisabledColors	❸	PadChar		

Methods

❸ WriteCellValue

Events

❶	AfterEnter		OnGetItem		OnUserCommand
❶	AfterExit		OnGetItemColor		OnUserValidation
	OnChange	❶	OnMouseWheel		
	OnError		OnSelect		

Reference Section

DataType property

```
property DataType : TSimpleDataType
```

Default: sftString

↳ The type of data that the array editor will process.

The possible values and their corresponding data types are:

Value	**Delphi**	**C++Builder**
sftString	string	AnsiString
sftChar	AnsiChar	unsigned char
sftBoolean	Boolean	bool
sftYesNo	Boolean	bool
sftLongInt	LongInt	long or int
sftWord	Word	Word (unsigned short int)
sftInteger	SmallInt	short int
sftByte	Byte	Byte (unsigned char)
sftShortInt	ShortInt	ShortInt (signed char)
sftReal	Real	double
sftExtended	Extended	Extended (long double)
sftDouble	Double	double
sftSingle	Single	Single (float)
sftComp	Comp	Comp (double)

DecimalPlaces property

```
property DecimalPlaces : Byte
```

Default: 0

↳ The number of decimal places to display in the edit cell.

Although this property remains active for all possible data types, DecimalPlaces has an effect only for floating point fields.

MaxLength property

```
property MaxLength : Word
```

Default: dependent on the array editor data type

The maximum number of characters that can be entered in the edit cell.

MaxLength determines the logical width (in characters) of the edit cell. This value does not alter the physical width of the array editor. It limits the number of characters that are accepted by the edit cell.

Caution: When editing string arrays, MaxLength must be less than or equal to the maximum length of the array elements. If MaxLength is greater than the maximum length, no error is generated and a memory overwrite will occur.

PictureMask property

```
property PictureMask : TSimpleFieldMask
```

Default: pmAnyChar

Defines which characters the cell accepts and the character format.

This property defines the input and display mask used for the cell. The possible values are:

Value	**Mask**
pmAnyChar	'X'; {allows any character}
pmForceUpper	'!'; {allows any character, forces upper case}
pmForceLower	'L'; {allows any character, forces lower case}
pmForceMixed	'x'; {allows any character, forces mixed case}
pmAlpha	'a'; {allows alphas only}
pmUpperAlpha	'A'; {allows alphas only, forces upper case}
pmLowerAlpha	'l'; {allows alphas only, forces lower case}
pmPositive	'9'; {allows numbers and spaces only}
pmWhole	'i'; {allows numbers, spaces, minus}
pmDecimal	'#'; {allows numbers, spaces, minus, period}
pmScientific	'E'; {allows numbers, spaces, minus, period, 'e'}
pmHexadecimal	'K'; {allows 0-9 and A-F, forces upper case}
pmOctal	'O'; {allows 0-7, space}

Value	Mask
pmBinary	'b'; {allows 0-1, space}
pmTrueFalse	'B'; {allows T, t, F, f}
pmYesNo	'Y'; {allows Y, y, N, n}

The value assigned to PictureMask is tested only against the list shown above. It is still possible to assign a character that is inappropriate because of the cell's data type. For example, if the cell has a data type of sftReal, setting the PictureMask property to 'A' is allowed but is invalid because it allows only alphabetic characters.

You can directly enter a mask character in the PictureMask edit field in the Object Inspector or you can invoke the picture mask property editor to view and select from a list of the picture mask characters. Double-click on the PictureMask property value or select the dialog button to display the "Simple Mask" property editor.

See "Picture Masks" on page 316 for more information about picture masks and their usage.

TOvcPictureArrayEditor Component

The TOvcPictureArrayEditor component is derived from a TOvcBaseArrayEditor. The TOvcPictureArrayEditor edit cells are just TOvcPictureFields. The PictureMask, MaxLength, and DecimalPlaces properties behave just like the TOvcPictureField properties of the same name.

Example

This example uses a picture array editor to provide editing capabilities for a list of 50 dates. By itself, an array of 50 dates probably doesn't represent something that you would normally see. However, an array of dates associated with an array of names would allow you to enter and edit a list of birth dates.

Create a new project and add StDate (this is the unit that defines the TStDate type) to the uses list in the source code window. Add the following declaration to the private area of the forms class definition.

```
Dates : array[0..49] of TStDate;  {50 items in the array}
```

This is the data structure that holds the information entered in the array editor. Initialize this array in the form's OnCreate event handler by modifying the generated method shell to look like the following method:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  I : Integer;
begin
  for I := 0 to 49 do
    Dates[I] := CurrentDate + I;
end;
```

Add components and set the property values as indicated in the following table:

Component	**Property**	**Value**
OvcPictureArrayEditor	DataType	pftDate
	NumItems	50

From the Object Inspector events tab for the picture array editor, double-click the OnGetItem event and add the following source code:

```
Value := @Dates[Index];
```

The OnGetItem event handler provides the address of the array element specified by Index. This method is called when the array editor needs to refresh the cell.

Run the project and experiment with the generated program.

You can link multiple array editors (even ones of different types) so that scroll actions in one are automatically reflected in the others. See the EXAE project in the source code for an example of how to accomplish this.

Hierarchy

TCustomControl (VCL)

- ❶ TOvcCustomContol (OvcBase) 16
  - ❷ TOvcCustomControlEx (OvcBase) 19
    - ❸ TOvcBaseArrayEditor (OvcAE) 1220
      - TOvcPictureArrayEditor (OvcAE)

Properties

❶ About	❸ HighlightColors	PictureMask
❶ AttachedLabel	❸ ItemIndex	❸ RangeHi
❸ BorderStyle	❶ LabelInfo	❸ RangeLo
❷ Controller	❸ LineColor	❸ RowHeight
DataType	MaxLength	❸ TextMargin
DecimalPlaces	❸ NumItems	❸ UseScrollBar
❸ DisabledColors	❸ PadChar	

Methods

❸ WriteCellValue

Events

❶ AfterEnter	OnError	❶ OnMouseWheel
❶ AfterExit	OnGetItem	
OnChange	OnGetItemColor	

Reference Section

DataType property

```
property DataType : TPictureDataType
```

Default: pftString

The type of data that the array editor will process.

The possible values and their corresponding data types are:

Value	**Delphi**	**C++Builder**
pftString	string	AnsiString
pftChar	AnsiChar	unsigned char
pftBoolean	Boolean	bool
pftYesNo	Boolean	bool
pftLongInt	LongInt	long or int
pftWord	Word	Word (unsigned short int)
pftInteger	SmallInt	short int
pftByte	Byte	Byte
pftShortInt	ShortInt	ShortInt (signed char)
pftReal	Real	double
pftExtended	Extended	Extended (long double)
pftDouble	Double	double
pftSingle	Single	Single (float)
pftComp	Comp	Comp (double)
pftDate	TStDate	TStDate
pftTime	TStTime	TStTime

DecimalPlaces property

```
property DecimalPlaces : Byte
```

Default: 0

The number of decimal places to display in the edit cell.

Although this property remains active for all possible data types, DecimalPlaces has an effect only for floating point fields.

Epoch property

```
property Epoch : Integer
```

Default : 0

Specifies the starting year of a 100-year period.

The default of zero means that the value specified in Controller.Epoch is used. See “The Epoch property” on page 321 for more information.

MaxLength property

```
property MaxLength : Word
```

Default: dependent on the array editor data type

The maximum number of characters that can be entered in the edit cell.

MaxLength determines the logical width (in characters) of the edit cell. This value does not alter the physical width of the array editor, but it limits the number of characters that are accepted by the edit cell. MaxLength cannot be set to a value less than the length of the string assigned to the PictureMask property. Values greater than the length of the PictureMask are allowed and imply that positions in the edit cell that don’t have a corresponding mask character will use the last mask character specified in the PictureMask property.

Caution: When editing string arrays, MaxLength must be less than or equal to the maximum length of the array elements. If MaxLength is greater than the maximum length, no error is generated and a memory overwrite will occur.

```
property PictureMask : TPictureFieldMask
```

Default: ‘XXXXXXXXXX’

Defines which characters the field accepts and the character format.

This property defines the input and display mask used for the cell. The possible values are:

Value	**Mask**
`pmAnyChar`	`'X'; {allows any character}`
`pmForceUpper`	`'!'; {allows any character, forces upper case}`
`pmForceLower`	`'L'; {allows any character, forces lower case}`
`pmForceMixed`	`'x'; {allows any character, forces mixed case}`
`pmAlpha`	`'a'; {allows alphas only}`
`pmUpperAlpha`	`'A'; {allows alphas only, forces upper case}`
`pmLowerAlpha`	`'l'; {allows alphas only, forces lower case}`
`pmPositive`	`'9'; {allows numbers and spaces only}`
`pmWhole`	`'i'; {allows numbers, spaces, minus}`
`pmDecimal`	`'#'; {allows numbers, spaces, minus, period}`
`pmScientific`	`'E'; {allows numbers, spaces, minus, period, 'e'}`
`pmHexadecimal`	`'K'; {allows 0-9 and A-F, forces upper case}`
`pmBinary`	`'b'; {allows 0-1, space}`
`pmOctal`	`'O'; {allows 0-7, space}`
`pmTrueFalse`	`'B'; {allows T, t, F, f}`
`pmYesNo`	`'Y'; {allows Y, y, N, n}`

The value assigned to PictureMask is not tested for validity. It is possible to assign a mask that is invalid.

You can directly enter a mask string in the PictureMask edit field in the Object Inspector or you can invoke the picture mask property editor to view and select from several example picture masks. Double-click the PictureMask property value in the Object Inspector or select the dialog button to display the “Picture Mask” property editor.

See “Picture Masks” on page 316 for more information about picture masks and their usage.

When PictureMask is set, the MaxLength property is automatically set to reflect the length of the picture mask.

If PictureMask contains a literal decimal character, the value assigned to the DecimalPlaces property is ignored.

The following example sets the input mask for all array editor cells to allow entry of any character and force alpha characters to upper case. It also sets the MaxLength value to eight (the length of the picture mask).

```
PictureMask := '!!!!!!!!';
```

TOvcNumericArrayEditor Component

The TOvcNumericArrayEditor component is derived from a TOvcBaseArrayEditor. The TOvcNumericArrayEditor edit cells are just TOvcNumericFields. The PictureMask property behaves just like the TOvcNumericField property of the same name.

Example

This example shows how to use a numeric array editor to allow editing an array of currency values.

Create a new project and add the following declaration to the private area of the form's class definition:

```
  Amounts : array[0..49] of Real;  {50 items in the array}
```

This is the data structure that holds the information entered in the array editor. This array is not initialized, since leaving it in its default initialized state of zeros is sufficient.

Add components and set the property values as indicated in the following table:

Component	**Property**	**Value**
OvcNumericArrayEditor	DataType	nftReal
	NumItems	50
	PictureMask	'######.##'
	RangeHi	50000
	RangeLo	0

From the Object Inspector events tab for the picture array editor, double-click the OnGetItem event and add the following source code:

```
  Value := @Amounts[Index];
```

The OnGetItem event handler provides the address of the array element specified by Index. This method is called when the array editor needs to refresh the cell.

Run the project and experiment with the generated program.

Hierarchy

TCustomControl (VCL)

- ❶ TOvcCustomContol (OvcBase) 16
  - ❷ TOvcCustomControlEx (OvcBase) 19
    - ❸ TOvcBaseArrayEditor (OvcAE) 1220
      - TOvcNumericArrayEditor (OvcAE)

Properties

❶	About	❸	HighlightColors		PictureMask
❶	AttachedLabel	❸	ItemIndex	❸	RangeHi
❸	BorderStyle	❶	LabelInfo	❸	RangeLo
❷	Controller	❸	LineColor	❸	RowHeight
	DataType	❸	NumItems	❸	TextMargin
❸	DisabledColors	❸	PadChar	❸	UseScrollBar

Methods

❸ WriteCellValue

Events

❶	AfterEnter	❶	AfterExit	❶	OnMouseWheel

Reference Section

DataType property

```
property DataType : TNumericDataType
```

Default: nftReal

The type of data that the array editor will process.

The possible values and their corresponding data types are:

Value	Delphi	C++Builder
nftLongInt	LongInt	long or int
nftWord	Word	Word (unsigned short in)
nftInteger	SmallInt	short int
nftByte	Byte	Byte (unsigned char)
nftShortInt	ShortInt	ShortInt (signed char)
nftReal	Real 7 double	
nftExtended	Extended	Extended (long double)
nftDouble	Double	double
nftSingle	Single	Single (float)
nftComp	Comp	Comp (double)

`property PictureMask : TNumericFieldMask`

Default: ‘##########’

Defines which characters the field accepts and the character format.

This property defines the input and display mask used for the cell. The possible values are:

Value	**Mask**
`pmPositive`	`'9'; {allows numbers and spaces only}`
`pmWhole`	`'i'; {allows numbers, spaces, minus}`
`pmDecimal`	`'#'; {allows numbers, spaces, minus, period}`
`pmScientific`	`'E'; {allows numbers, spaces, minus, period, 'e'}`
`pmHexadecimal`	`'K'; {allows 0-9 and A-F, forces upper case}`
`pmOctal`	`'O'; {allows 0-7, space}`
`pmBinary`	`'b'; {allows 0-1, space}`

The value assigned to PictureMask is not tested for validity. It is possible to assign a mask that is invalid.

You can directly enter a mask string in the PictureMask edit field in the Object Inspector or you can invoke the picture mask property editor to view and select from several example picture masks. Double-click the PictureMask property value in the Object Inspector or select the dialog button to display the “Numeric Mask” property editor.

See “Picture Masks” on page 316 for more information about picture masks and their usage.

TOvcBaseDbArrayEditor Class

The TOvcBaseDbArrayEditor class is the immediate ancestor of the three data-aware array editor components. It provides methods and properties common to all data-aware array editors.

Although the data-aware array editors do not descend from the standard array editors, they are very similar in operation to their non-data-aware counterparts. The data-aware array editors encapsulate their corresponding data-aware entry fields (similar to the use of the standard entry fields by the non-data-aware array editors discussed earlier in this section).

The DbArrayEditors differ mainly in the fact that they connect to a data source, giving them the ability to display and edit the data of a given field in a database.

You can link multiple data-aware array editors (even ones of different types) so that scroll actions in one are automatically reflected in the others.

Array Editor Commands

The following commands are available in any Orpheus data-aware array editor. The commands and key assignments described here are available when you use the “Default” or “WordStar” command tables (several command tables can be used at the same time). See “TOvcCommandProcessor Class” on page 50 for information on how to customize the command tables.

Many of the commands mentioned here work differently when used in a data-aware numeric array editor. The reason for the difference in behavior is that there is no caret movement in a numeric edit cell. The caret always remains in a fixed position at the right of the cell. Attempting to select any portion of the value displayed in a numeric array editor cell will highlight the complete contents. The differences are noted in the following descriptions.Hierarchy

Table 33.4: *Commands and their functions*

Command	**Default**	**WordStar**
ccBack	<BkSp>	<CtrlH>
	Delete the character to the left of the caret.	
ccCompleteDate	not assigned	not assigned
	Insert the current day, month, or year (depending on the position of the caret) in an array editor cell.	
ccCompleteTime	not assigned	not assigned
	Insert the current hour, minute, or second (depending on the position of the caret) in an array editor cell.	

Table 33.4: *Commands and their functions (continued)*

`ccCopy`	`<CtrlIns>, <CtrlC>`	`not assigned`
	`Copy the selected text to the clipboard.`	
`ccCtrlChar`	`not assigned`	`<CtrlP>`
	`Treat the next character as a control character.`	
`ccCut`	`<ShiftDel>, <CtrlX>`	`not assigned`
	`Delete the selected text after copying it to the clipboard.`	
`ccDec`	`not assigned`	`not assigned`
	`Decrement the current value by one ordinal amount.`	
`ccDel`	`<Del>`	`<CtrlG>`
	`Delete the current character. Or, if text is selected, delete the selection.`	
`ccDelBol`	`not assigned`	`not assigned`
	`Delete text from the caret position to the beginning of the edit cell. This command does nothing in a numeric array editor.`	
`ccDelEol`	`not assigned`	`<ShiftCtrlY>, <CtrlQ><Y>`
	`Delete text from the caret position to the end of the edit cell. This command does nothing in a numeric array editor.`	
`ccDelLine`	`not assigned`	`<CtrlY>`
	`Delete the edit cell contents.`	
`ccDelWord`	`not assigned`	`<CtrlT>`
	`Delete the word to the right of the caret. This command does nothing in a numeric array editor.`	
`ccDown`	`<Down>`	`<CtrlX>`
	`Move the focus down to the next list item.`	
`ccEnd`	`<End>`	`<CtrlQ><D>`
	`Move the caret to the end of the edit cell. This command does nothing in a numeric array editor.`	
`ccExtendEnd`	`<ShiftEnd>`	`not assigned`
	`Extend the selection to the end of the edit cell.`	
`ccExtendHome`	`<ShiftHome>`	`not assigned`
	`Extend the selection to the start of the edit cell.`	
`ccExtendLeft`	`<ShiftLeft>`	`not assigned`
	`Extend the selection to the left by one character.`	
`ccExtendRight`	`<ShiftRight>`	`not assigned`

Table 33.4: *Commands and their functions (continued)*

	Extend the selection to the right by one character.	
ccExtWordLeft	<CtrlShiftLeft>	not assigned
	Extend the selection to the left by one word.	
ccExtWordRight	<CtrlShiftRight>	not assigned
	Extend the selection to the right by one word.	
ccFirstPage	<CtrlHome>	not assigned
	Move the focus to the first list item.	
ccHome	<Home>	<CtrlQ><S>
	Move the caret to the beginning of the edit cell. This command does nothing in a numeric array editor.	
ccInc	not assigned	not assigned
	Increment the value of the edit cell.	
ccIns	<Ins>	<CtrlV>
	Toggle insert mode. The caret reflects the current mode. By default, a solid line indicates insert mode; a block indicates overwrite mode. Although this command is recognized by numeric array editors, it doesn't do anything except change the caret.	
ccLastPage	<CtrlEnd>	not assigned
	Move the focus to the last list item.	
ccLeft	<Left>	<CtrlS>
	Move the caret left one character. If the caret is already at the beginning of the edit cell (or this is a numeric array editor) and AutoAdvanceLeftRight is *true*, the caret is moved to the previous component, just as it is if <ShiftTab> is pressed.	
ccNextPage	<PgDn>	<CtrlC>
	Scroll the list down one page.	
ccPaste	<ShiftIns>, <CtrlV>	not assigned
	Paste the text from the clipboard.	
ccPrevPage	<PgUp>	<CtrlR>
	Scroll the list up one page.	
ccRestore	<AltBkSp>, <CtrlZ>	<CtrlQ><L>
	Restore the original contents of the edit cell (i.e., the value the edit cell contained when the array editor last had the focus).	
ccRight	<Right>	<CtrlD>

Table 33.4: *Commands and their functions (continued)*

	Move the caret right one character. If the caret is already at the end of the edit cell(or this is a numeric array editor) and the AutoAdvanceLeftRight property is True, the caret is moved to the next component, just as it is if <Tab> is pressed.	
`ccUp`	`<Up>`	`<CtrlE>`
	Move the focus to the previous list item.	
`ccWordLeft`	`<CtrlLeft>`	`<CtrlA>`
	Move the caret left one word. This command does nothing in a numeric array editor.	
`ccWordRight`	`<CtrlRight>`	`<CtrlF>`
	Move the caret right one word. This command does nothing in a numeric array editor.	

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase) 16

❷ TOvcCustomControlEx (OvcBase) 19

TOvcBaseDbArrayEditor (OvcDbAE)

Properties

- ❶ About
- ❶ AttachedLabel
- AutoRowHeight
- BorderStyle
- Canvas
- ❷ Controller
- DataField
- DataLink
- DataSource
- DateOrTime
- DecimalPlaces
- EpochField
- FieldType
- HighlightColors
- ❶ LabelInfo
- LineColor
- MaxLengthOptions
- PadChar
- PageScroll
- RangeHi
- RangeLo
- RowHeight
- RowIndicatorWidth
- ShowIndicator
- TextMargin
- UseScrollBar
- ZeroAsNull

Methods

- Reset
- Scroll
- UpdateRecord

Events

- ❶ AfterEnter
- ❶ AfterExit
- OnChange
- OnError
- OnGetItem
- OnGetItemColor
- OnIndicatorClick
- ❶ OnMouseWheel
- OnSelect
- OnUserCommand
- OnUserValidation

AutoRowHeight property

```
property AutoRowHeight : Boolean
```

Default: True

Determines whether the array editor's row height should be automatically adjusted according to the font size.

If AutoRowHeight is True, the array editor automatically calculates the row height based on the size of the currently selected font. If this property is True, changing the font, at run time, will force the row height to be recalculated.

BorderStyle property

```
property BorderStyle : TBorderStyle
```

Default: bsSingle

Determines the style used when drawing the border.

The possible values are bsSingle (a single line border is drawn around the component) or bsNone (no border line).

The TBorderStyle type is defined in the VCL's Controls unit.

DataField property

```
property DataField : string
```

Identifies the field (in the data source component) from which the array editor component displays data.

DataSource property

```
property DataSource : TDataSource
```

Specifies the data source component where the array editor obtains the data to display.

DateOrTime property

```
property DateOrTime : TDateOrTime
```

```
TDateOrTime = (ftUseDate,
  ftUseTime, ftUseBothEditDate, ftUseBothEditTime);
```

Default: ftUseDate

Determines whether ftDateTime database fields are edited as dates or times.

Orpheus does not provide an array editor capable of editing both dates and times. This property is used when the data-aware array editor is connected to an ftDateTime field to determine which of the two values to edit.

The behavior of the array editor for various values of DateOrTime is as follows:

Value	**Behavior**
ftUseDate	The cell displays and edits the date portion of the ftDateTime value.
ftUseTime	The cell displays and edits the time portion of the ftDateTime value.
ftUseBothEditDate	The cell displays the date and time of the ftDateTime value when the field is unfocused, and allows you to edit the date portion when focused.
ftUseBothEditTime	The cell displays the date and time of the ftDateTime value when the field is unfocused, and allows you to edit the time portion when focused.

In all cases, the portion of the ftDateTime field that is not edited is preserved.

DecimalPlaces property

```
property DecimalPlaces : Byte
```

Default: 0

↳ The number of digits that are displayed to the right of the decimal point.

This property can be used to set a decimal position that will display a fractional portion of the value only if one exists. Otherwise setting the decimal position in the picture mask of a picture entry field causes the decimal point to always be displayed.

DecimalPlaces is useful only for simple and picture data-aware array editors configured to edit real data types and is not used by others.

Field run-time, read-only property

```
property Field: TField
```

↳ Returns the TField object to which the array editor component is linked.

Use the Field object to change the value of the data in the field programmatically. Any changes made to the datasource are automatically reflected to the array editor if necessary.

FieldType property

```
property FieldType : TFieldType
```

```
TFieldType = (ftUnknown, ftString, ftSmallint, ftInteger,
  ftWord, ftBoolean, ftFloat, ftCurrency, ftBCD, ftDate,
  ftTime, ftDateTime, ftBytes, ftVarBytes, ftBlob, ftMemo,
  ftGraphic);
```

Default: ftUnknown

↳ Determines the type of the database field.

When a data-aware array editor is attached to an active data source and the DataField property is set to the name of the field, this property is automatically set to match the data type of the database field. If it is not desirable to activate the data source, this field must be set to the correct field type so that the array editor can configure its internal data type and initialize the picture mask.

The data-aware array editors do not support all of the database field types shown in the definition of TFieldType (see the VCL's DB unit). If you select one that is not supported, an error message is displayed. For example, if the ftGraphic field type is selected, an error is displayed because none of the data-aware array editors support that field type.

See also: DataField, DataSource

HighlightColors property

```
property HighlightColors : TOvcColors
```

Default: clHighlightText and clHighlight

↳ Determines the colors used to display the highlighted portion of the cell.

See "TOvcColors Class" on page 1186 for more information. Refer to the VCL's Color property for a list of available values and their meanings.

The following example displays highlighted text in blue with a white background.

```
  AE1.HighlightColors.TextColor := clBlue;
  AE1.HighlightColors.BackColor := clWhite;
```

LineColor property

```
property LineColor : TColor
```

Default: clSilver

Determines the color of the row divider lines.

MaxLength property

```
property MaxLength : Word
```

Default: dependent on the array editor's data type

The maximum number of characters that can be entered in a given cell in the array editor.

MaxLength determines the logical width (in characters) of the array editor. This value does not change the physical width of the array editor. It limits the number of characters that are accepted. The maximum value for MaxLength is 255.

OnChange event

```
property OnChange : TNotifyEvent;
```

Defines an event handler that is called when the contents of the active edit cell are modified.

The method assigned to the OnChange event is called when the active edit cell is modified. Sender is the class instance of the array editor containing the active cell.

TNotifyEvent is defined in the VCL's Classes unit.

OnError event

```
property OnError : TValidationErrorEvent;
```

```
TValidationErrorEvent = procedure(Sender : TObject;
  ErrorCode : Word; ErrorMessage : string) of object;
```

Defines an event handler that is called when an array editor error occurs.

The method assigned to the OnError event is called when an error occurs. If no method is assigned to this event, the event handler assigned to the array editor's controller OnError event is called.

The possible values for ErrorCode are:

Value	Description
oeRangeError	The entered value is not within the accepted range.
oeInvalidNumber	An invalid number was entered in a numeric cell.
oeInvalidDate	An invalid date was entered in a date cell.
oeInvalidTime	An invalid time was entered in a time cell.
oeCustomError	The first error code reserved for user applications.

All error values less than oeCustomError are reserved for use by Orpheus.

ErrorMessage is a short description of the error.

See “TOvcController Component” on page 28 for additional information about error handling.

OnGetItemColor event

```
property OnGetItemColor : TGetItemColorEvent;
```

```
TGetItemColorEvent = procedure(
  Sender : TObject; Field : TField; Row : LongInt;
  var FG, BG : TColor) of object;
```

Defines an event handler that is called to get the colors used to draw an array element.

The method assigned to the OnGetItemColor event is called by the array editor to get the colors to use when displaying the item specified by Index. On entry to the event handler, FG and BG contain the default foreground and background colors. Set FG and BG to the desired colors. Sender is the class instance of the array editor.

This event is not called to get colors for the active cell. The colors for the active cell are obtained from the current Font.Color and Color properties.

OnIndicatorClick event

```
property OnIndicatorClick : TIndicatorClickEvent
```

```
TIndicatorClickEvent = procedure(
  Sender : TObject; Row : LongInt) of object;
```

↳ Defines an event handler that is called to provide notification that the indicator has been clicked.

Row is the visible row clicked. Sender is the class instance of the array editor.

OnUserCommand event

```
property OnUserCommand : TUserCommandEvent
```

```
TUserCommandEvent = procedure(
  Sender : TObject; Command : Word) of object;
```

↳ Defines an event handler that is called when a user-defined command is entered.

The method assigned to the OnUserCommand event is called when you enter the key sequence corresponding to one of the user-defined commands, (ccUser1, ccUser2, etc.). For example, suppose you add the <CtrlD> key sequence to one of the active command tables and assign it to the ccUser1 command. When you press <CtrlD>, the method assigned to the OnUserCommand event is called and ccUser1 is passed in Command.

See “TOvcCommandProcessor Class” on page 50 for additional information about user-defined commands and command tables.

OnUserValidation event

```
property OnUserValidation : TUserValidationEvent
```

```
TUserValidationEvent = procedure(
  Sender : TObject; var ErrorCode : Word) of object;
```

↳ Defines an event handler that is called when cell validation is required.

The method assigned to the OnUserValidation event is called after the default validation is performed on the current array editor cell. Validation is performed when the array editor loses the focus. ErrorCode contains the result of the validation. If it is zero, then the default validation didn’t find any errors and you can perform any additional validation that is required. The error constants are defined in the OnError event.

If the OnUserValidation event handler determines that the cell contents are invalid, assign a value to ErrorCode before exiting the method. The error value should be greater than or equal to oeCustomError (32768).

Caution: The event handler assigned to the OnUserValidation event must not perform any actions that could cause the focus to change. This could cause Windows to lose track of where the focus is and could cause other problems as well. Do not display a form or dialog reporting the error—that should only be done in a method assigned to the components OnError event or the TOvcController's OnError event. The OnUserValidation event should only do what is necessary to confirm that the cell contents are valid.

See "TOvcController Component" on page 28 for additional information about error handling for multiple components.

See also: OnError

PageScroll property

```
property PageScroll : Boolean
```

Default: False

Defines the scroll behavior of the array editor.

When PageScroll is set to False (the default), the active cell moves incrementally in the direction of the scroll until the last visible row is reached. All visible cells are then scrolled.

When PageScroll is set to True, the active cell jumps by one page in the direction of the scroll before scrolling all cells. This behavior is consistent with the scrolling behavior of the VCL's grids.

RangeHi property

```
property RangeHi : string
```

Default: depends on the array editor's data type

The upper limit for the range of allowable cell values.

This property is intended primarily for use at design time. However, it can be used at run time by assigning a string version of the desired range to the property. Range settings are ignored for string type array editors.

You can directly enter a range limit in the RangeHi edit field in the Object Inspector or you can invoke the Range Property Editor, which allows entry and validation of the range limit into a cell of the appropriate data type.

See also: RangeLo

RangeLo property

```
property RangeLo : string
```

Default: depends on the array editor’s data type

↳ The lower limit for the range of allowable cell values.

This property is intended primarily for use at design time. However, it can be used at run time by assigning a string version of the desired range to the property. Range settings are ignored for string type array editors.

You can directly enter a range limit in the RangeLo edit field in the Object Inspector or you can invoke the Range Property Editor, which allows entry and validation of the range limit into a cell of the appropriate data type.

See also: RangeHi

RowHeight property

```
property RowHeight : Integer
```

Default: dependent on the parent font

↳ The height of the array editor cells in pixels.

The row height is initially determined based on the current font size. Assigning a new value to the RowHeight property causes the array editor to resize each editing cell. The array editor is also resized if necessary to avoid displaying a partial row.

Assigning a value to this property sets the AutoRowHeight property to False.

RowIndicatorWidth property

```
property RowIndicatorWidth : Integer
```

Default: 11

↳ Determines the width of the row indicators.

The RowIndicatorWidth property is used to determine the width of the row indicator that is shown to the left of the array editor. The minimum value for RowIndicatorWidth is 1.

See also: ShowIndicator

Scroll method

```
procedure Scroll(Delta : Integer);
```

↳ Defines the scroll behavior of the editor.

Scroll causes the datasource to scroll by the number of records specified by Delta. If Delta is negative, the datasource scrolls toward the beginning of the file. If Delta is positive, the datasource scrolls toward the end of the file. If the value of Delta would cause the datasource to scroll past the end or before the beginning of the file, it is ignored and the datasource is positioned at the beginning or the end as appropriate.

ShowIndicator property

```
property ShowIndicator : Boolean
```

Default: True

↳ Determines if the active record indicator is displayed.

If ShowIndicator is True, an indicator showing the active record and its editing state is displayed at the left edge of the array editor.

See also: RowIndicatorWidth

TextMargin property

```
property TextMargin : Integer
```

Default: 2

↳ Determines the cell's display indent.

The TextMargin property controls the left indent in pixels for the simple and picture array editors. For the numeric array editor, this property is the right margin since the painting is done from right to left. The minimum value for TextMargin is 2.

UseScrollBar property

```
property UseScrollBar : Boolean
```

Default: True

↳ Determines if a vertical scrollbar is displayed.

If UseScrollBar is True, a vertical scrollbar is displayed at the right of the array editor.

```
property ZeroAsNull : Boolean
```

Default: False

Determines if zero values are stored as null.

If ZeroAsNull is set to True, fields editing a numeric data type that contain zero are stored as null database values. If False, the numeric zero is stored in the attached database field.

Setting this property to True is not necessary if you just want to initialize the database field to null. You can call the Field.Clear method to do this on an as-needed basis.

See also: Field

TOvcDbSimpleArrayEditor Component

The TOvcDbSimpleArrayEditor component is derived from a TOvcBaseDbArrayEditor. The TOvcDbSimpleArrayEditor edit cells are just TOvcDbSimpleFields. The PictureMask, MaxLength, and DecimalPlaces properties behave just like the TOvcDbSimpleField properties of the same name.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase). 16

❷ TOvcCustomControlEx (OvcBase) 19

❸ TOvcBaseDbArrayEditor (OvcDbAE)

TOvcBaseDbSimpleArrayEditor (OvcDbAE)

Properties

❶ About	❸ DecimalPlaces	PictureMask
❶ AttachedLabel	❸ EpochField	❸ RangeHi
❸ AutoRowHeight	❸ FieldType	❸ RangeLo
❸ BorderStyle	❸ HighlightColors	❸ RowHeight
❸ Canvas	❶ LabelInfo	❸ RowIndicatorWidth
❷ Controller	❸ LineColor	❸ ShowIndicator
❸ DataField	❸ MaxLength	❸ TextMargin
❸ DataLink	❸ Options	❸ UseScrollBar
❸ DataSource	❸ PadChar	❸ ZeroAsNull
❸ DateOrTime	❸ PageScroll	

Methods

❸ Reset	❸ Scroll	❸ UpdateRecord

Events

❶ AfterEnter	❸ OnGetItem	❸ OnSelect
❶ AfterExit	❸ OnGetItemColor	❸ OnUserCommand
❸ OnChange	❸ OnIndicatorClick	❸ OnUserValidation
❸ OnError	❶ OnMouseWheel	

PictureMask property

`property PictureMask : TSimpleFieldMask`

Default: pmAnyChar

↳ Defines which characters the cell accepts and the character format.

This property defines the input and display mask used for the cell. The possible values are:

Value	Mask
`pmAnyChar`	`'X'; {allows any character}`
`pmForceUpper`	`'!'; {allows any character, forces upper case}`
`pmForceLower`	`'L'; {allows any character, forces lower case}`
`pmForceMixed`	`'x'; {allows any character, forces mixed case}`
`pmAlpha`	`'a'; {allows alphas only}`
`pmUpperAlpha`	`'A'; {allows alphas only, forces upper case}`
`pmLowerAlpha`	`'l'; {allows alphas only, forces lower case}`
`pmPositive`	`'9'; {allows numbers and spaces only}`
`pmWhole`	`'i'; {allows numbers, spaces, minus}`
`pmDecimal`	`'#'; {allows numbers, spaces, minus, period}`
`pmScientific`	`'E'; {allows numbers, spaces, minus, period, 'e'}`
`pmHexadecimal`	`'K'; {allows 0-9 and A-F, forces upper case}`
`pmOctal`	`'O'; {allows 0-7, space}`
`pmBinary`	`'b'; {allows 0-1, space}`
`pmTrueFalse`	`'B'; {allows T, t, F, f}`
`pmYesNo`	`'Y'; {allows Y, y, N, n}`

The value assigned to PictureMask is tested only against the list shown above. It is still possible to assign a character that is inappropriate because of the cell's data type. For example, if the cell has a data type of sftReal, setting the PictureMask property to 'A' is allowed but is invalid because it allows only alphabetic characters.

You can directly enter a mask character in the PictureMask edit field in the Object Inspector or you can invoke the picture mask property editor to view and select from a list of the picture mask characters. Double-click on the PictureMask property value or select the dialog button to display the “Simple Mask” property editor.

See “Picture Masks” on page 316 for more information about picture masks and their usage.

TOvcDbPictureArrayEditor Component

The TOvcDbPictureArrayEditor component is derived from a TOvcBaseDbArrayEditor. The TOvcDbPictureArrayEditor edit cells are just TOvcDbPictureFields. The PictureMask, MaxLength, and DecimalPlaces properties behave just like the TOvcDbPictureField properties of the same name.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase) 16

❷ TOvcCustomControlEx (OvcBase) 19

❸ TOvcBaseDbArrayEditor (OvcDbAE). 1247

TOvcBaseDbPictureArrayEditor (OvcDbAE)

Properties

❶ About	❸ DecimalPlaces	PictureMask
❶ AttachedLabel	❸ EpochField	❸ RangeHi
❸ AutoRowHeight	❸ FieldType	❸ RangeLo
❸ BorderStyle	❸ HighlightColors	❸ RowHeight
❸ Canvas	❶ LabelInfo	❸ RowIndicatorWidth
❷ Controller	❸ LineColor	❸ ShowIndicator
❸ DataField	❸ MaxLength	❸ TextMargin
❸ DataLink	❸ Options	❸ UseScrollBar
❸ DataSource	❸ PadChar	❸ ZeroAsNull
❸ DateOrTime	❸ PageScroll	

Methods

❸ Reset	❸ Scroll	❸ UpdateRecord

Events

❶ AfterEnter	❸ OnGetItem	❸ OnSelect
❶ AfterExit	❸ OnGetItemColor	❸ OnUserCommand
❸ OnChange	❸ OnIndicatorClick	❸ OnUserValidation
❸ OnError	❶ OnMouseWheel	

PictureMask property

```
property PictureMask : TPictureFieldMask
```

Default: 'XXXXXXXXXX'

Defines which characters the field accepts and the character format.

This property defines the input and display mask used for the cell. The possible values are:

Value	Mask
pmAnyChar	'X'; {allows any character}
pmForceUpper	'!'; {allows any character, forces upper case}
pmForceLower	'L'; {allows any character, forces lower case}
pmForceMixed	'x'; {allows any character, forces mixed case}
pmAlpha	'a'; {allows alphas only}
pmUpperAlpha	'A'; {allows alphas only, forces upper case}
pmLowerAlpha	'l'; {allows alphas only, forces lower case}
pmPositive	'9'; {allows numbers and spaces only}
pmWhole	'i'; {allows numbers, spaces, minus}
pmDecimal	'#'; {allows numbers, spaces, minus, period}
pmScientific	'E'; {allows numbers, spaces, minus, period, 'e'}
pmHexadecimal	'K'; {allows 0-9 and A-F, forces upper case}
pmOctal	'O'; {allows 0-7, space}
pmBinary	'b'; {allows 0-1, space}
pmTrueFalse	'B'; {allows T, t, F, f}
pmYesNo	'Y'; {allows Y, y, N, n}

The value assigned to PictureMask is not tested for validity. It is possible to assign a mask that is invalid.

You can directly enter a mask string in the PictureMask edit field in the Object Inspector or you can invoke the picture mask property editor to view and select from several example picture masks. Double-click the PictureMask property value in the Object Inspector or select the dialog button to display the "Picture Mask" property editor.

See "Picture Masks" on page 316 for more information about picture masks and their usage.

When PictureMask is set, the MaxLength property is automatically set to reflect the length of the picture mask.

If PictureMask contains a literal decimal character, the value assigned to the DecimalPlaces property is ignored.

The following example sets the input mask for all array editor cells to allow entry of any character and force alpha characters to upper case. It also sets the MaxLength value to eight (the length of the picture mask).

```
PictureMask := '!!!!!!!!';
```

TOvcDbNumericArrayEditor Component

The TOvcDbNumericArrayEditor component is derived from a TOvcBaseDbArrayEditor. The TOvcDbNumericArrayEditor edit cells are just TOvcDbNumericFields. The PictureMask property behaves just like the TOvcDbNumericField property of the same name.

Hierarchy

TCustomControl (VCL)

❶ TOvcCustomControl (OvcBase). 16

❷ TOvcCustomControlEx (OvcBase) 19

❸ TOvcBaseDbArrayEditor (OvcDbAE). 1247

TOvcDbNumericArrayEditor (OvcDbAE)

Properties

❶ About
❶ AttachedLabel
❸ AutoRowHeight
❸ BorderStyle
❸ Canvas
❷ Controller
❸ DataField
❸ DataLink
❸ DataSource
❸ DateOrTime
❸ DecimalPlaces
❸ EpochField
❸ FieldType
❸ HighlightColors
❶ LabelInfo
❸ LineColor
❸ MaxLength
❸ Options
❸ PadChar
❸ PageScroll
PictureMask
❸ RangeHi
❸ RangeLo
❸ RowHeight
❸ RowIndicatorWidth
❸ ShowIndicator
❸ TextMargin
❸ UseScrollBar
❸ ZeroAsNull

Methods

❸ Reset
❸ Scroll
❸ UpdateRecord

Events

❶ AfterEnter
❶ AfterExit
❸ OnChange
❸ OnError
❸ OnGetItem
❸ OnGetItemColor
❸ OnIndicatorClick
❶ OnMouseWheel
❸ OnSelect
❸ OnUserCommand
❸ OnUserValidation

PictureMask property

```
property PictureMask : TNumericFieldMask
```

Default: ‘iiiiiii’

Defines which characters the field accepts and the character format.

This property defines the input and display mask used for the cell. The possible values are:

Value	Mask
pmPositive	'9'; {allows numbers and spaces only}
pmWhole	'i'; {allows numbers, spaces, minus}
pmDecimal	'#'; {allows numbers, spaces, minus, period}
pmScientific	'E'; {allows numbers, spaces, minus, period, 'e'}
pmHexadecimal	'K'; {allows 0-9 and A-F, forces upper case}
pmOctal	'O'; {allows 0-7, space}
pmBinary	'b'; {allows 0-1, space}

The value assigned to PictureMask is not tested for validity. It is possible to assign a mask that is invalid.

You can directly enter a mask string in the PictureMask edit field in the Object Inspector or you can invoke the picture mask property editor to view and select from several example picture masks. Double-click the PictureMask property value in the Object Inspector or select the dialog button to display the “Numeric Mask” property editor.

See “Picture Masks” on page 316 for more information about picture masks and their usage.

Miscellaneous:

This section describes the Orpheus components that don't fit neatly into any other category, and several classes used by other Orpheus components.

The following components are described:

- TOvcCustomEdit is the ancestor of the TOvcEdit based components. It provides the attached label and Orpheus controller support.
- TOvcEdit is identical to the VCL TEdit, with the added capability of an attached label and Orpheus controller support.
- TOvcEdPopup is a TOvcCustomEdit descendant control with the ability to display a button in the client area. The button is used to "pop" an associated window.

TOvcCustomEdit Class

The TOvcCustomEdit class is the immediate ancestor of the TOvcEdit component. It implements all the methods and properties used by the TOvcEdit component and is identical to TOvcEdit except that no properties are published.

TOvcCustomEdit is provided to facilitate creation of descendent edit components. For property and method descriptions, see “TOvcEdit Component” on page 1273.

Hierarchy

TCustomEdit (VCL)

    TOvcCustomEdit (OvcEditF)

TOvcEdit Component

TOvcEdit is identical to the standard TEdit with the addition of the capability to have an attached label and to provide access to an Orpheus Controller (TOvcController class) for its descendants. See page 796 for additional information about the attached label.

The primary purpose of the TOvcEdit is to act as a common ancestor to the other edit-related components in Orpheus, but you can use this control anywhere you would normally use the standard TEdit control.

Hierarchy

TCustomEdit (VCL)

  TOvcCustomEdit (OvcEditF) 1272

    TOvcEdit (OvcEditF)

Properties

About	Controller
AttachedLabel	LabelInfo

Reference Section

About — read-only property

```
property About : string
```

↳ Shows the current version of Orpheus.

About is provided so you can identify your Orpheus version if you need technical support. You can display the Orpheus about box by double-clicking this property or selecting the dialog button to the right of the property value.

AttachedLabel — property

```
property AttachedLabel : TOvcAttachedLabel
```

↳ Provides access to the TOvcAttachedLabel component.

The TOvcAttachedLabel is normally accessed only at design time, but this property makes it possible to alter the attached label at run time if necessary.

See “TOvcAttachedLabel Component” on page 796 for more information.

Controller — property

```
property Controller : TOvcController
```

Default: the first TOvcController object on the form.

↳ The TOvcController object that is attached to this component.

If this property is not assigned, some or all the features provided by the component will not be available. The Controller property is not published, but descendants can publish it as needed (as most of the Orpheus components do). See “TOvcController Component” on page 28for additional information.

LabelInfo — property

property LabelInfo : TOvcLabelInfo

LabelInfo provides access to the status of the attached label.

TOvcLabelInfo (see page 798) groups the Visible, OffsetX, and OffsetY properties that determine whether the attached label is visible and where it is positioned.

TOvcEdPopup Class

TOvcEdPopup implements an edit control with the ability to display a button embedded within the client area of the control. The action performed when the button is pressed is not defined in this class. As is indicated by the name of the class, TOvcEdPopup was created to provide the capability to “pop-up” some other control. For example, TOvcEdPopup is used as the ancestor for a pop-up calendar control and a popup calculator control.

This class is designed to be used as the basis for creating other edit controls. You should not create an instance of TOvcEdPopup directly. Descendants must override the PopupOpen and PopupClose methods to implement the specific action to take when the button is pressed.

Hierarchy

TCustomEdit (VCL)

TOvcCustomEdit (OvcEditF) 1272

TOvcEdPopup (OvcEdPop)

Properties

ButtonGlyph	PopupAnchor	ShowButton

Methods

PopupClose	PopupOpen

Reference Section

PopupAnchor property

```
property PopupAnchor : TOvcPopupAnchor
```

```
TOvcPopupAnchor = (paLeft, paRight);
```

Default: paLeft

↳ Determines which corner (lower left or lower right) of the edit control is used as a reference to position the pop-up window.

ShowButton property

```
property ShowButton : Boolean
```

↳ Determines if the pop-up button is displayed.

If ShowButton is False, the button is not displayed. If True, the pop-up button is displayed within the client area of the edit control at the right side.

ButtonGlyph property

```
property ButtonGlyph : TBitmap
```

↳ Determines the bitmap that is displayed on the pop-up button.

PopupClose virtual method

```
procedure PopupClose(Sender : TObject);
```

↳ Defines the action to take to close the pop-up control.

The exact behavior of PopupClose is determined by its implementation in descendants of this class.

See also: PopupOpen

PopupOpen virtual method

```
procedure PopupOpen;
```

↳ Defines the action to take to open (display) the pop-up control.

The exact behavior of PopupOpen is determined by its implementation in descendants of this class.

See also: PopupClose

Chapter 34: Demo Programs

This chapter describes the demonstration programs that are provided to illustrate the use of Orpheus.

The address book demonstration shows how to use the various entry field components in a database application. It also shows how to use the notebook, the virtual listbox, and the editor.

The text viewer demonstration implements a viewer for ASCII text files. The file size is limited by available memory.

The file viewer demonstration implements a viewer for ASCII text files and binary files. Files of up to 2 billion lines of text can be viewed.

The text editor demonstration implements an editor for ASCII text files. The file size is limited only by available memory.

The MDI editor demonstration implements an editor for multiple ASCII text files. It also allows editing of different parts of the same file. The file size is limited only by available memory and the number of files that are open.

Address Book

The ADDRBOOK program demonstrates how to implement an application using the entry field components in a database application. It also demonstrates the use of the TOvcNotebook, TOvcVirtualListbox, TOvcEditor and TO32FlexEdit components.

Although ADDRBOOK doesn't use an actual database engine such as the Borland Database Engine—its databases are just text files—it nevertheless demonstrates the basic techniques to use in an actual database application.

Load ADDRBOOK and select File|Open from the main menu, then select the SAMPLE.ADR file supplied with Orpheus. You'll see something like this:

The first page of the TOvcNotebook, the Directory page, contains a TOvcVirtualListbox. This listbox shows all the records in the database.

To insert a new record into the database, click on the Insert button on the tool bar, press <Ins>, or select Edit|Insert from the main menu.

To delete a record, click on the Delete button on the tool bar, press <Del>, or select Edit|Delete from the main menu. You must confirm that you want to delete the record.

To edit a record, click on the Client Info or Notes pages, move the highlight bar to the record and press <Enter>, or double-click on the item. Switching back to the Directory page of the notebook or moving to a different record using the buttons on the toolbar automatically saves your changes.

When you insert or edit a record, the second page of the TOvcNotebook, the Client Info page, is shown:

Several of the edit components on this page of the notebook are TOvcPictureField components. They are not all described here, but the more noteworthy ones are.

At the top of the Name and Address box is a TO32FlexEdit component set to read-only mode. At the first glance it appears to contain only the first and last names of the entry, but hovering your mouse over the field you can see that it actually contains much more.

The State field uses a custom validation routine to ensure that the field is either empty or contains a valid abbreviation. It also demonstrates how to use a TOvcSpinner component with a field that doesn't know how to increment or decrement its value automatically.

The Zip field uses a custom validation routine to ensure that either the entire field is empty, or the first subfield is filled in completely and the second subfield is either empty or filled in completely.

The two Phone Number fields use custom validation routines to ensure that the first subfield (the area code) is either empty or filled in completely. The validation routines also ensure that either the entire field is empty, or the last two subfields are filled in completely.

The two Important Date fields both have TOvcFourWaySpinner components. They can also be filled in using a calendar dialog, which is invoked by double-clicking on the date field or pressing the <F2> key. The calendar dialog looks like this:

To scroll through the calendar month-by-month, click on one of the arrows on the scroll bar or use <PgUp> and <PgDn>. The scroll bar's thumb indicates the position of the current month within the year. To scroll through the calendar year-by-year, click on the buttons to the right of the calendar or use <CtrlPgUp> and <CtrlPgDn>. The calendar dialog is designed to be general-purpose, and you can use it as-is in your programs or modify it as necessary.

The third page of the notebook component, the Notes page, has a memo field based on the TOvcEditor component. The data in the memo field corresponds to the current record. Switching back to the Directory page of the notebook or moving to a different record using the buttons on the toolbar automatically saves changes to the memo field.

The ADDRBOOK demo uses Orpheus' facilities for determining whether a record was modified during editing. If it was, the address book flags itself as modified, and this allows it to prompt whether the changes should be saved to disk before changing address books or exiting from the program.

Order Entry

The main purpose of the ORDENTRY program is to demonstrate a table control. It is very similar to ADDRBOOK, but its data entry page contains a table control. To see the table control, load ORDENTRY and select the Order Entry page:

The top half of this screen is made up of picture entry fields that are very similar to the ones used in ADDRBOOK. The bottom half contains the table control. The table control has a vertical scroll bar that allows you to move through the records (or you can use the cursor keys). It does not have a horizontal scroll bar because all the columns are visible in the window.

To enter an order, click on the Insert button on the tool bar, press <Ins>, or select Edit|Insert from the main menu. Enter the information in the top half of the entry screen and then click on the first cell in the table control or press <Tab> to go to the table control. The Qty column is a simple entry field with a range of 1-255. Enter the quantity for the record and press <Enter> to move to the Description column. This column is also a simple entry field that allows you to enter up to 20 characters to describe the item. Press <Enter> to move to the Unit Price column and enter a value between 0 and 999.99. When you press <Enter> to exit this field, notice that the total price (Quantity*Unit Price) is automatically calculated and displayed in the Total column and the cursor moves to the next record.

If you need to leave the table control and return to any of the picture entry fields above it, click on the desired control with the mouse or press <Tab> until the desired control is focused.

You can save the orders that you enter by selecting File|Save in the main menu. If you do not specify an extension, the default extension ORD is used.

To edit an existing order file, select File|Open. A line containing the date, company name, and person's name is displayed in the listbox for each record. If all the records in the file do not fit in the window, a vertical scroll bar is displayed. To edit a record, select Edit|Modify from the main menu, double click on the record, or switch to the Order Entry page when the record is highlighted.

To delete the currently highlighted record, click on the Delete button on the tool bar or select Edit|Delete from the main menu.

Text Viewer

TEXTVIEW implements a viewer component that allows you to view ASCII text files. The size of the viewed file is limited by available memory. TEXTVIEW uses the TOvcTextFileViewer component (see page 852).

TEXTVIEW is intended to be used strictly with ASCII text files. If you need to view binary files, use the FILEVIEW demonstration program instead (see page 1283).

You can load a file to be viewed by choosing the Open item from the File menu. For this discussion, the Pascal source file for the main form of TEXTVIEW itself will be viewed. Enter the pathname of TXTVIEW1.PAS in the Open dialog to display the source file:

All the viewer commands supported by the TOvcBaseViewer class are available in TEXTVIEW. See “Viewer Commands” on page !!! for a list of the commands.

The TEXTVIEW main menu provides the following commands (accelerator keys, if any, are shown after the command):

File

Open

Open a file for viewing. Only one file can be open at a time.

Print

Print the entire file or just the selected portion.

Print setup

Allows you to select from the list of installed printers or change a printer setup.

Exit <AltF4>

Exit TEXTVIEW.

Clipboard

Copy <CtrlC>

Copy selected text to the clipboard for use by another application. Another common hotkey for Copy, <CtrlIns>, is supported by using a hidden menu item.

Select all

Select the entire ASCII text file.

Search

Find

Search for the specified text. A modal dialog box is displayed. The search options are Match case (case-sensitive search) and Search forward or backward. Set the desired options, enter a search string, and select OK to perform the search. If the search string is found, the search can be repeated by pressing <F3>. Once the dialog is closed, the last search operation can be repeated either by selecting the Find next menu item or by pressing its associated accelerator key, <F3>.

Find next <F3>

Repeat the previous search.

Go to line <F4>

Move the cursor to the specified line.

Options

Tab expansion

Tabs are automatically expanded to spaces on the screen. Turn this option off if you want tabs treated as ordinary characters (their appearance on the screen depends on the font used).

WordStar commands

To use WordStar commands for cursor movement, turn this option on. For a list of the WordStar commands, see “Viewer Commands” on page 711.

Tab size

Change the tab size (the default is 8 characters).

Font

This option demonstrates how to set the font for a viewer control. Selecting this command displays the Font common dialog. The Font dialog is set to allow only fixed- pitch fonts because the design of the viewer depends on fixed character spacing.

Help

About

Display information about the TEXTVIEW demonstration program.

The status line at the bottom of the TEXTVIEW window consists of several Panel components (or a Status Bar control in the C++Builder examples). The status line displays the line and column of the cursor, the total number of lines in the file, the number of the line at the top of the window, and the number of bytes in the file.

File Viewer

FILEVIEW is very similar to TEXTVIEW, but it allows you to view binary files in addition to ASCII text files. Files of up to 2 billion characters can be viewed. FILEVIEW uses the TOvcFileViewer component (see page !!!).

You can load a file to be viewed by choosing the Open item from the File menu. For this discussion, the executable file for FILEVIEW itself will be viewed. Enter the pathname of FILEVIEW.EXE in the Open dialog to display the source file. Since this is a binary file, select Options|Hex mode to view it in hex mode:

When hex mode is on, the first column contains the offset in the file, the center area shows the hex values, and the last column shows the ASCII representation of the hex values. If you use the Clipboard|Copy function while in this mode, be aware that what you copy is exactly what you see on the screen. That is, you copy the offset, hex values, and ASCII values shown on the screen. If you want to copy the actual binary file contents, turn off hex mode before selecting the area to be copied.

All the viewer commands supported by the TOvcBaseViewer class are available in FILEVIEW. See “Viewer Commands” on page !!! for a list of the commands.

The main menu for FILEVIEW is the same as the main menu for the TEXTVIEW demonstration program (see page !!!) with the addition the Hex mode option in Options.

The status line at the bottom of the FILEVIEW window is a Header component. It displays the line and column of the cursor, the total number of lines in the file, the number of the line at the top of the window, and the number of bytes in the file.

Text Editor

TEXTEDIT implements an editor component that allows you to edit ASCII text files. The size of the edited file is limited by available memory. TEXTEDIT uses the TOvcTextFileEditor component (see page !!!).

You can load a file to be edited by choosing the Open item from the File menu. For this discussion, the Pascal source file for the main form of TEXTEDIT itself will be edited. Enter the pathname in the Open dialog to display the source file.

All the editor commands supported by the TOvcEditor component are available in TEXTEDIT. See “Editor Commands” on page !!! for a list of the commands.

The TEXTEDIT main menu provides the following commands (accelerator keys, if any, are shown immediately after the command):

File

New

Open a new file for editing. Only one file can be open at a time.

Open

Open an existing file for editing. Only one file can be open at a time.

Save

Save an edited file.

Save as

Save an edited file under a new name.

Print

Print the entire file, the selected portion only, or a range of pages.

Print setup

Select from the list of installed printers or change a printer setup.

Send (32-bit only)

Sends the current document using the default mail client.

Exit <AltF4>

Exit TEXTEDIT.

Edit

Undo <AltBkSp>

Undo the last edit command. Multiple edit commands can be undone.

Redo <ShiftAltBkSp>

Redo the last edit command (undo an undo).

Cut <ShiftDel>

Copy the selected text to the clipboard and delete it.

Copy <CtrlIns>

Copy the selected text to the clipboard.

Paste <ShiftIns>

Copy the text from the clipboard to the location of the cursor.

Delete <Del>

Delete the selected text.

Search

Find

Search for the specified text. A modeless dialog box is displayed. The search options are Match case (case-sensitive search), Whole words only (ignore embedded match strings), Search forward or backward, and Search globally or Selected text only or Start at cursor. Set the desired options, enter a search string, and select the default button, Find next. If the search string is found, the search can be repeated by selecting Find next again. The dialog box remains on screen until you select Cancel. Once the dialog is closed, the last search operation can be repeated either by selecting the Find next menu item or by pressing its associated accelerator key, <F3>.

Replace

Search for the specified text and replace it. A modeless dialog box is displayed. Set the desired options, enter search and replacement strings, and select the Find next button to find the next occurrence of the search string. If a match is found, you can select Find next to search for the next match, Replace to replace the highlighted match string and search for the next one, or Replace all to replace all matching strings from that point on.

Find next <F3>

Repeat the previous search.

Go to line <F4>

Move the cursor to the specified line.

Options

Auto indentation

Indents a new paragraph with the same indentation level as the last line of the previous paragraph. Since the second and subsequent lines of a paragraph cannot be indented, this option is typically desirable only when word wrap is off.

Create backup file

If this option is on (as it is by default), a backup file is created by renaming the existing file before saving. For example, if you save MYFILE.DOC and a file by that name already exists, the existing file is renamed to MYFILE.BAK before the new MYFILE.DOC is created. If a MYFILE.BAK already exists, it is deleted.

Font

This option allows you to choose a font using the Font common dialog. The Font dialog is set to allow only fixed-pitch fonts because the editor component cannot work with variable-pitch fonts.

Tabs

This option allows you to choose fixed, real, or smart tabs. Fixed tabs means that spaces are inserted in the text when you press the tab key (the position of existing text will not change if the tab size is later changed, as it would if real tabs were used). Real tabs means that a tab character is inserted in the text when you press the tab key. Smart tabs means that spaces are inserted in the text when you press the tab key, but the positions of the tab stops are a function of the positions of the words in the previous line. See “Tabs” on page !!! for more information on tab types.

This option also allows you to set the tab size (the valid range is 2 to 20). The default tab size is 8 characters.

Tab expansion cannot be turned off in an editor control, so tab characters in the text stream are always expanded to spaces on the screen.

Word wrap

Turn word wrap on and off.

Wrap at column

Change the column for word wrap (the valid range is 5 to 32767). The default is column 80.

WordStar commands

To use WordStar commands for cursor movement, turn this option on. For a list of the WordStar commands, see “Editor Commands” on page !!!.

Help

About

Display information about the TEXTEDIT demonstration program.

MDI Editor

MDIEDIT is very similar to TEXTEDIT, except that it allows you to edit multiple files or different parts of the same file. It implements editor components to allow you to edit ASCII text files. The size of the edited file is limited only by available memory and the number of files open. MDIEDIT uses the TOvcTextFileEditor component (see page !!!).

You can edit two portions of the same file by splitting the edit window into two panes using the Window|Split menu item. The size of the edit panes can be adjusted by dragging the middle bar with the mouse.

You can load a file to be edited by choosing the Open item from the File menu. For this discussion, three Pascal source files will be edited. Enter the pathnames of MDIEDITA.PAS, MDIEDITC.PAS, and MDIEDITM.PAS in the Open dialog. Note that you can also open all three files at once in the file listbox of the Open dialog by selecting them all.

After you have opened the files, the screen should look something like this (the top level window was split using the Window|Split menu option):

All the editor commands supported by the TOvcEditor component are available in MDIEDIT.

The main menu for MDIEDIT is the same as the main menu for TEXTEDIT (see the description on page 1284), with the addition of the following:

Window

Cascade

Display all open windows on the screen in a cascade pattern.

Tile

Display all open windows on the screen in a tiled pattern.

Arrange Icons

Arrange the icons of all minimized windows.

Minimize all

Minimize all windows.

Split

Split the editor window to allow editing of two portions of the same file.

Identifier Index

Subject Index

Subject Index