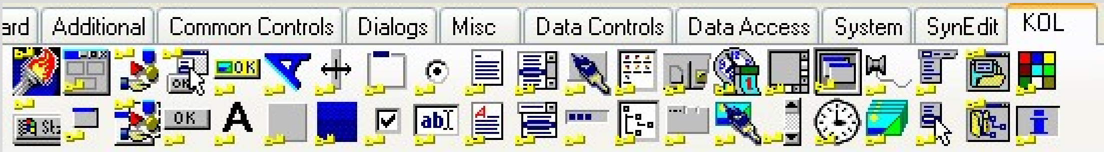
KOL/MCK - User Guide



© 2024 Carl Peeraer

KOL / MCK & Documentation created by Vladimir Kladov

1. Foreword 13
1.1 Vladimir Kladov ... 15
1.2 What's new? ... 15
2. Introduction 17
2.1 KOL Start ... 19
2.1.1 KOL architectural concepts ... 19
2.1.2 Further development of KOL. ... 22
2.2 First conclusions ... 24
2.2.1 Save memory costs ... 25
2.3 Mirror Classes Kit ... 25
2.4 Search for information... ... 26
2.5 Compatibility with VLC projects ... 27
2.6 KOL and the CBuilder compiler ... 28
3. Installing KOL and MCK 31
3.1 Installing KOL ... 32
3.2 Installing MCK ... 32
3.3 KOL64 and Free Pascal ... 33
3.4 Conditional Compilation Symbols ... 37
4. Programming in KOL 47
4.1 String Functions ... 49
4.1.1 String Functions - Syntax ... 50
4.2 Working with long integers & Floating Point ... 59
4.2.1 Long Integers & Floating Point - Syntax ... 60
4.3 Working with Date and Time ... 62
4.3.1 Date and Time - Syntax ... 63
4.4 Files and Folders ... 67
4.4.1 Files and Folders - Syntax ... 70
4.5 Working with the Registry ... 78
4.5.1 Registry functions - Syntax ... 79
4.6 Working with Windows ... 81
4.6.1 Working with Windows - Syntax ... 82
4.7 Messageboxes ... 86
4.7.1 Messageboxes - Syntax ... 86
4.8 Clipboard Operations ... 88
4.8.1 Clipboard Operations - Syntax ... 88
4.9 Arithmetics, geometry, utilities ... 89
4.9.1 Arithmetics, geometry, utilities - Syntax ... 89

 © 2024 Carl Peeraer

4.10 Sorting Data ... 91
4.10.1 Sorting Data - Syntax ... 91
4.11 Object Type Hierarchy ... 92
4.11.1 _TObj and TObj objects ... 92
4.11.1.1 TObj - Syntax ... 94
4.11.2 Object inheritance from TObj ... 96
4.11.3 Event Handlers ... 98
4.12 TList Object (Generic List) ... 100
4.12.1 Speeding up work with large Lists ... 102
4.12.2 TList Object - Syntax ... 102
4.13 Data Streams in KOL ... 105
4.13.1 Data Streams - Syntax ... 108
4.14 List of Strings ... 115
4.14.1 List of Strings - Syntax ... 118
4.15 List of Files and Directories ... 126
4.15.1 List of Files and Directories - Syntax ... 127
4.16 Tracking Changes on Disk ... 130
4.16.1 Tracking Changes on Disk - Syntax ... 130
4.17 INI Files ... 131
4.17.1 INI Files - Syntax ... 133
4.18 An Array of Bit Flags ... 135
4.18.1 An Array of Bit Flags - Syntax ... 136
4.19 Tree in Memory ... 137
4.19.1 Tree in Memory - Syntax ... 138
4.20 Elements of Graphics ... 141
4.20.1 Elements of Graphics - Syntax ... 146
4.20.2 TCanvas - Syntax ... 147
4.20.3 TGraphicTool - Syntax ... 151
4.20.4 Color Conversion - Syntax ... 155
4.21 Image in Memory ... 156
4.21.1 The methods and properties of the TBitmap object ... 157
4.21.1.1 Pixel descriptor and format ... 157
4.21.1.2 Dimensions ... 158
4.21.1.3 Loading and Saving ... 158
4.21.1.4 Drawing an Image in a different Context ... 159
4.21.1.5 Canvas and modification of your own image through it ... 159
4.21.1.6 Direct access to pixels and image modification without canvas ... 160
4.21.1.7 DIB image parameters ... 161
4.21.2 Image in Memory - Syntax ... 161
4.22 Pictogram ... 170
4.22.1 Pictogram - Syntax ... 171
4.23 List of Images ... 174
4.23.1 The methods and properties of the TImageList object ... 175

 © 2024 Carl Peeraer

4.23.1.1 Descriptor and parameters 175
4.23.1.2 Image manipulation: add, remove, load 175
4.23.1.3 Accessing images 176
4.23.1.4 Drawing 176
4.23.2 List of Images - Syntax 177
4.24 Before getting started with Visual Objects 183
4.25 Common Properties and Methods - TControl 184
4.25.1 Properties and Methods of window objects 185
4.25.1.1 Window handle 185
4.25.1.2 Parent and Child controls 186
4.25.1.3 Availability and visibility 187
4.25.1.4 Position and dimensions 188
4.25.1.5 Painting 190
4.25.1.6 Window text and font for the window 191
4.25.1.7 Window color and window frame 191
4.25.1.8 Messages (all window objects) 192
4.25.1.9 Dispatching messages in KOL 193
4.25.1.10 Keyboard and tabs between controls 196
4.25.1.11 Mouse and mouse cursor 197
4.25.1.12 Menu and Help 198
4.25.1.13 Form and applet properties, methods, and events 198
4.25.1.13.1 Appearance (form, applet) 198
4.25.1.13.2 Messages (form, applet) 200
4.25.1.13.3 OnFormClick event (for form) 201
4.25.1.14 Modal dialogs 202
4.25.1.15 Reference system 202
4.25.2 Common Properties and Methods - Syntax 203
4.26 Programming in KOL (without MCK) 279
4.27 MCK Design 281
4.27.1 Creation of on MCK project 281
4.27.2 Form customization 285
4.27.3 Coding 287
4.28 Application graphic resources 287
4.29 Graphics Resources and MCK's 288

5. Window Objects 291

5.1 Labels (label, label effect) 293
5.2 Panel (Panel, Gradient Panel, Gradient Style) 294
5.3 Groupbox 296
5.4 Paintbox 296
5.5 Imageshow 297
5.6 Splitter 298
5.7 Scrollbar 299
5.8 Progressbar 300
5.9 Scrollbox 300

 © 2024 Carl Peeraer

5.10 Buttons ... 301

5.11 Switches (Checkbox, Radiobox) ... 304

5.12 Visual objects with a list of items ... 304

5.13 Text input fields (editbox, memo, richedit) ... 305

5.13.1 Text input field constructors (edit) ... 306
5.13.2 Specifics of using common properties (edit) ... 306
5.13.3 Input field options (edit) ... 307
5.13.4 General properties of input fields (edit) ... 308
5.13.5 Empowering: direct API access (edit) ... 309
5.13.6 Features of Rich Edit ... 309
5.13.7 Mirrored input field classes (edit) ... 314

5.14 List of Strings (Listbox) ... 314

5.15 Combobox ... 316

5.16 General List (List View) ... 318

5.16.1 List Views ... 320
5.16.2 Column management ... 320
5.16.3 Working with items and selection ... 321
5.16.4 Adding and removing items ... 322
5.16.5 Element values and their change ... 322
5.16.6 Location of items ... 323
5.16.7 List view ... 324
5.16.8 Sorting and searching ... 324

5.17 Tree View ... 325

5.17.1 Properties of the whole tree ... 327
5.17.2 Adding and removing nodes ... 327
5.17.3 Properties of parent nodes ... 328
5.17.4 Properties of child nodes ... 328
5.17.5 Node attributes: text, icons, states ... 328
5.17.6 Node geometry and drag ... 329
5.17.7 Editing text ... 329

5.18 Tool Bar ... 330

5.18.1 General properties, methods, events ... 333
5.18.2 Setting up the ruler ... 334
5.18.3 Button properties ... 335
5.18.4 Some features of working with the toolbar ... 335

5.19 Tab Control ... 336

5.20 Frames (TKOLFrame) ... 339

5.21 Data Module (TKOLDataModule) ... 340

5.22 The Form ... 341

5.23 "Alien" Panel ... 342

5.24 MDI Interface ... 342

 © 2024 Carl Peeraer

5.25 DateTime Picker 344
5.26 Visual objects - Syntax 344
5.26.1 Function NewLabel 346
5.26.2 Function NewWordWrapLabel 346
5.26.3 Function NewLabelEffect 347
5.26.4 Function NewPanel 347
5.26.5 Function NewGradientPanel 347
5.26.6 Function NewGradientPanelEx 347
5.26.7 Function NewGroupBox 348
5.26.8 Function NewPaintBox 348
5.26.9 Function ImageShow 348
5.26.10 Function NewSplitter 348
5.26.11 Function NewScrollBar 349
5.26.12 Function NewProgressBar 350
5.26.13 Function NewScrollBox 350
5.26.14 Function NewButton 350
5.26.15 Function NewBitBtn 351
5.26.16 Function NewCheckBox 352
5.26.17 Function NewCheckBox3State 353
5.26.18 Function NewRadiobox 353
5.26.19 Function NewEditBox 353
5.26.20 Function NewRichEdit 354
5.26.21 Function NewListbox 358
5.26.22 Function NewCombobox 359
5.26.23 Function NewListView 360
5.26.24 Function NewTreeView 362
5.26.25 Function NewToolbar 364
5.26.26 Function NewTabControl 366
5.26.27 Function NewForm 367
5.26.28 Function NewApplet 369
5.26.29 Function NewMDIClient 370
5.26.30 Function NewMDIChild 370
5.26.31 Function NewDateTimePicker 370

6. Graphic Visual Elements 371

6.1 Graphic Label 373
6.2 Graphic Canvas for Drawing 374
6.3 Graphic Button 374
6.4 Graphic Flags 375
6.5 Graphic Input Field 375
6.6 XP Themes 376

7. Non-Visual Objects 377

© 2024 Carl Peeraer

7.1 Menu (TMenu) 379
7.1.1 Events for the entire menu or its child items 381
7.1.2 Events, methods, properties of an individual menu item as an object 382
7.1.3 Access to properties of subordinate menu items 383
7.1.4 Main menu 383
7.1.5 Pop-up menu 384
7.1.6 Accelerators 385
7.1.7 Menu at MCK 385
7.1.8 Menu - Syntax 386
7.2 Tray Icon (TTrayIcon) 394
7.2.1 Tray Icon - Syntax 395
7.3 File Selection Dialog (TopenSaveDialog) 397
7.3.1 File Selection Dialog - Syntax 399
7.4 Directory Selection Dialog (TOpenDirDialog) 401
7.4.1 Directory Selection Dialog - Syntax 403
7.5 Alternative Directory Selection Dialog (TOpenDirDialogEX) 404
7.5.1 Alternative Directory Selection Dialog - Syntax 407
7.6 Color Selection Dialog (TColorDialog) 409
7.6.1 Color Selection Dialog - Syntax 410
7.7 Clock (TTimer) 411
7.7.1 Multimedia Timer (TMMTimer) 413
7.7.2 Clock - Syntax 414
7.8 Thread, or thread of commands (TThread) 416
7.8.1 Thread - Syntax 419
7.9 Pseudo Streams 422
7.10 Action and ActionList 424
7.10.1 Action and ActionList - Syntax 425

8. KOL Extensions 429

8.1 Exception Handling 432
8.1.1 Exception Handling - Syntax 434
8.2 Floating Point Math 436
8.3 Complex Numbers 436
8.4 Dialogues 437
8.4.1 Font selection 437
8.4.2 Find and replace dialog 437
8.4.3 System dialogue "About the program" 437
8.5 Printing and Preparing Reports 437
8.5.1 Dialogs for choosing a printer and printing settings. 438
8.5.2 Printing reports 438
8.6 Working with Databases 439

© 2024 Carl Peeraer

8.6.1 KOLEDB 439
8.6.2 KOLODBC 440
8.6.3 KOLIB 441
8.6.4 KOLSQLite 441
8.6.5 Working with DBF files and other databases 441
8.7 Graphics Extensions 441
8.7.1 Metafiles WMF, EMF 442
8.7.1.1 Metafiles - Syntax 442
8.7.2 JPEG images 443
8.7.3 GIF Images, GIFShow, AniShow 444
8.7.4 KOLGraphic Library 446
8.7.5 Using GDI + (KOLGdiPlus) 446
8.7.6 Other image formats 447
8.7.7 Additional utilities for working with graphics 447
8.7.8 Open GL: KOLOGL12 and OpenGLContext modules 447
8.8 Sound and Video 447
8.8.1 KOLMediaPlayer 447
8.8.1.1 KOLMediaPlayer - Syntax 449
8.8.2 PlaySoundXXXX 459
8.8.3 KOLMP3 460
8.8.4 Other means for working with sound 460
8.9 Working with Archives 460
8.9.1 TCabFile 460
8.9.1.1 TCabFile - Syntax 460
8.9.2 KOLZLib 462
8.9.3 KOL_UnZip 462
8.9.4 KOLZip 462
8.9.5 DIUCL 462
8.9.6 KOLmdvLZH 463
8.10 Cryptography 463
8.10.1 TwoFish 463
8.10.2 KOLMD5 463
8.10.3 KOLAES 463
8.10.4 KOLCryptoLib 463
8.11 ActiveX 463
8.11.1 Active Script 464
8.12 OLE and DDE 464
8.12.1 KOL DDE 464
8.12.2 Drag-n-Drop 464
8.13 NET 464
8.13.1 Sockets and protocols 465
8.13.2 Working with ports 465
8.13.3 CGI 466

© 2024 Carl Peeraer

8.14 System Utilities ... 466
8.14.1 NT services ... 466
8.14.2 Control Panel Applet (CPL) ... 467
8.14.3 Writing your own driver ... 467
8.14.4 NT Privilege Management ... 467
8.15 Other Useful Extensions ... 467
8.15.1 Working with shortcuts, registering file extensions ... 467
8.15.2 Sharing memory between applications ... 467
8.15.3 Saving and restoring form properties ... 467
8.15.4 Additional buttons on the title bar ... 468
8.15.5 Macroassembly in memory (PC Asm) ... 468
8.15.6 Collapse Virtual Machine ... 469
8.15.7 FormCompact Property ... 470
8.16 Additional Visual Objects ... 470
8.16.1 Progress bar ... 470
8.16.2 Track bar (marked ruler) ... 471
8.16.3 Header (tables) ... 471
8.16.4 Font selection ... 471
8.16.5 Color selection ... 471
8.16.6 Disk selection ... 471
8.16.7 Entering the path to a directory ... 472
8.16.8 Selecting a file name filter ... 472
8.16.9 List of files and directories ... 472
8.16.10 IP Input ... 472
8.16.11 Calendar and date and / or time selection ... 472
8.16.12 Double List ... 472
8.16.13 Two-position button (up-down) ... 473
8.16.14 Button, non-rectangular ... 473
8.16.15 Extended panel ... 473
8.16.16 Label with image ... 473
8.16.17 Separator ... 473
8.16.18 Table ... 473
8.16.19 Syntax highlighting ... 473
8.16.20 GRush Controls ... 474
8.16.21 Other additional visual elements ... 476
8.16.22 Tooltips ... 477
8.17 XP Themes ... 477
8.18 Extensions of MCK itself ... 479
8.18.1 Improved font customization ... 479
8.18.2 Alternative component icons ... 479

9. Working with Extensions 481

9.1 Installing Extensions ... 482

© 2024 Carl Peeraer

9.2 Using Extensions **482**
9.3 Developing your own Extensions **483**
9.3.1 Development of non-visual extensions 483
9.3.2 Development of visual extensions (controls) 484
10. Appendix **487**
10.1 Errors of programmers starting to learn KOL **488**
10.2 Developer Tools **491**
10.3 Demonstration Examples **491**
10.4 KOL with Classes instead of Objects **494**

© 2024 Carl Peeraer

Foreword

Foreword from the author of this document: Carl Peeraer

1 Foreword

All this documentation was created by **Vladimir Kladov**, the designer of the **KOL/MCK** projects. Some contributions also come from other authors such as **Thaddy De Koning**.

This user guide is just an attempt to summarize the information about **KOL/MCK** in a format that can be easily used while programming. **Vladimir Kladov** 's KOL manual was translated from Russian by Carl Peeraer.

The user guide is made available as:

- Responsive Website: This version is currently hosted at https://www.artwerp.be, the website of Carl Peeraer, the compiler of this user guide. It is possible that this version will disappear sooner or later. Those who wish to save this information themselves can download the following formats:
- Executable Windows EXE file: This is a Windows EXE file, guaranteed virus-free, even though it might show a false positive in Virustotal.
- PDF that can be printed for reference
- CHM helpfile: After downloading the CHM file, in properties of the file, the block must be cleared. This is a limitation of Windows after downloading CHM files.
- EPUB EBook file: Book to read on an eReader such as KOBO - limited functionality!

As the creator of this user guide, I refer to the respective authors for further questions and information. Whenever this manual is written in the I form, it is **Vladimir Kladov** who is speaking. I have not changed this out of respect for the hard work **Vladimir Kladov** has put into programming KOL/MCK and all the documentation. Therefore, all credit always goes to him. The main source of information for the version of KOL / MCK being used is still the source code of the relevant files. Using the genius **xHelpGen** 183 utility, help files can be created in HTML format. Thus, this manual is in no way a substitute for the information contained in the source files of KOL / MCK!

There are many active links in this user guide, pointing to more information about the topic. However, not all topics contain links, but that has been solved by the **powerful search function** integrated in this guide.

I wrote this tutorial for myself, to make programming with KOL / MCK easier for myself. I don't know if this work can be of any use to others. If it is: use the document. If not: delete the document from your computer. Suggestions are welcome, but there is no guarantee of response or implementation of your suggestion(s). Negative remarks or comments will be ignored.

© 2024 Carl Peeraer
Version: 1.1.4

1.1 Vladimir Kladov

Vladimir Kladov

Age: 58 (at year 2024).
(~40 on the left photo)

Married, a son Alexander, 18 years.

Education: Novosibirsk State University, 1995. Mechanics, Applied Mathematics.

Job: Chief Engineer, Novosibirsk Chemical Concentration Plant (NCCP).

Experience: Delphi32, C++, C#, Assembler (TASM, MASM), FoxPro, SQL, ...

Travels: Moscow, St.Petersburg, Pushkin, Petergof, Kronstadt, Tikhvin, Staraya Ladoga, Paris, Hannofer, Bremen, Chabarovsk, Krasnodar, Sochi, Ghelendzhik.

Location: Novosibirsk, Russia.

1.2 What's new?

Here are the updates to this user-guide:

Version 1.1.0 - 17 August 2024:

- New **Web User Interface**:
  - forward and backward buttons to browse chapters smoothly
  - Better responsive mode with navigation buttons at the bottom of each page (for those who want to read on Smartphone)
- Download chapter, with all important **KOL/MCK** downloads

Version 1.1.1 - 19 August 2024:

- Cosmetic enhancements

© 2024 Carl Peeraer

Version 1.1.2 - 20 August 2024:

- Update in Clipboard Operations 88 chapter: free license for **MultiClipBoard** for readers of this User-Guide: https://www.artwerp.be/MultiClipboard/setup_multiclipboard.exe
- New Chapter TAction and TActionList 424

Version 1.1.3 - 21 August 2024:

- A mass of additional links to more detailed information
- Various cosmetic improvements

Version 1.1.4 - 5 November 2024:

- Extra functions added in chapter: Messageboxes 86: ShowQuestion, ShowQuestionEx, ShowMsgModal.
- Carl Peeraer, the author of this user guide has added a new program: **VrtDrive** 493 to link directories to a drive letter. This is a GUI replacement of the old **subst console** program. However, **VrtDrive** 493 will keep the assigned links even after logging off or rebooting the computer.

© 2024 Carl Peeraer

Introduction

Introduction by Vladimir Kladov, creator of KOL/MCK

2 Introduction

Water wears away the stone.
A rolling stone gathers no moss.
Patience and a little effort.
(Russian folk proverbs)

This description was conceived by me, the author of this library, and started by numerous requests from KOL users, and from those who would like to learn how to use this (I'm not afraid to be immodest) wonderful tool. First of all, I'll talk a little about the origins of the Key Objects Library, abbreviated: KOL, and let me use this word in masculine rather than feminine, even though it is a library. It's just that there is a consonant Russian word "stake" just of the masculine gender, and it's more convenient for me. Someone does not like this word too much, but what can you do about it, there is no comrade for taste and color, as they say.

A bit of history

Around 1996 or 97, I started thinking about moving from the surviving DOS platform to the Windows platform. At that time, the operating system Windows 95 had already gone into life, sweeping away the OS / 2 monster on its way, and filling almost the entire niche of personal computers. It was necessary to change, and for a programmer such a transition means, first of all, the need to choose a new tool for work. It seemed most natural to take Borland C ++ (version 4 or 5) and just in addition to what was already known, learn the Windows API. But now I understand it. And then it was not obvious. And my attempts to program in Windows one after another have not been crowned with success. I continued to sculpt DOS-style interfaces, because it was easier for me to use a bunch of my own work-ups than to learn something how to do a message dispatching loop between windows. And in general it was not clear: why such a cycle is needed? After all, my program is that when I want, then I output when I need to - then I expect input. At least that's what I thought then.

Eventually, when I had some free time, I started experimenting with new compilers designed specifically for the new environment (Windows 95 and Windows NT 3.5). And on the advice of my good friend Alexei Shadrin (admins - hello from programmers!), I also tried Delphi 2, which was just released. And then he was amazed at the simplicity of work and, most importantly, by the obvious logic of work in the IDE. (I was also pleased with the high speed of compilation of the code, I must pay tribute to Borland - I have never seen such a fast compiler).

For the sake of such convenience in work, I agreed to sacrifice attachment to C / C ++, and remember how to write code in Pascal. (At first, I was sick of the need to write: = instead of just =, and begin / end instead of curly braces *, but soon I got used to: =, and appreciated the obvious advantages of begin / end for people with non-100% vision , for us this is a much more convenient notation than curly braces, which are easy to confuse with normal ones, or even not notice at all). I will only add that the first working program was ready in a couple of days (!), And it worked perfectly in a multi-window environment, doing exactly

© 2024 Carl Peeraer

> what was required of it (printing payment orders, and what did you think is the most demanded software in the conditions spontaneously developing LLC and PE).
>
> From that moment on, I became a staunch supporter of Pascal, bought the books on Delphi needed for a beginner, even learned how to create components and made a couple of my own (as I remember, these were TCloudHint and TBaloonHint - to show tooltips in the form of intricate windows - clouds and the fact that comic book heroes use it to make speeches).
>
> But gradually I began to be very unhappy with one rather significant detail, namely: the size of the programs received. It turned out to be gigantic, and in order, for example, to upload my works on the Internet (and I assumed that I would be engaged in blooming), a rather thick channel was required. In addition, the disk space was also not rubber (I remind you: in those days a 40 megabyte hard drive was the norm, now 200 Gigabytes does not seem to be something excessive).
>
> I thought about this problem, and finally decided to make an alternative class library that would allow doing smaller programs. I named it XCL (eXtreme Class Library). It really was "extreme". Without fully understanding the true reasons for the monstrousness of Delphi programs, I, among other things, decided in the heat of the moment to abandon the use of the Windows API wherever possible. Those. windows were registered with Windows, but were used only as an underlay — all rendering and other interactions were handled by their own code. Surprisingly, however, the programs still came out smaller than the VCL. True, as we moved forward, the tasks became more complicated, and I did not manage to get to the implementation of my own TListView.

Thousands of hours of programming spent on XCL, although I barely used the API, taught me the basics of the Application Programming Interface. I finally realized my mistake when the project was almost a year old. And then I conceived and started another project - the Key Objects Library, in which the emphasis was precisely on using the windows' ability to draw themselves and process most of the messages on their own.

2.1 KOL Start

2.1.1 KOL architectural concepts

Analysis of the reasons for the cumbersome application size. KOL architectural concepts

Small spool but precious.
(Russian folk saying)

But before starting, I analyzed more carefully the possible reasons for the increase in the size of the code, and thought about various ways to prevent this situation in my library. The main reason for the cumbersome size of programs in which classes are used is the fact that some classes use others, those, in turn, others, and so on, and so on, to such an extent that it is no

longer possible to break the links. You specify in uses a link to the Forms unit, or to Dialogs, and that's all - your 350-400 Kilobytes are added to the program. This means that you need to create your own class hierarchy, in which you only use TObject as an ancestor for all your classes, and in no case address all the goodness that is ready for use in the VCL.

I decided to go even further and "remember" the very basics of Object Pascal. ("Remember" is in quotation marks, because I myself did not have to write in Object Pascal, as I mentioned above, I came to Delphi from C / C ++, and before that I took Pascal only exams at the university, and I strongly scolded this wonderful language, just not understanding its benefits). So, in Object Pascal there is the word object, which means nothing more than "structure" + "set of methods". This is what the classes were later born from. My experiments have shown that a simple object created using the word object saves program size (and most importantly, if classes are not used at all, but only objects, then a few more kilobytes of code from system modules are saved from the very beginning) of course.

First When destroying an object, unlike an instance of a class, you have to manually write code to destroy objects, strings, dynamic arrays that are fields of this object. It's not very convenient, but saving code is more expensive than having to do some manual work.

Secondly The construction of such objects looked very unusual. In order not to get confused in the future, writing something like new (List, Create); , I decided to make all the "constructors" of objects global functions of the form NewTypename (parameters): PTypename. (Just in case, I also made "constructors" inside the TControl object to create various kinds of visual objects, but apparently no one uses them, including myself).

Thirdly Unfortunately, thirdly, that is, about compatibility, I learned much later: in another Pascal compiler, Free Pascal, the word object was not initially supported. But there were people who, for the sake of being able to compile KOL programs in this popular, and what is important - free - compiler, forced, persuaded - I don’t know exactly how to say, finally did some work themselves, and from version 2.10 Free Pascal began to fully support object. Although it is a bit late, and by this time a solution had already been found: in the automatic conversion of KOL to classes, and KOL programs compiled perfectly in Free Pascal, even without the support of primitive objects in it.

Another side of this incompatibility is that there are some problems when viewing the values of object properties in the Watch List when performing step-by-step debugging. Delphi can show anything as the property value instead of the true value. The solution to this problem is to specify the internal name of the field instead of a property when possible (fCount instead of Count, for example).

There are still a number of differences in the use of objects versus classes: namely, since object is just a structure in memory, in order to organize a pointer to some object, it is necessary for each object type to provide a corresponding pointer type. (This is not required for classes, since the type of the class is already equivalent to the type of the pointer, i.e. the type of the class representative is the same as the type of the class itself). For the same reason, in object

© 2024 Carl Peeraer

methods, unlike class methods, the Self variable is not a pointer, but the structure of the object's fields, and to get a pointer, you need to use the operation of taking the @ address, writing @ Self wherever you need to pass or use a pointer the object itself for which this method is written. (In the implementation, this operation does not require additional code, since Self is passed to the method by reference,

So, I have become firmly established in the decision not to use classes (classes), but only objects (objects). My further research and experiments showed that, firstly, too many different object types, and any too branched tree of the inheritance hierarchy are unacceptable if you need to save code size. And secondly, splitting the code into modules also increases the size of the program. Although, in the second case - and not a lot, but saving means saving, and I decided to cram the entire library into one large source file, which was named so: KOL.pas. *

With the problem of reducing the tree of the inheritance hierarchy, I decided to fight the most radical means, namely: all visual objects are represented by the same TControl object type, directly derived from the TObj object type, which I created as the base type for my hierarchy. The constructors are used differently, the sets of methods sometimes overlap, sometimes quite dramatically differ from one kind of visual object to another, but in any case, the same object type is used. As a result, they all use the same copy of the virtual method table (vmt), less code duplication, and less virtual methods are required.

And I also used one very important technical trick that I invented while building XCL (at least for something my first library came in handy, although no, of course, without XCL and KOL there would not have been). Namely, when initializing objects, in no case should you initialize all possible fields (which, in turn, are objects). This operation should be postponed "for later" if possible. For example, when creating a visual object in KOL, the font for the window of this object is initialized to the smallest possible amount. That is, the font is "inherited" from the parent window object, while, in fact, a stub is called - a pointer to a function, which begins to point to a valid function only if at least one font parameter has been modified in the program. Of course, the likelihood that in his application the programmer will change the default font, in the usual case is great. But the case when a programmer uses KOL is unusual in itself: he says that the programmer does not want to add extra code to the program. And this means that the decision to initialize the fields should be postponed until the moment when such code is required in the application. Naturally, if there is such an opportunity.

In fact, the above technique would not have been possible without using a compiler that has the ability not to insert procedures and functions into the program code that are not referenced in the project (even if these procedures and functions are present in connected modules, involved classes / objects - if only they weren't virtual). This is exactly the ability Delphi has (Free Pascal too, but at the moment when I started KOL, there was no question of compatibility with Free Pascal, and there was no such compiler then - if I'm not mistaken). In Delphi, this ability is called smart-linking.

Unfortunately, this trick, as mentioned, does not work for virtual methods. As far as I understand this problem (and Delphi experts offered their understanding, and it sometimes differed), the reason is banally simple: since the reference to all virtual methods is already present in the vmt

© 2024 Carl Peeraer

table of virtual methods of the class / object type, the method is counted as used, even if in reality, it is never addressed. For example, if another class "B" is inherited from class "A", in which this method is completely overridden, and there is no call to this method of the ancestor "A", and only instances of this inherited class "B" are created in the project modules. Anyway, since the link is already in the vmt table, the method will be "credited". So I decided to use the virtual method mechanism with great care.

2.1.2 Further development of KOL.

Further development of KOL. We reduce everything we can. Replacing System.pas and other system modules

And he is no longer what he was in the beginning.
Other people's destinies, becoming his destiny,
They take him away ...
(Rilke)

Already at the initial stage of writing the library, I had the idea to cut the System.pas module as much as possible. If someone is not in the know, then this unit is, as it were, automatically added to the uses section of any unit, and it contains a set of functions that are needed practically (in fact, theoretically) always. And in particular, it contains the code that is responsible for handling exceptions, for allocating memory in the heap - the so-called Memory Manager - Memory Manager, functions for working with variants, with dynamic arrays, dynamic strings, etc.

Having obtained all possible information on the bottom of the barrel, I discovered that it is quite possible to write and "substitute" my own system module compiled with the help of the Delphi compiler itself. So I did this by doing this work for Delphi 5, the version I was using then (in fact, I still mainly use Delphi 5 when creating projects on KOL, with rare exceptions - when I need, for example, to use in assembler inserts MMX commands).

When creating my own version of system.pas, if possible, I didn't just "disable" and replace the standard methods with my own, but made them optional. These features are turned off by default, but there is always the option to turn them back on. For example, the standard memory manager is enabled by calling the UseDelphiMemoryManager procedure, the ability to work with console I / O is enabled by calling UseInputOutput, etc. Including a standard memory manager instead of my primitive adapter (wrapper) to Windows API functions (GlobalAlloc, GlobalFree, GlobalRealloc), which takes literally tens of bytes of code instead of several kilobytes, usually there is no need - unless the program requires constant work with allocating and reallocating memory in heap, for example, when dealing with dynamic ANSI strings.

Now, in almost any KOL project, it is enough to add a directory in the project options in the search path, which contains the compiled alternative modules system.dcu and others like it, and the program is immediately reduced by 9-11 Kilobytes. For a gigantic size of 300-400 Kilobytes of a typical application made in Delphi, this would not be too much of an effect, but for a KOL program with a size of up to 40-60 Kilobytes, this is already a very significant gain. (Later, various authors made adaptations of my rework of System.pas for other Delphi versions: 3, 4, 6, 7).

© 2024 Carl Peeraer

Note: to replace system modules, it is not necessary to actually replace those modules in the system libraries. Moreover, this way you will not replace anything. The supplied replacement files should be unpacked into a separate directory, and in the project options specify the path to this directory - this is the replacement.

Note: to replace system modules, it is not necessary to actually replace those modules in the system libraries. Moreover, this way you will not replace anything. The supplied replacement files should be unpacked into a separate directory, and in the project options specify the path to this directory - this is the replacement.

Already doing this job of shortening system modules, I came across writing code in inline assembly. The PC assembler was not a particular problem for me, although before that I had never had to deal with it. I had to fill in some gaps in my education, and after gaining some experience in translating Pascal code into assembler, I decided to make an alternative asm version of the code for almost all KOL functions, which could be reduced at least a little by this.

As a result of both improvements made - replacement of system modules, and translation of most of the code into assembler, the size of the minimum KOL project with one visual form stopped at 13.5 Kilobytes, and the minimum console application of the form

```
Program P1; {$ APPTYPE CONSOLE}
begin
end.
```

decreased to 6 kilobytes (if you insert a call to ShowMessage with the parameter 'Hello, world! "- a standard test for the size of the generated code - then the size of such a program turns out to be 6.5K).

Correction: since version 2.39, the size of the minimal KOL application with one form has been reduced to 12.5K, the minimum DLL using KOL - to 6K, the minimal console application using KOL - to 5.5K. The compilation was done in Delphi6. In recent versions, an application with an empty form takes only 11.5K.

In addition, ahead of the events, the Collapse project was recently completed, which reduces the code by about half, even compared to rewriting it into assembler, (although the effect becomes noticeable only for fairly large applications - from approximately 40 Kilobytes). Collapse uses the translation of a part of the Pascal code into the P-code of a virtual Collapse machine, which is emulated during program execution. For this, the P-code is converted into byte-code using a specially created P-compiler. To complete this fantastic project, it remains only to write your own Pascal compiler, which could turn almost any Pascal code into P-code.

© 2024 Carl Peeraer

2.2 First conclusions

First conclusions. The need to minify code: Who needs it?

The topic of writing minimal (in size) applications in Delphi is widely covered. Any beginner can easily find information on how to make their application small. Almost any source claims to ditch the VCL and write in a pure API (Application Programming Interface). The ability to use other libraries instead of VCL - ACL and KOL - is quite often mentioned. I am a convinced supporter of the fact that you do not need to write in a pure API, except for those cases when it is simply impossible to do the work differently.

Let me explain why. API functions are generally quite general, and the number and types of parameters that are used in API functions are prone to bugs. (For example, instead of an integer, in some cases, a pointer to a string of the PChar type may be passed in the same place). The code using a direct call to these functions looks cumbersome, inconvenient to read and modify. And, for example, drawing on the so-called DC (Device Context) by directly referring to the GDI (Graphic Device Interface) methods is a more than non-trivial art. It is still better in many cases to use encapsulations in objects or classes, even if they are simpler than VCL.

However, I am not forcing anyone to use the Key Objects Library. But experience shows that creating and maintaining applications on KOL is not at all more difficult than on VCL (you just need some practice, as well as to get started on VCL). At the same time, the size of KOL programs is quite comparable to what is possible to get by hand through the API.

Now to the question of why you should minify your application code at all. Why - everyone decides for himself. Or he wants to save his traffic and user traffic when uploading his application to the Internet. Or he writes ActiveX - an application that is loaded from the server to the client side (and it is desirable to reduce the load time as well). Or he is writing a CGI application and wants to offload the server, which will have to execute hundreds and thousands of such CGI applications per second. Of course, a large program with a size of 400 KB can also be considered a CGI application, but it will probably take several times more resources and time for the system to launch. Yes, virus and Trojan writers are another category of "programmers" (so to speak) who need to write small programs.

Recently, the "fashion" for creating applications as large as possible (as proof of their coolness) is fading away. Despite the fact that the majority of Internet users are no longer limited by the speed of Internet access (and these speeds only grow over time), many of them are beginning to understand that the size of an application has practically nothing to do with its capabilities, quality of performance, or ease of use. And it only speaks about the qualifications of a programmer (and here the relationship is just the opposite: a seasoned programmer, as a rule, will have much less code than a beginner).

© 2024 Carl Peeraer

2.2.1 Save memory costs

He's shriveled!
(parody of David Blaine)

(This paragraph was added on 10.2010 in conjunction with KOL / MCK release 3.00).

KOL was originally aimed only at saving code, not system resources such as memory. Starting from version 3.00, the code has been reworked so that the size of TControl object instances has been reduced by about 6 times. First of all, due to the fact that the fields used by various types of controls do not overlap, and they were combined using a structure with variable content (record CASE). In addition, more than 600 bytes occupied by method pointers for processing messages began to be created dynamically only for those controls for which message handlers were assigned (and this is often far from all controls, for example, many panels and labels, and input fields are completely dispensed with message handlers). Same, how the dynamic structure TCommandActions began to be created and used (about 80 bytes) - now the application allocates one such object for a separate kind of control, instead of storing this record inside the fields of each TControl instance. In addition, about 40 bytes of flags, which previously occupied one byte for each Boolean flag, were compressed into several bytes, with one bit being spent on the flag.

As a result, the memory consumption for the TControl instance has been reduced from more than 1.5 KB to 300-350 bytes, excluding the optional event block (depending on the set of compilation options used). It is assumed that such an approach, in the case of adapting innovations in the version intended for Windows CE (KOL-CE branch), will help create applications that are less demanding on RAM. But for the main branch of the KOL project, saving memory can also be useful.

2.3 Mirror Classes Kit

Development of GUI applications in KOL: Mirror Classes Kit 32

If the mountain does not go to Mohammed,
then Mohammed will come to the mountain.
(Arabic proverb)

It is impossible not to dwell on a very important topic indicated in the title of this chapter. Delphi programmers, sitting in the IDE, have long been accustomed to the fact that a project using the VCL library (and now CLX - "kylix") is very convenient to develop visually, i.e. by sketching components on the form and visually setting their properties in the Object Inspector.

Of course, in the first versions, KOL was not visual (the library was conceived as not visual). But at some point the X hour came, and under the influence of numerous demands from KOL users, I was forced to develop a set of visual ("mirror") classes - Mirror Classes Kit - it is called that. What it is. It is a set of Design Time classes that are only used to determine which KOL objects are used in a module at run-time.

© 2024 Carl Peeraer

These mirror classes themselves are not involved in the program at runtime, and do not even exist, but at the development stage they are engaged in generating code for initializing forms, initializing and launching the application. And that's all. Those. MCK works like an add-on or plug-in to the Delphi IDE, modifying project files so that when they are processed into machine code, the Delphi compiler "does not see" mirror classes, links to .dfm form resources, but compiles only the code generated by MCK mirrors in the process of setting them up by the programmer.

In fact, application development in MCK - unlike "pure KOL" (this term appeared, apparently, by analogy with VCL - against "pure API") in no way increases the size of the application, just simplifying the developer's work (so the analogy - very distant).

In addition to what has already been said, I note that MCK projects, unlike handwritten KOL projects, automatically support Collapse and FormCompact technologies. The Collapse technique is that when the Pcode conditional compilation symbol is included in the project options, the mirror classes automatically generate the P-code for the Collapse machine, and thus it is possible to somewhat reduce the code of any large KOL applications using MCK. The FormCompact technique is even simpler: just enable the FormCompact property in the Object Inspector, and the generated pseudo-code for creating the form starts being processed by the interpreter of this pseudo-code, automatically. Reducing the code, however, will be noticeable (in both cases) only for the case of a sufficiently large number of controls on the form.

2.4 Search for information...

You can't get a fish out of the pond without difficulty.
(Russian folk proverb)

Now let's discuss a topic that is very important for any developer using any development environment, any programming language, any library, any API. Namely, where to get information (where to get help on KOL, preferably in Russian - this is one of the most frequently asked questions). People are primarily interested in information about the list of available functions, object types, methods.

Let's agree right away: KOL.pas is the main source of information. The functions and object types themselves in the interface part are already information. In addition, they are almost always provided with comments (the comment is placed after the declarations, in brackets like {* ...} - this is done for the purposes of the help autogenerator, which I will talk about right now.

Secondly, if someone finds it more convenient, you can use the automatic help generator - the xHelpGen utility. All that is required for its operation is to place it in the directory with the KOL.pas file and other library modules (and there are some, since the entire KOL has long ceased to fit into one module), and run. The result will be a set of html files that can be viewed using your favorite browser. (The xHelpGen utility first appeared for XCL, and was later rewritten

© 2024 Carl Peeraer

in KOL). There are other KOL help files (recently there is a help in chm format, it takes up several megabytes).

Third, on the main KOL site http://f0460945.xsph.ru/ and on other sites (links to which can be found on this resource) there is a sea of information: articles, FAQ (FAQ, ie Frequently Asked Questions and Answers), demo projects. Including there is a fairly large number of applications, many of which are provided by the authors along with the source code. Those. there is where to learn, it's up to little: you just need to find the time and start learning. (Of course, if you really need it, no one is coercing).

Fourth, if you really have a question that turned out to be too tough for you, I recommend going to the github kol-mck page: https://github.com/ebta/kol-mck... There will always be people who can competently answer really serious questions (including if the question is asked in English). This, however, should not be taken as an invitation to immediately contact the forum with the question "how to install KOL". No one will like to explain to someone else the already detailed chewed and stated in the documentation.

And a little more about distributions. The KOL / MCK package and update to the latest version can be obtained from the main site, the address is given above. To perform the update, you must use the Updater utility (take in the same place). Since the KOL.zip and MCK.zip archives are quite weighty, I upload only every 10th version in its entirety, all intermediate ones are obtained using small update files. If someone has a wide enough channel, then it is possible to pick up the latest distribution from other sources - the addresses are indicated in the links in the Download section (Archives).

2.5 Compatibility with VLC projects

Compatibility and conversion issues for existing VCL projects

Due to the fact that the original syntax used in KOL projects is forced to follow the restrictions of objects, unlike classes, I did not worry too much about other syntax compatibility. For example, objects do not support for properties of the default modifier, i.e. in an object type representing a list, for example, you cannot set the Items property as the default and write just as easily for a TList in the VCL: MyList [i]. You always have to write MyList.Items [i].

With function names similar to those found in the standard SysUtils module, I also took the liberty of deviating from the standards. Using this module in a KOL project increases the size of the application by about 10-20 KB, so it's better not to use it. Most of the SysUtils function counterparts are in KOL itself, and you can use alternative modules that are more compatible in function names. In particular, I decided to call the functions a little differently also in order to keep the possibility of simultaneous access to functions from both KOL and SysUtils. (Of course, you could always use modifiers like SysUtils.IntToStr, but what's done is done, and shouldn't be changed now).

© 2024 Carl Peeraer

As for more complex objects, such as lists, trees, and others, in KOL you can often find analogs that provide no less, if not more, capabilities than the standard set of VCL classes. Moreover, I now prefer to do almost all the work in KOL just because for KOL, thanks to the dedicated work of many programmers, a huge variety of various components of a very different direction have been adapted, and they turn out to be more accessible than similar tools for VCL. At the very least, they are already easier to find than similar VCL solutions: they are centered on a few KOL sites, built in almost the same style as defined by the tighter KOL framework, easier to customize to work with, and easier to integrate into your KOL project. (And they are free and with source, which can be important too).

But there is no need to talk about full compatibility with VCL components from the very beginning. You may find that a similar property is called differently, and is, for example, not a property, but a function, or quite the opposite. In the case when in VCL to create your own thread class (for example) you had to create your own class inheriting from TThread and override the Execute method, in KOL it is enough to call the constructor of the PThread object in the project and assign the OnExecute event to it.

There are a lot of such incompatibilities. And this is not at all because I specifically wanted the KOL library to be incompatible with the VCL. I just wanted to keep the projects developed in KOL as small as possible. Therefore, there is no Run-time Type Information (RTTI) that Delphi is so proud of, and there are no components that load themselves from a thread, and therefore the syntactic differences are great. But the language remained the same - Pascal. And the compiler is the same, with all its advantages and disadvantages.

But KOL itself is just very well compatible (if you can say so at all), in terms of transferring projects from any version of Delphi to any other. Almost everything that works in KOL for Delphi 5 or Delphi 7 is also available for Delphi 3 and even Delphi 2. There are a few exceptions, for example, Delphi 2 and Delphi 3 do not support Unicode WideStrings to ensure compatibility to work with doubled integers, it is necessary to use functions specially made for this in the project.

Throughout the entire time that the KOL library was created and developed, many "converts" asked me a question about how they can turn a ready-made VCL project into a KOL-compatible one. Despite the fact that there are already several projects that convert a VCL application or VCL component to an analog for KOL, I recommend doing this work manually anyway. There are not many differences, in fact, between KOL and VCL, but they are quite varied, and it is better to control the modification process personally than to trust the machine. (No, I do not insist, you can always try, the attempt is not torture, especially since if it works out at least partially, then manual work after that can be reduced, and this is also a plus).

2.6 KOL and the CBuilder compiler

Sometimes programmers using the CBuilder environment for development ask the following question: is it possible to use KOL from CBuilder. There are no fundamental obstacles to this. The CBuilder compiler understands Delphi code. And although he refuses to work with objects in the old style of object Pascal, and only agrees on classes, this is also not a problem: for KOL you can

© 2024 Carl Peeraer

always make a version with classes, just run the corresponding batch file from the GlueCut package. (Only on the batch file it would be necessary to work, because the incompatibilities between the syntax perceived by CBuilder and what is obtained even in KOL with classes are much greater even than when switching to the old version of the Free Pascal compiler).

In fact, it is not this that confuses, but the large size of the runtime library. If you do not connect it, and make an application that can be transferred to any machine, then a minimal application like "Hello, World!" It already takes up 50KB, not 16, as in Delphi.

If you leave the compilation option with the use of these libraries enabled, then without the presence of such a library of about 1.5 megabytes in the system, the application will not be able to run. But the good thing about the KOL library is that applications built on the principle of "I carry everything with me" remain extremely small, and at the same time programming remains object-oriented.

And it is worth adding KOL.pas to the project, and the starting size of the application is like "Hello World!" immediately grows to 360KB, and this is with the debug information disabled. If you look at the .map file created by the linker, you will find a very large number of functions from KOL.pas, although none of them have been called yet. Either CBuilder does not support smart-linking, or it does not support it only for pluggable source code in Pascal, but the bottom line is the same: the whole point of using KOL in the CBuilder environment is lost.

© 2024 Carl Peeraer

Installing KOL and MCK

Installation instructions for KOL and MCK

3 Installing KOL and MCK

- Installing KOL 32
- Installing MCK 32
- Conditional Compilation Symbols 37

You can download KOL + MCK from this link: https://www.artwerp.be/kol/kol-mck-master_3.23.zip. Everything you need to get started with KOL / MCK can be found in this archive.

3.1 Installing KOL

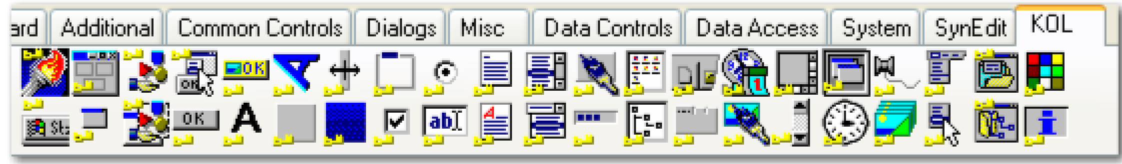
Installing the KOL library is as follows. First you need to create an empty folder. For example, let it be "C: \ KOL". And then unpack the contents of the KOL.zip archive into it. No other action is required. KOL is not a set of components for working in VCL, but a set of units that are included in a project by writing a reference to the unit used in the uses section of your unit or project. You just need to remember to write the path to the directory where the module is located in the project options (Project | Options | Directories / Conditionals | Search paths ...) or in the development environment options (Tools | Environment options ... | Library | Library paths).

3.2 Installing MCK

To install MCK, you need to unpack the contents of the MCK.zip archive (preferably into the same directory, answering “YES” to all questions about file replacement - some files in these archives are duplicated). Then you need to open the MirrorKOLPackageXX.dpk package from the Delphi IDE, while XX must correspond to the Delphi version (D3 for Delphi3, D4 for Delphi 4, D6 for Delphi6, D7 for Delphi7, and only for the Delphi5 version the extension is empty, i.e. the package carries name MirrodKOLPackage.dpk).

For Delphi, Borland Developer Studio, Turbo-Delphi versions, for which there is no package, but work with packages is supported, you can create an MCK package yourself. Three files should be added to the package: mirror.pas, mckObjs.pas, and mckCtrls.pas.

Then you need to click the Install button - on the toolbar of the package. If there is no ruler, the command is selected from the menu (in BDS, from the context menu on the project node). If you have problems with the installation, perhaps follow in the package options, or better - in Tools | Environment Options | Library | Library Path, add the path to $ (DELPHI) \ Source \ Toolsapi. This completes the MCK installation (during the first installation, you should see a very large list of installed components, all of them are installed in the KOL tab on the component bar).



© 2024 Carl Peeraer

Only in the Delphi2 version there is no need to open the package for installation, but a slightly different procedure is required (see the installation instructions, I do not intend to replace or duplicate it here).

Please note that when updating KOL and MCK to a new version (and this has to be done sometimes quite often, since the library is constantly evolving), you also need to open the MCK package, and perform Build (rebuilding) the package, but in no way the case is not Compile (recompilation). The fact is that when recompiling, unlike Build, the Delphi compiler erroneously uses the version of the pre-compiled KOL file that remains after working with current projects, and does not take into account the presence of significantly different compilation options in the package. As a result, after such an incomplete recompilation, the Delphi shell begins to malfunction, even to a permanent crash.

And note that when you go from working with a package back to working with applications, then at least the first time you should build again (Build), not compile (Compile) for the application. Otherwise, the compiler will again not notice that the conditional compilation character set has changed, and will try to use the KOL.dcu file generated when building the MCK package. And since in this case there were indirect references to VCL modules of design time, then Delphi will certainly require it to find the proxies.dcu file (which simply does not exist).

3.3 KOL64 and Free Pascal

Thanks to the **FreePascal (FPC)** compatible version of KOL, 64-bit programs can also be created with KOL.
You can download this version of KOL here: **https://www.artwerp.be/kol/kolx64.zip**

You can compile 32 bit and 64 bit programs with **FreePascal** and this version of KOL.

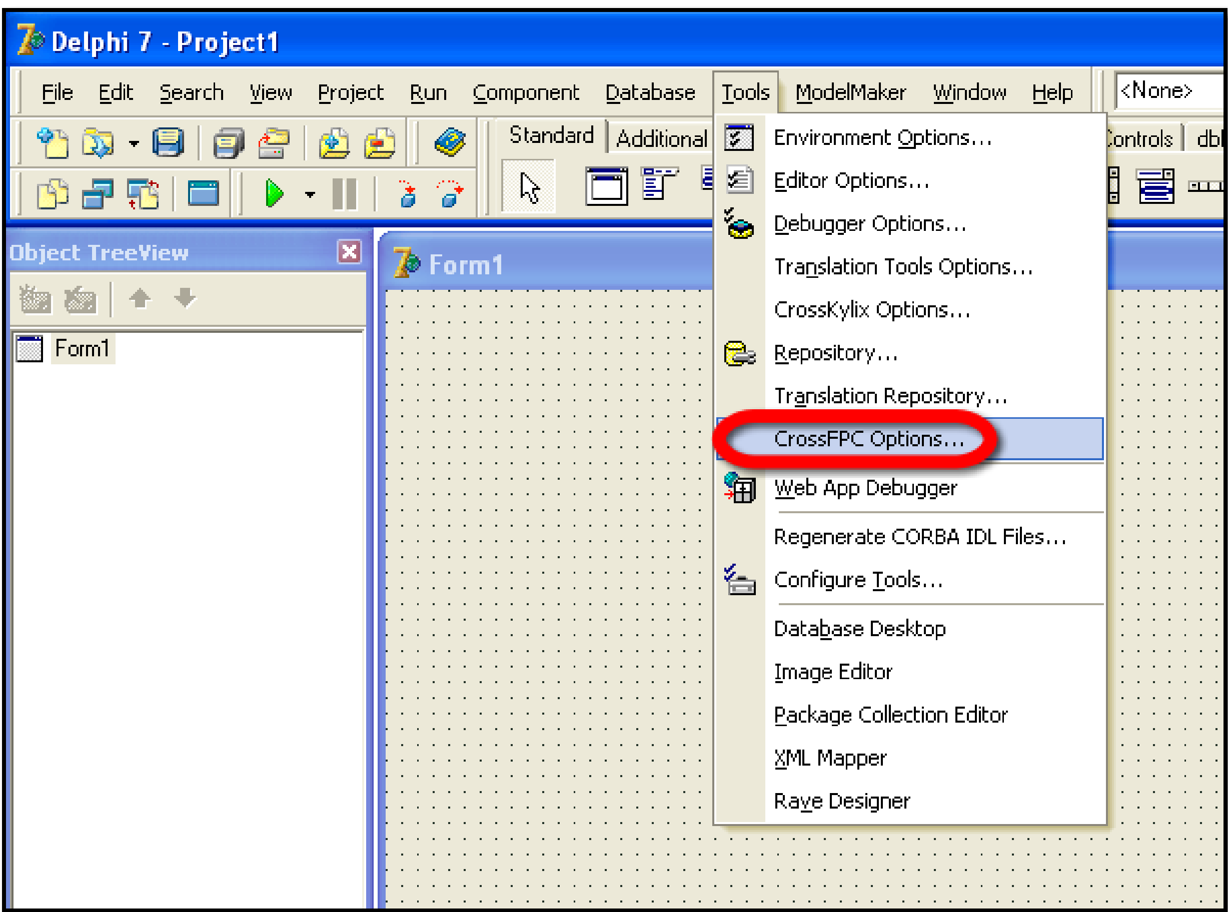
If you like to do this from your familiar Delphi environment, you can use **CrossFPC**. This program integrates the **FreePascal** compiler into Delphi. This works perfectly in **Delphi 7**, for example.

Here are the instructions for installation in **Delphi 7**:

- Download and install **CrossFPC** at this link: **https://www.artwerp.be/kol/crossfpc_setup.exe**
- After installation of **CrossFPC**, start **Delphi 7**.

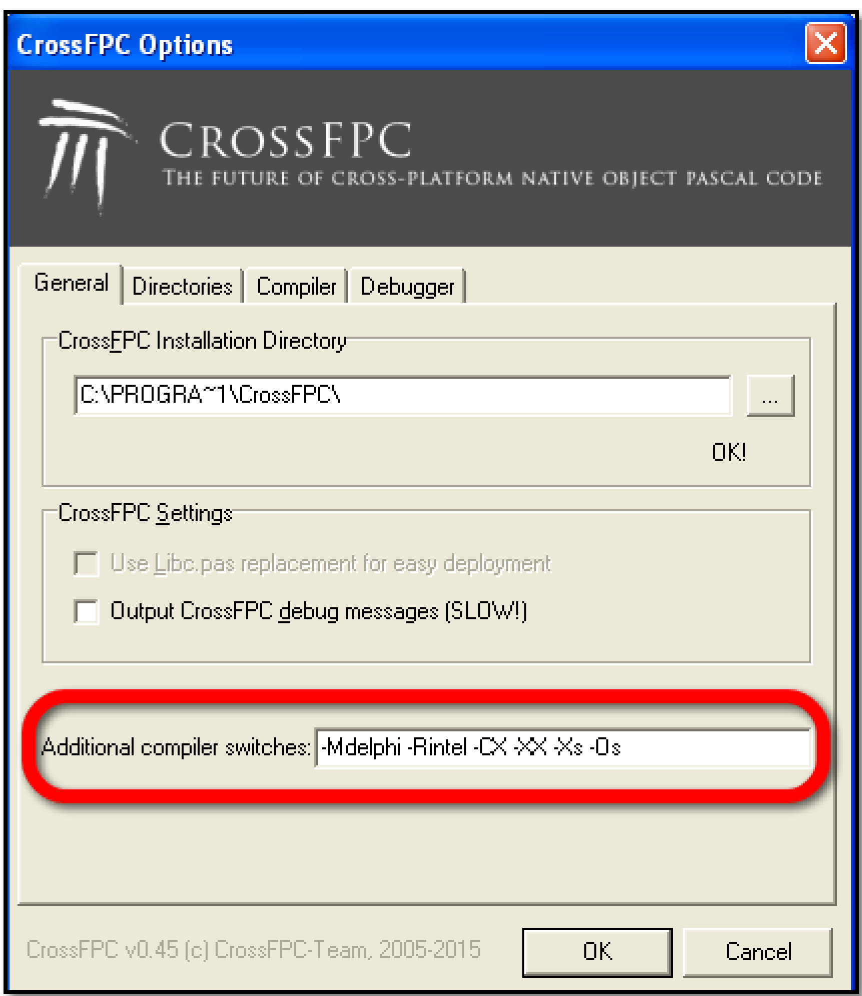
 © 2024 Carl Peeraer

- Open the **Tools menu** and select **CrossFPC Options**...



 © 2024 Carl Peeraer

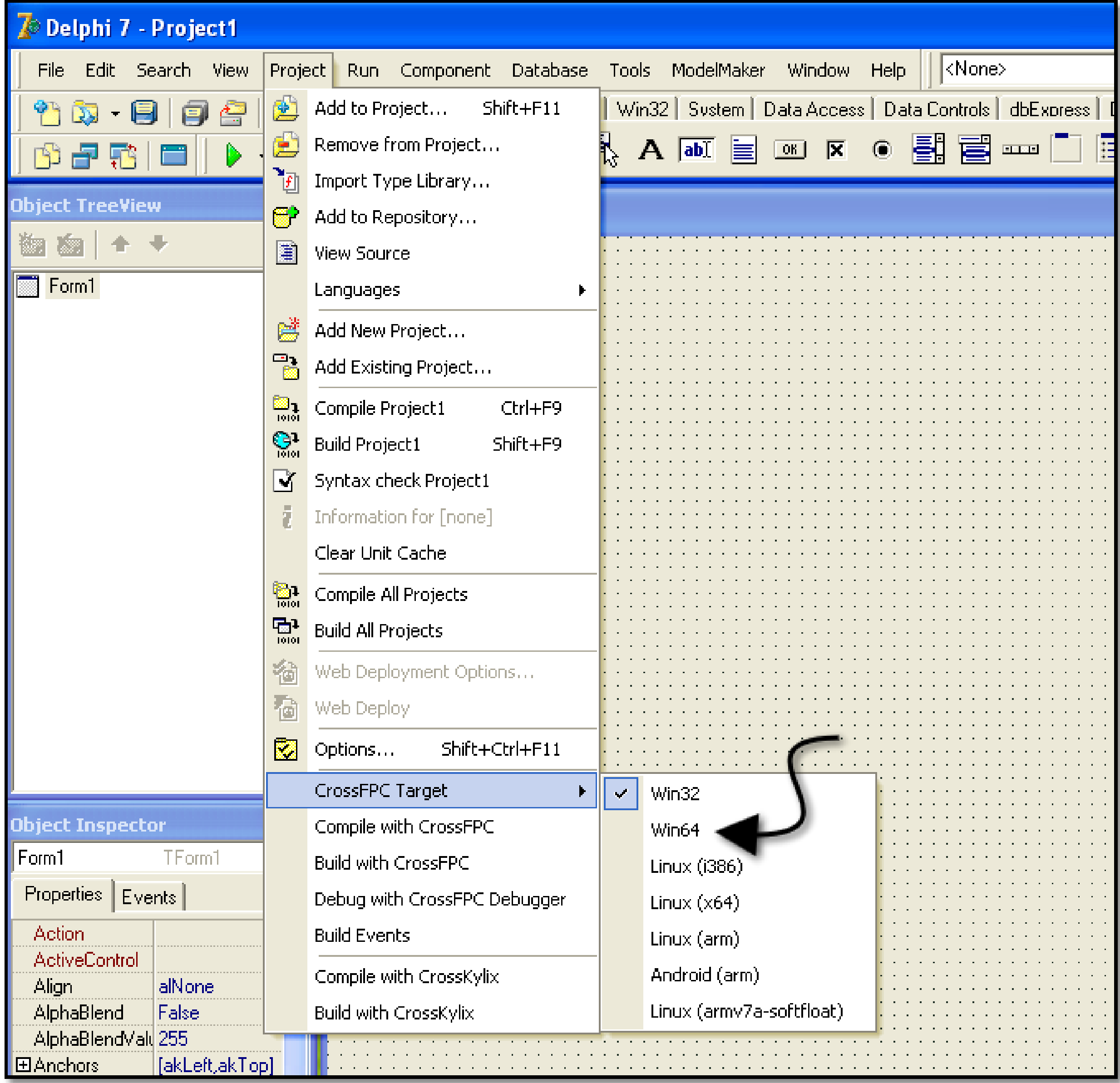
- This dialog box opens:



- If you wish to compile 64 bit programs with **CrossFPC** as well, you must write in the **Additional compiler switches** field: `-Mdelphi -Rintel -CX -XX -Xs -Os`

© 2024 Carl Peeraer

- Now, you can select **Win64** as **CrossFPC Target**, as in the picture below in the **Delphi Project menu**:



From now on, you can compile programs in Delphi KOL with:

- The Delphi native compiler
- FreePascal 32 bit
- FreePascal 64 bit

Important: Currently, KOL is not compatible with Linux by default. Therefore, these options cannot be used...

Carl Peeraer's **Piano Chords Maker** program is an example of a 64 bit program, written with **KOL / MCK**, compiled in Delphi 7, with **CrossFPC**.

© 2024 Carl Peeraer

3.4 Conditional Compilation Symbols

Since the KOL library is designed for a very wide range of tasks, it is almost impossible to satisfy all requirements using the same code, while keeping its size small. The way out is to allow the programmer to choose the code variants that he needs in his specific task. This is what the conditional compilation symbols are for.

To change the set of symbols or options that control conditional compilation, open the project properties dialog (Project | Options) and on the Directories / Conditionals tab correct the Conditional defines line by listing the required symbols separated by semicolons. *An alternative is to add {$ DEFINE option} declarations to the project file (DPR) for each option you want to use for the entire project. They say it works - I have not tested it.*

In MCK projects, the symbol KOL_MCK must always be present here, which "hides" the VCL code of the project from the Delphi compiler at the time of compilation. Some options may not apply to KOL, but to third-party modules. For example, some component packages for KOL are dual, that is, they can work with VCL and KOL. In this case, a conditional compilation symbol is often required to include a version of the code adapted for KOL (usually this symbol is the string 'KOL' - without quotes, but you still need to read the component description before using it in practice).

The list of options that can control sections of the code of the KOL library itself is located near the beginning of the KOL.pas file, with a brief description of the purpose of each symbol. Here I will try to give a detailed overview of these symbols, but it is almost impossible to describe everything without going into details. Therefore, further in the text, these symbols are mentioned more specifically in the context of each specialized function or object to which these symbols can be applied.

In most cases, adding one of the following characters to the project options list will result in some form of increment in the application code. In the opposite cases, I record this separately.

PAS_VERSION symbol (disable assembly code). Including this option in a project will significantly increase the size of the code, and make it somewhat slower. But there is a chance that the application will be more resilient. Basically, this symbol is intended for testing purposes and for identifying "bugs" in the assembler version. The Pascal code is largely self-documenting, and is actually kept in KOL, including as a commentary to the assembler version.

However, using the **PAS_VERSION** symbol is not a panacea. In the history of the development of the library, it often happened that the error lurked precisely in the Pascal version of the "guilty" procedure, while the assembler version was more correct.

The **PARANOIA** symbol is intended, on the other hand, to deepen the optimization of the assembler code - when using older Delphi compilers, version 5 and below. (Of course, this symbol only works if the project uses the assembler version of the library code). In these versions, the compiler used double-byte versions of some machine instructions, regardless of

the fact that there is a single-byte version for them, which is no different in functionality. Beginning with Delphi 6, Borland has removed this compiler flaw and is no er needed.
The **SMALLEST_CODE** symbol is designed to turn off everything you can do without by default. The form in this case will look very Spartan, since this also disables support for processing WM_CTLCOLORXXXX messages, which are responsible for coloring visual elements according to the developer's preferences, and the system fonts are used by default. (In this case, the color is provided by the system itself, its palette is not rich, but if functionality interests you more, then why not give the artist's functions to the system). Some checks are disabled in the code (for example: in the Int2Hex function it is no longer checked that the second parameter is> 15).
In fact, quite a lot of code fragments, mostly small ones, are disabled, and the savings in the code as a whole are very small. But it does take place, so I decided that the symbol should be.

The **SMALLEST_CODE_PARENTFONT** symbol, when using the above **SMALLEST_CODE**, nevertheless ensures that child visual objects inherit the font from their parents at the time of creation. If you install a different font for the parent than the system default for windows, then at least you don't have to repeat this operation for all its children. Thus, this option partially overrides the previous one, but only in this particular application to font inheritance.

The **SMALLER_CODE** symbol does almost the same thing as **SMALLEST_CODE**, but to a lesser extent, affecting the appearance and behavior of controls as little as possible.

The **SPEED_FASTER** symbol, included by default, increases the performance of some of the functions and algorithms used, at the expense of additional code. For example, the SortArray function is used to sort lists and strings, which increases the speed compared to using the more versatile SortData function by about 5-10% (in assembler version). To compare Ansi strings, the preliminary construction of an ordered set of Ansi-symbols is used, with the refusal of the subsequent call of API functions, after which the speed of comparison operations of the AnsiStrCompare and AnsiStrCompareNoCase functions increases several times. At the same time, this reduces the sorting of the list of strings StrList by several times (AnsiSort method). If improved performance is not required for these operations, this option can be disabled by adding the conditional compilation symbol SPEED_NORMAL.

The **TLIST_FAST** symbol changes the internal representation of lists. In the Delphi VCL, lists are actually arrays. When developing KOL, this approach was also initially taken as a basis, since provides high performance for small lists, and does not require a lot of code. With the TLIST_FAST option, the algorithms for working with lists and their internal representation are changed in such a way as to provide greater speed when inserting and deleting elements at arbitrary positions in the list. Namely, the elements are no longer stored as a solid array, but as a list of blocks with a maximum of 256 elements per block. This approach can, if used incorrectly, not only fail to increase performance, but, conversely, reduce it. For example, the operation of random access to the elements of the list at arbitrary indices can be relatively slow compared to the standard approach. To provide an opportunity for individual lists to keep the usual algorithm of operation, the list has the propertyUseBlockswhich can be installed in FALSEfor these lists. In addition, adding the DFLT_TLIST_NOUSE_BLOCKS option allows you to disable the TLIST_FAST option for default lists, and then assignUseBlocks TRUE for a select set of lists only.

© 2024 Carl Peeraer

The **USE_NAMES** symbol was added at the request of numerous programmers who are used to the fact that in the VCL every component has a Name property. Including this code adds this property to all objects starting with TObj. This makes it possible to search for components by name using the form's FindObj method. This option can also be useful for step-by-step debugging in order to make it easier to navigate in the context of code execution when stopping at an arbitrary point. MCK automatically generates the assignment of the component name to the visual and non-visual objects of the form, "hidden" in the parentheses of the conditional compilation {$ IFDEF USE_NAMES}. So, in the case of MCK projects, no additional effort is required to naming the components: it is enough to include this option in the project.

The **USE_CONSTRUCTORS** symbol was originally intended to use Delphi constructors for initial object initialization instead of using NewXXXXX's own constructing functions. Since there are no requirements from users to maintain this mechanism, support for the correctness of the code generated if this option is enabled is not guaranteed. In short: it is better not to use this option.

The **USE_CUSTOMEXTENSIONS** symbol is intended for those programmers who want to include their own additions in the KOL library, namely in the KOL.pas module itself, and precisely in the code of the TControl object. Position the cursor over this symbol, and press Ctrl + F in order to find all uses of this symbol in the KOL.pas module. You will find that it is used three times. 1) To add arbitrary code from the CUSTOM_TCONTROL_EXTENSION.inc file to the TControl object definition, 2) add some declarations to the interface section from the CUSTOM_KOL_EXTENSION.inc file, and 3) place some code into the implementation section from the CUSTOM_CODE_EXTENSION file. inc. You can prepare these files yourself and place them in the project folder. This method is good for extending the functionality of KOL without making changes to the code of the library itself, or at the stage of testing additions,

UNICODE_CTRLS symbol is intended for converting visual objects of TControl into windows that work directly in Unicode encoding. This work has been completed almost completely (although some problems are periodically found, but they are eliminated). It is enough to include this option in the project, and the application will almost completely support UNICODE encodings. All calls to API functions are redirected to UNICODE versions of these functions (with the ending W). Controls like TREEVIEW and LISTVIEW start working with UNICODE versions of window messages. Etc.

The **USE_MHTOOLTIP** symbol allows you to include in the library (and use in the project) tooltips implemented by Dmitry Zharov aka Gandalf. At a minimum, you will need to download the appropriate package and “install” it before adding this option to the project. After that, by assigning the Hint and ShowHint properties to regulate the use of tooltips on controls.

The **USE_OnIdle** symbol includes a call to the ProcessIdle procedure in the message loops, which, if there is any downtime, calls the OnIdle handler that you assigned (by the RegisterIdleHandler procedure).

 © 2024 Carl Peeraer

The **ENUM_DYN_HANDLERS_AFTER_RUN** symbol quite dramatically changes the behavior of the message manager at the end of an application. By default, as soon as the AppletTerminated variable is true, dynamically attached message handlers are no longer invoked from that point on. This is done mainly in order to prevent unnecessary code activity at the time of the application termination. In some cases, for example, the activation of timers that are still not turned off, or some other handlers that respond to changes in the states of window elements (and at the moment of termination of the work, the state of windows begins to actively change, otherwise it cannot be), they could try to address objects that do not exist, or try to get handles of already destroyed windows. All this led to some glitches. To avoid such failures,
If you want your event handlers to continue working until all processes are complete, then add this option to the project, and make sure that your event handlers behave correctly when the application is closed. By the way, enabling this option does not just increase the code, but even slightly reduces it (checking the value of the AppletTerminated variable is disabled).

There are a number of conditional compilation symbols to control the appearance and behavior of buttons, regular (button) and "drawn" (bitbtn).

The **BUTTON_DBLCLICK_AS_CLICK** symbol for all buttons (buttons) of the application changes the functionality in such a way that the event of a double click with the mouse (left key) is no longer recognized as a double click, but in reality leads to two clicks on the button.

The **ALL_BUTTONS_RESPOND_TO_ENTER** symbol provides all kinds of buttons (button and bitbtn) with the ability to respond to the Enter key. The fact is that by default in Windows, buttons respond only to pressing the "space" key, and this is not my invention, this is how the window message handlers of the operating system work. In order for the buttons to be pressed with the Enter key, it is required to add some insignificant code. As KOL strives for leaner application code, this functionality has been made optional. If you want to get it in your application, add this option to your project.

The **ESC_CLOSE_DIALOGS** character adds a response to the Escape key for all dialog forms, ensuring that they are closed.

The **CLICK_DEFAULT_CANCEL_BTN_DIRECTLY** symbol changes the functionality of the default buttons (DefaultBtn property) and cancel buttons (CancelBtn property), namely, pressing these buttons from the keyboard becomes "non-visual". By default, when this symbol is not included in the project options, pressing the corresponding buttons on the keyboard results in visual clicks of buttons on the form, and switching the focus to these buttons on the form.

The **DEFAULT_CANCEL_BTN_EXCLUSIVE** symbol prevents the same button from assigning the CancelBtn and DefaultBtn properties at the same time. Adds some code that, when set to a property, checks for an alternative and disables the opposite property.

The **NO_DEFAULT_BUTTON_BOLD** character disables the special visual appearance of the default button (DefaultBtn), in which it is surrounded by a wider shadow than other buttons on

© 2024 Carl Peeraer

the form. Disabling the special design does not increase, but even slightly reduces the code by a couple of machine instructions.

The **KEY_PREVIEW** symbol provides filtering of button press messages in a form handler for which the KeyPreview property is set (that is, you must both enable this option and set the KeyPreview property for the form to true so that the form can always be the first to process the keys intended for its visual objects).

The **OpenSaveDialog_Extended** symbol significantly expands the functionality of the standard file open and save dialogs (TOpenSaveDialog). This option allows you to use its OSHook and OSTemplate options in the dialog properties, and specify the template name (Template property), for example, to add your own control elements (buttons, checkboxes, labels, etc.) to the dialog window. In addition, using the NoPlaceBar property, it becomes possible to turn off the "standard placements panel" on the left in the dialogs of the new standard. (This may be needed, for example, to speed up the process of opening a dialog, for some reason this panel may slow it down noticeably). If this symbol is not added to the project option, all these options are unavailable (and the placement panel is always present in this case), but the code is somewhat shorter.

The **AUTO_CONTEXT_HELP** symbol provides an automatic response to the WM_CONTEXTMENU message. If the target visual object (TControl) has a nonzero value of the HelpContext property, the application's help system is called for it, passing this context to it. Of course, you must take care of the formation of the help system in the form of a file with the HLP or CHM extension.

The **NOT_FIX_CURINDEX** symbol is for backward compatibility with older KOL projects. In the initial versions of KOL, there was a bug related to visual objects oriented to work with elements (listbox, combobox). This error led to a shift in the value of the CurIndex property in the process of programmatically assigning values to the overlying items (Items), since the assignment is performed by deleting the item and inserting a new value into its position. In the absence of this option in the project properties, this error is now eliminated automatically, but with some increase in the code. In case this error is irrelevant for your application working with visual lists, or if it is fixed in the application itself, you can add this symbol.

The **NOT_FIX_MODAL** symbol returns the situation to the time when KOL applications “did not know how” to activate when clicking on any of their windows at the moment when the modal window is active. By default, KOL applications now respond correctly, activating as expected. But this requires a little extra code. If you do not need this behavior correction for some reason, you can cancel it with this option.

The **NEW_MODAL** symbol is an alternative implementation of the modality proposed by Alexander Pravdin. The implementation of this version of the modality is a little more code-based, and even then mainly because it is not translated into assembler. And it provides more modality organization service for applications. For example, it becomes possible to use the

© 2024 Carl Peeraer

ShowModalParented method, which allows you to show a model form only in relation to a specific form, without affecting other active forms of the application.

The **USE_SETMODALRESULT** symbol somewhat speeds up the application when the form is assigned a new value to the ModalResult property. By default, the value of this property will be analyzed only when processing the next message from the message queue, but the regularity of messages to the window is not guaranteed until nothing happens to it (for example, the mouse does not move and is outside the window, no keys are pressed, no redrawing) required). And the dialog, if its property is assigned a new ModalResult value programmatically, possibly after the completion of any internal operations, may "learn" that it is time to close, with some delay (sometimes a great delay). In the usual case, when the ModalResult change occurs as a response to a key press by the user, there can be no problems, since the button will still have to be pressed out, redrawing will occur, a number of window messages will appear in the queue, and the dialog will react immediately and close quickly. If your situation differs from usual, then use this option: it will provide, in addition to changing the property value, also forced activation of the message reading cycle by sending an empty WM_NULL message to the queue.

The **USE_MENU_CURCTL** symbol allows you to analyze in the event handler that responds to the triggering of the context menu items, which visual object was the "initiator" of the context menu. In fact, the initiator, of course, is usually the user. It is he who presses the right mouse button on any visual element of the form, and enabling this option only ensures that a pointer to the object corresponding to this visual element is entered into the property of the called CurCtl pop-up menu.

The **NEW_MENU_ACCELL** symbol includes alternate code for working with accelerator keys corresponding to menu items without using the system accelerator table. This is one of the few cases where the presence of an option gives a shorter code than its absence. (Probably, one should even enter the opposite symbol OLD_MENU_ACCELL, and make this version of the code the main one).

The **USE_DROPDOWNCOUNT** symbol allows you to change the number of dropdown items displayed in the combo box (by the DropDownCount property). If this option is absent in the project, the number of drop-down elements is entirely determined by the operating system. When this option is present, the default value is set to 8 items, and the DropDownCount property becomes available for editing, which allows you to change this value for each combo box separately.

The **NOT_UNLOAD_RICHEDITLIB** symbol excludes from the KOL module the part of the finalization code that is responsible for unloading the richedXXXX library, if one was loaded. In fact, even if rich edit controls are used in the project, there is no special need to unload this library, and this is done solely for pro forma. The operating system, after the application terminates, will correctly disable all dynamic DLLs in use. The only possible purpose of using this option in a project is to save some code size.

© 2024 Carl Peeraer

The **NOT_USE_RICHEDIT** symbol excludes all references to **richedit** 354 from the KOL module altogether. You can use this option only if your project does not really use rich edit controls. The savings when using this symbol is about 60 bytes of code.

The **RICHEDIT_XPBORDER** symbol adds code to make the border of a rich edit visual appear correctly when using XP themes. With the addition of support for themes in KOL with the **GRAPHCTL_XPSTYLES** symbol, this symbol is automatically included together with **GRAPHCTL_XPSTYLES**, which provides a change of themes and more adequate rendering of visual elements (as well as their transparency) in accordance with the themes of XP / Vista / Windows7.

The **USE_PROP** symbol includes the old version of the code responsible for binding the window to its object. Initially, to provide such binding, API functions GetProp and SetProp were used, which create a named "property" for the window with the identifier 'SELF_'. Later it was decided that for this in most cases it is more convenient and economical to use the GWL_USERDATA field (which is obtained and set by the API functions Get / SetWindowLong). Use this option if, for some reason, you need to use the GWL_USERDATA field, as well as if you use previously written components that call GetProp to get the object associated with the window.

The **PROVIDE_EXITCODE** symbol provides the application exit code set in the WM_QUIT message. If this option is present, to terminate the application with the required exit code, simply executePostQuitMessage (exit_code)... If the option is missing, the exit code will always be 0.

The **INITIALFORMSIZE_FIXMENU** symbol provides an initial form size equal to the form design time set for the MCK project, regardless of whether the form has a main menu bar. In fact, this option ensures that the overall size of the form is saved before the main menu object of the form is created, and the form is restored to this size immediately after the menu is installed on the form. If this is not done, the system keeps the client part of the window unchanged, and for this it increases the total window size.

The **USE_GRAPHCTLS** symbol should be used if your project contains graphics visual objects that do not have their own windows. Prior to version 2.40, it was possible to use graphic controls without any additional character, but this option was introduced, since non-window controls are not used too often, and completely disabling the code associated with their support saves more than a hundred bytes in the final application.

The **GRAPHCTL_XPSTYLES** symbol allows both graphical (non-windowed) visuals and a number of window controls to look almost like XP - Vista - Seven themes when using XP themes. This requires a decent amount of code by KOL standards (**visual_xp_styles.inc module**), and therefore this option should be used only if you really care about the external side of the interface as much as functionality. See also the next option.

© 2024 Carl Peeraer

The **GRAPHCTL_HOTTRACK** symbol aggravates the previous option, allowing graphic visuals not only to statically look "inscribed" in the current XP theme, but also to support the visual effects associated with mouse hovering (as you know, visuals caught under the mouse cursor are slightly "highlighted" slightly changing its appearance). This option includes some more code to achieve the desired effect.

The **ICON_DIFF_WH** symbol provides support for TIcon objects for the Width and Height properties, allowing you to work with non-square thumbnail images (simply "icons"). Initially, KOL only had support for square icons. The functionality to support rectangular icons requires some additional code and is not required as often, which is why it was added as an option.

The **NEW_GRADIENT** symbol includes an alternative fill for the gradient bar suggested by Alexander Karpinsky aka homm. It is faster and smaller in code. Elliptical and rhombic fill is not supported with this option.

The **NEW_ALIGN** symbol enables a new (faster) way to align visuals. Currently, this option is enabled by default, and the OLD_ALIGN symbol should be used to include the old code (in the future, it will probably have to be dropped in order to simplify support).

The **FILE_EXISTS_EX** symbol affects the code of the FileExists function, checking for the existence of the file more carefully. The normal shortcode just takes the file attributes (GetFileAttributes) and checks that the result is received and the attribute is not a directory. In fact, DOS is not dead, and some of the file names in the system have remained reserved for the I / O devices of this prehistoric operating system, for example, PRN. *, CON. *. You can never create such files, neither programmatically nor from explorer. Moreover, these files always "exist", and the GetFileAttributes API function will return attributes that the FileExists function deems acceptable. This option exists just to use an alternative, slightly larger code that searches for the requested file on disk, and as a result, the correct answer was received to the request for the existence of the file, regardless of its phantom. If your application is not going to work with completely arbitrary files (for example, it always receives file names only as a result of executing open and save dialogs, that is, from "trusted" sources), then you will not need this option.

The **NOT_USE_AUTOFREE4CONTROLS** symbol, added in version 2.40, returns the previous behavior when child controls were destroyed in a separate loop in the parent control's destructor. Starting with version 2.40, this functionality is assigned to the general Add2AutoFree method, which uses the code that is always present in the KOL application. Usually, the old visual release mechanism is unnecessary (and will probably be removed from your code).

The **ENDSESSION_HALT** symbol adds code that immediately terminates the process in response to the WM_ENDSESSION message. Moreover, the completion occurs in this case in a rather dangerous way: through a call to Halt. This means that the application will be terminated rather abnormally, without having time to save its states and unsaved data, without performing all the other actions required by the protocol. Conclusion: this option is not recommended for use.

© 2024 Carl Peeraer

The **PSEUDO_THREADS** symbol turns all command threads (NewThreadXXX) into pseudo-threads. Basically, such a transformation can be used for debugging purposes - to increase determinism and make it easier to find errors in a multithreaded application. Another potential use of pseudo-threads is greater programmatic control over the execution priority of threads, but you must implement this functionality yourself.

The **WAIT_SLEEP** symbol, when present with the **PSEUDO_THREADS**45 symbol, adds 10 milliseconds of wait to the loop of its own version of the WaitForMultipleObjects function, in this case replacing the standard API function. The purpose of this addition is to reduce the processor utilization indicator displayed in the task manager (without this symbol, the dispatcher shows 100% processor utilization during the entire waiting cycle, since the cycle is spinning continuously).

And finally, about a number of options for debugging purposes. These options can help you both find problems in your own application and find bugs in the KOL library itself.

The **FILESTREAM_POSITION** symbol makes a copy of the current position in the fData.fPosition field of the file stream object (TStream), although this is not necessary. In fact, when accessing the Position property, the application obtains reliable information about the position in the stream in a different way; duplicating this value in the fPosition field should be equated with a debugging tool for step-by-step debugging. Thus, it becomes possible to find out what the read or write position in the stream is using the variable inspectors (without this option, it is simply impossible to find out this position when stopping at an arbitrary point until some function of the stream working with the position is called, yes and it is rather difficult to do there).

The **DEBUG_GDIOBJECTS** symbol includes code that counts GDI resources (fonts, brushes, pencils). If at the end of the work there are unreleased objects of these types, or after a call to a procedure that should not leave trash after itself, the balance has changed, then you should immediately start the source of the leak.

The **CHK_BITBLT** symbol includes code that analyzes the results of BitBlt operations (in the TBitmap.Draw method), and in case of an error, informs the user about it. I recommend using it only at the debugging stage, especially in cases where you have already noticed any artifacts during drawing.

The **DEBUG_ENDSESSION** symbol is used in conjunction with the ENDSESSION_HALT option to log all window messages after WM_ENDSESSION to the es_debug.txt file in the application folder. Or, this option can be used independently, just provide your own code that sets the EndSession_Initiated variable to true (and it is not at all necessary that this happens in response to the WM_ENDSESSION message).

© 2024 Carl Peeraer

The **DEBUG_CREATEWINDOW** symbol can help you debug window creation problems. If present, the Session.log file records information about requests to create windows.

The **CRASH_DEBUG** symbol is very useful for finding problems with pointer memory misreplorations. Fills the memory occupied by the object when it is freed with hexadecimal DD bytes. If after freeing the object the application will still try to access this memory, then the problem will be detected very quickly, since the object's data is forcibly corrupted after its "death".

DEBUG_OBJKIND symbol adds to objects TControl field fObjKind type PChar... When creating a control, this pointer receives one of the strings that specifies the kind of control (and, possibly, its construction method, for example,'TControl: BitBtn'). This feature can be useful for step-by-step debugging to understand what type of control is processing the message at the moment.

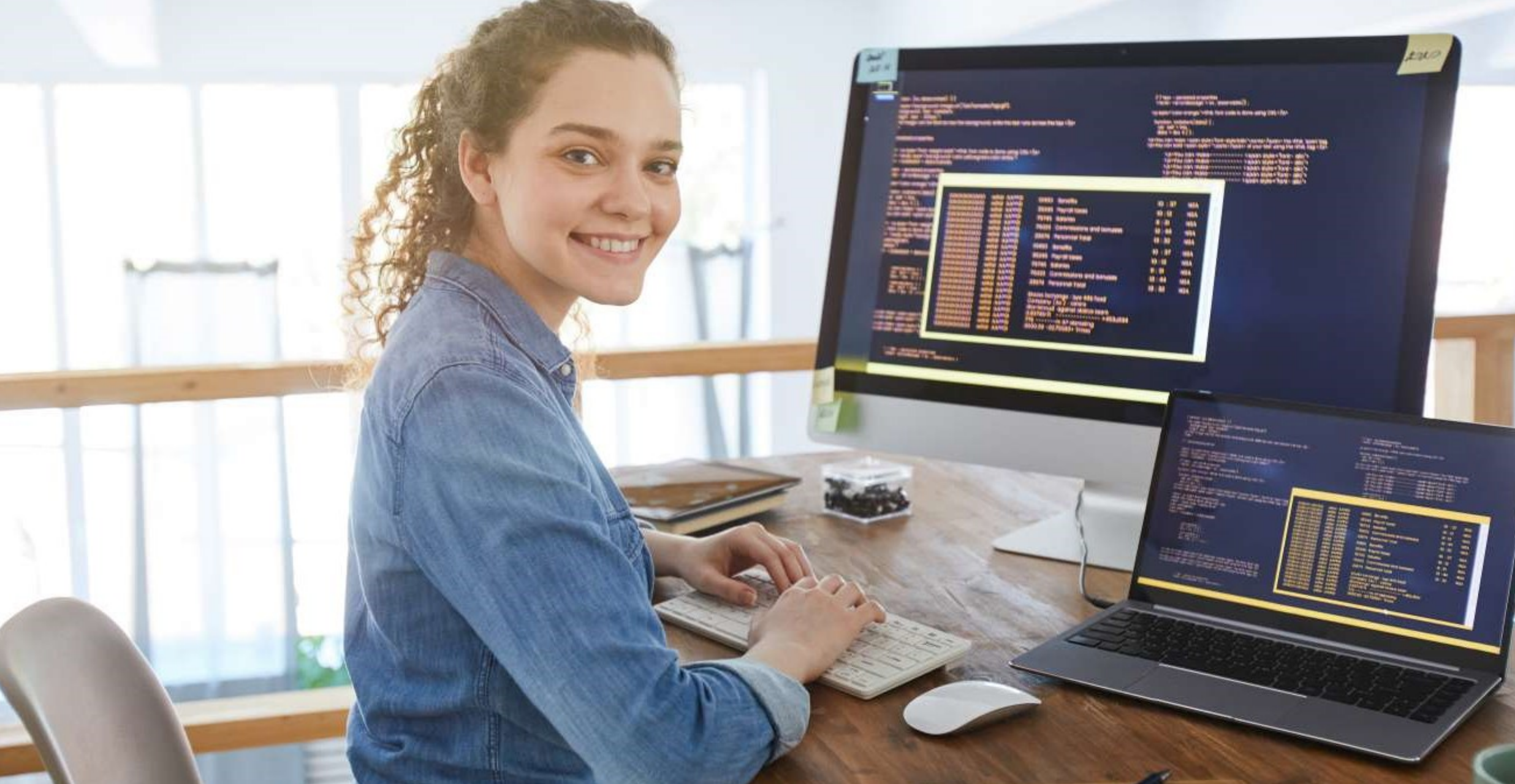
special conditional compilation symbols EXTERNAL_KOLDEFS and EXTERNAL_DEFINES

Since the total number of conditional compilation symbols can easily exceed the size limit, after which the Delphi compiler refuses to accept the remaining symbols, special conditional compilation symbols **EXTERNAL_KOLDEFS** and **EXTERNAL_DEFINES** have been introduced into KOL. If present, the **PROJECT_KOL_DEFS.INC** and **EXTERNAL_DEFINES.INC** files are included at the beginning of KOL.pas, respectively. You provide them, and you place your conditional compilation symbols in it as a set of preprocessor statements {$ DEFINE symbol}. Thus, the limitation on the number of conditional compilation symbols is removed, and it becomes more convenient to manage them than by editing project properties. Do not forget that after changing the composition of symbols, it is better to rebuild the project using the Build command (not Compile).

F_P must be used for **FreePascal** compatibility. It should be added to the list of defined project symbols when compiled by this compiler.

Other options will be described throughout this text in the context of the various KOL components, if required.

© 2024 Carl Peeraer



Programming in KOL

Programming in KOL / MCK

4 Programming in KOL

- String Functions 49
- Working with long integers & Floating Point 59
- Working with Date and Time 62
- Files and Folders 67
- Working with the Registry 78
- System Functions and working with Windows 81
- Messageboxes 86
- Clipboard Operations 88
- Arithmetics, Geometry, Utilities 89
- Sorting Data 91
- Object Type Hierarchy 92
- TList Object (Generic List) 100
- Data Streams in KOL 105
- List of Strings 115
- List of Files and Directories 126
- Tracking Changes on Disk 130
- INI Files 131
- An Array of Bit Flags 135
- Tree in Memory 137
- Elements of Graphics 141
- Image in Memory 156
- Pictogram 170
- List of Images 174
- Before getting started with Visual Objects 183
- Common Properties and Methods - TControl 184
- Programming in KOL (without MCK) 279
- MCK Design 281
- Application graphic resources 287
- Graphics Resources and MCK's 288

© 2024 Carl Peeraer

4.1 String Functions

So let's start with the basics:
The names of the functions for converting strings to numbers and vice versa in KOL, as already mentioned, differ from the names of similar functions in **SysUtils**. Particle "To" in most cases is replaced by a consonant (for English) number 2: not IntToStr, but **Int2Str**, for example. Here is an incomplete list of such functions: **Int2Str (i)**, **Str2Int (i)**, **UInt2Str (i)**, **Int2Hex (i, n)**, **Hex2Int (s)**, **Copy (s, i, n)**, **CopyEnd (s, i)**, **CopyTail (s, n)**, etc ..

Additional conversion functions:
Int2Rome (i) - "writes" a number from 1 to 8999 in Roman numerals;
Int2Ths (i) - the same as **Int2Str**, but the triples of digits are separated from each other by a special separator (by default - a space, but this can be easily changed by assigning your own separator to the global variable ThsSeparator);
Int2Digs (i, n) - the same as **Int2Str**, but the required number of leading spaces is added before the number so that the resulting string is at least n;
Num2Bytes (d) - forms the representation of the number of bytes (specified by the double precision floating point parameter) in the form n or nK or nM or nG or nT - depending on the parameter value;
S2Int (s) - the same as **Str2Int**, but works with a parameter of type PChar, not ANSI string;
cHex2Int (s) - similar to **Hex2Int**, but understands hexadecimal numbers, written according to the rules of the C language (leading 0x characters are discarded);
Octal2Int (s), **Binary2Int (s)** - the purpose of these functions is obvious.
There is also a **Format (s, ...)** function - but in KOL it uses the wvsprintf API, so it doesn't understand floating point formatting.

In addition, KOL has a number of functions to facilitate parsing (parsing) of strings:
Parse (s, d) - returns part of string s up to the first of the encountered characters from string d, leaving only the part after the encountered delimiter in the string s itself.
StrIsStartingFrom (s, p) - Checks that the beginning of string s matches string p.
StrSatisfy (s, p) - checks the string s against a pattern (the pattern can contain the mask characters '*' and '?', Interpreted, respectively, as "any characters" and "one arbitrary character").
SkipSpaces(s) - skips spaces, moving s to the next printable character in the line.
DelimiterLast(s, d) - returns the position of the last separator character from string d in string s;
IncludeTrailingChar (s, c) - returns s, adding character c if it is not already the last in the line;
ExcludeTrailingChar (s, c) - on the contrary, it removes the trailing c.

 © 2024 Carl Peeraer

A number of functions for working with strings in KOL have been moved, sometimes with some changes, from the standard SysUtils module

I do not make any secret from this, the main reason is that in KOL projects it is undesirable to use the SysUtils module itself, due to the increase in the size of the application by 20-30 KB, at the same time these functions are often needed:
StrComp (s1, s2), **StrLComp (s1, s2, n)**, **StrCopy (s1, s2)**, **StrCat (s1, s2)**, **StrLen (s)**, **StrScanLen (s, c, n)**, **StrScan (s, c)**, **StrRScan (s, c)**, **Trim (s)**, **TrimLeft (s)**, **TrimRight (s)**, **LowerCase (s)**, **UpperCase (s)**, **AnsiLowerCase (s)**, **AnsiUpperCase (s)**.
And in addition to them there are: **StrComp_NoCase (s1, s2)**, **StrLComp_NoCase (s1, s2, n)**, **Str2LowerCase (s)** - takes a PChar parameter, and performs string modification in place. And also: **RemoveSpaces (s)**, **WAnsiUpperCase (s)**, **WAnsiLowerCase (s)**.

4.1.1 String Functions - Syntax

```
function Int2Hex( Value: DWord; Digits: Integer ): KOLString;
```
Converts integer Value into string with hex number. Digits parameter determines minimal number of digits (will be completed by adding necessary number of leading zeroes).

```
function Int2Str( Value: Integer ): KOLString;
```
Converts an integer to a string.

```
procedure Int2PChar( s: PAnsiChar; Value: Integer );
```
Converts Value to string and puts it into buffer s. Buffer must have enough size to store the number converted: buffer overflow does not checked anyway!

```
function UInt2Str( Value: DWORD ): AnsiString;
```
The same as Int2Str 50, but for unsigned integer value.

```
function Int2StrEx( Value, MinWidth: Integer ): KOLString;
```
Like Int2Str 50, but resulting string filled with leading spaces to provide at least MinWidth characters.

```
function Int2Rome( Value: Integer ): KOLString;
```
Represents number 1..8999 to Rome number.

```
function Int2Ths( I: Integer ): KOLString;
```
Converts integer into string, separating every three digits from each other by character ThsSeparator. (Convert to thousands).

```
function Int2Digs( Value, Digits: Integer ): KOLString;
```
Converts integer to string, inserting necessary number of leading zeroes to provide desired length of string, given by Digits parameter. If resulting string is greater then Digits, string is not truncated anyway.

© 2024 Carl Peeraer

```
function Num2Bytes( Value: Double ): KOLString;
```
Converts double float to string, considering it as a bytes count. If Value is sufficiently large, number is represented in kilobytes (with following letter K), or in megabytes (M), gigabytes (G) or terabytes (T). Resulting string number is truncated to two decimals (.XX) or to one (.X), if the second is 0.

```
function S2Int( S: PKOLChar ): Integer;
```
Converts null-terminated string to Integer. Scanning stopped when any non-digit character found. Even empty string or string not containing valid integer number silently converted to 0.

```
function Str2Int( const Value: KOLString ): Integer;
```
Converts string to integer. First character, which can not be recognized as a part of number, regards as a separator. Even empty string or string without number silently converted to 0.

```
function Hex2Int( const Value: KOLString ): Integer;
```
Converts hexadecimal number to integer. Scanning is stopped when first non-hexadicimal character is found. Leading dollar ('$') character is skept (if present). Minus ('-') is not concerning as a sign of number and also stops scanning.

```
function cHex2Int( const Value: KOLString ): Integer;
```
As Hex2Int 51, but also checks for leading '0x' and skips it.

```
function Octal2Int( const Value: AnsiString ): Integer;
```
Converts octal number to integer. Scanning is stopped on first non-octal digit (any char except 0..7). There are no checking if there octal numer in the parameter. If the first char is not octal digit, 0 is returned.

```
function Binary2Int( const Value: AnsiString ): Integer;
```
Converts binary number to integer. Like Octal2Int 51, but only digits 0 and 1 are allowed.

```
function ToRadix( number: Radix_int; radix, min_digits: Integer ): KOLString;
```
Converts unsigned number to string representing it literally in a numeric base given by radix parameter.

```
function FromRadixStr( var Rslt: Radix_int; s: PKOLChar; radix: Integer ): PKOLChar;
```
Converts unsigned number from string representation in a numeric base given by a radix parameter. Returns a pointer to a character next to the last digit of the number.

```
function FromRadix( const s: AnsiString; radix: Integer ): Radix_int;
```
Converts unsigned number from string representation in a numeric base given by a radix parameter. See also: FromRadixStr 51 function.

© 2024 Carl Peeraer

`function` **`InsertSeparators`**`( const s: KOLString; chars_between: Integer; Separator: KOLChar ): KOLString;`
Inserts given Separator between symbols in s, separating each portion of chars_between characters with a Separator starting from right side. See also: Int2Ths 50 function.

`function` **`oem2char`**`( const s: AnsiString ): AnsiString;`
Converts string from OEM to ANSI.

`function` **`ansi2oem`**`( const s: AnsiString ): AnsiString;`
Converts ANSI string to OEM.

`function` **`smartOem2ansiRus`**`( const s: AnsiString ): AnsiString;`
Smartly converts string from OEM to ANSI (only Russian!). See code.

`function` **`StrComp`**`( const Str1, Str2: PAnsiChar ): Integer;`
Compares two strings fast. -1: Str1<Str2; 0: Str1=Str2; +1: Str1>Str2

`function` **`StrLComp_NoCase`**`( const Str1, Str2: PAnsiChar; MaxLen: Cardinal ): Integer;`
Compare two strings fast without case sensitivity. Terminating 0 is not considered, so if strings are equal, comparing is continued up to MaxLen bytes. Since this, pass minimum of lengths as MaxLen.

`function` **`StrComp_NoCase`**`( const Str1, Str2: PAnsiChar ): Integer;`
Compares two strings fast without case sensitivity. Returns: -1 when Str1<Str2; 0 when Str1=Str2; +1 when Str1>Str2

`function` **`StrLComp`**`( const Str1, Str2: PAnsiChar; MaxLen: Cardinal ): Integer;`
Compare two strings (fast). Terminating 0 is not considered, so if strings are equal, comparing is continued up to MaxLen bytes. Since this, pass minimum of lengths as MaxLen.

`function` **`StrCopy`**`( Dest, Source: PAnsiChar ): PAnsiChar;`
Copy source string to destination (fast). Pointer to Dest is returned.

`function` **`StrCat`**`( Dest, Source: PAnsiChar ): PAnsiChar;`
Append source string to destination (fast). Pointer to Dest is returned.

`function` **`StrLen`**`( const Str: PAnsiChar ): Cardinal;`
StrLen returns the number of characters in Str, not counting the null terminator.

`function` **`StrScanLen`**`( Str: PAnsiChar; Chr: AnsiChar; Len: Integer ): PAnsiChar;`
Fast scans string Str of length Len searching character Chr. Pointer to a character next to found or to Str[Len] (if no one found) is returned.

© 2024 Carl Peeraer

```
function StrScan( Str: PAnsiChar; Chr: AnsiChar ): PAnsiChar;
```
Fast search of given character in a string. Pointer to found character (or nil) is returned.

```
function StrRScan( Str: PAnsiChar; Chr: AnsiChar ): PAnsiChar;
```
StrRScan returns a pointer to the last occurrence of Chr in Str. If Chr does not occur in Str, StrRScan returns NIL. The null terminator is considered to be part of the string.

```
function StrIsStartingFrom( Str, Pattern: PKOLChar ): Boolean;
```
Returns True, if string Str is starting from Pattern, i.e. if Copy(Str, 1, StrLen 52 (Pattern)) = Pattern. Str must not be nil!

```
function StrIsStartingFromNoCase( Str, Pattern: PAnsiChar ): Boolean;
```
Like StrIsStartingFrom 53 above, but without case sensitivity.

```
function TrimLeft( const S: KOLString ): KOLString;
```
Removes spaces, tabulations and control characters from the starting of string S.

```
function TrimRight( const S: KOLString ): KOLString;
```
Removes spaces, tabulates and other control characters from the end of string S.

```
function Trim( const S: KOLString ): KOLString;
```
Makes TrimLeft 53 and TrimRight 53 for given string.

```
function RemoveSpaces( const S: KOLString ): KOLString;
```
Removes all characters less or equal to ' ' in S and returns it.

```
procedure Str2LowerCase( S: PAnsiChar );
```
Converts null-terminated string to lowercase (inplace).

```
function LowerCase( const S: Ansistring ): Ansistring;
```
Converts Ansistring to lowercase.

```
function UpperCase( const S: Ansistring ): Ansistring;
```
Converts Ansistring to uppercase.

```
function AnsiUpperCase( const S: Ansistring ): Ansistring;
```
Converts Ansistring to uppercase.

```
function AnsiLowerCase( const S: Ansistring ): Ansistring;
```


© 2024 Carl Peeraer

Converts Ansistring to lowercase.

`function` **KOLUpperCase**`( const S: KOLString ): KOLString;`
Converts KOLstring to uppercase.

`function` **KOLLowerCase**`( const S: KOLString ): KOLString;`
Converts KOLstring to lowercase.

`function` **WUpperCase**`( const S: KOLWideString ): KOLWideString;`
Converts KOLwidestring to uppercase.

`function` **WLowerCase**`( const S: KOLWideString ): KOLWideString;`
Converts KOLWidestring to lowercase.

`function` **WAnsiUpperCase**`( const S: KOLWideString ): KOLWideString;`
Converts KOLwideansistring to uppercase.

`function` **WAnsiLowerCase**`( const S: KOLWideString ): KOLWideString;`
Converts KOLwideansistring to lowercase.

`function` **WStrComp**`( const S1, S2: KOLWideString ): Integer;`
Compare two KOLwidestrings (fast). Terminating 0 is not considered, so if strings are equal, comparing is continued up to MaxLen bytes. Since this, pass minimum of lengths as MaxLen.

`function` **_WStrComp**`( S1, S2: PWideChar ): Integer;`

`function` **_WStrLComp**`( S1, S2: PWideChar; Len: Integer ): Integer;`

`function` **WStrScan**`( Str: PWideChar; Chr: WideChar ): PWideChar;`
Fast search of given character in a string. Pointer to found character (or nil) is returned.

`function` **WStrRScan**`( Str: PWideChar; Chr: WideChar ): PWideChar;`
StrRScan 53 returns a pointer to the last occurrence of Chr in Str. If Chr does not occur in Str, StrRScan 53 returns NIL. The null terminator is considered to be part of the string.

`function` **AnsiCompareStr**`( const S1, S2: KOLString ): Integer;`
AnsiCompareStr compares S1 to S2, with case-sensitivity. The compare operation is controlled by the current Windows locale. The return value is the same as for CompareStr.

`function` **_AnsiCompareStr**`( S1, S2: PKOLChar ): Integer;`
The same, but for PChar ANSI strings

`function` **AnsiCompareStrNoCase**`( const S1, S2: KOLString ): Integer;`
AnsiCompareStrNoCase compares S1 to S2, without case-sensitivity. The compare operation is controlled by the current Windows locale. The return value is the same as for CompareStr.

© 2024 Carl Peeraer

`function` **_AnsiCompareStrNoCase**`( S1, S2: PKOLChar ): Integer;`
The same, but for PChar ANSI strings

`function` **AnsiCompareText**`( const S1, S2: KOLString ): Integer;`

`function` **AnsiEq**`( const S1, S2: KOLString ): Boolean;`
Returns True, if AnsiLowerCase 53(S1) = AnsiLowerCase 53(S2). I.e., if ANSI stringsare equal to each other without caring of characters case sensitivity.

`function` **AnsiCompareStrA**`( const S1, S2: AnsiString ): Integer;`
AnsiCompareStr 54 compares S1 to S2, with case-sensitivity. The compare operation is controlled by the current Windows locale. The return value is the same as for CompareStr.

`function` **_AnsiCompareStrA**`( S1, S2: PAnsiChar ): Integer;`
The same, but for PChar ANSI strings.

`function` **AnsiCompareStrNoCaseA**`( const S1, S2: AnsiString ): Integer;`
AnsiCompareStr 54 compares S1 to S2, with case-sensitivity. The compare operation is controlled by the current Windows locale. The return value is the same as for CompareStr.

`function` **_AnsiCompareStrNoCaseA**`( S1, S2: PAnsiChar ): Integer;`
The same, but for PChar ANSI strings.

`function` **AnsiCompareTextA**`( const S1, S2: AnsiString ): Integer;`

`function` **LStrFromPWCharLen**`( Source: PWideChar; Length: Integer ): AnsiString;`
from Delphi5 - because D2 does not contain it.

`function` **LStrFromPWChar**`( Source: PWideChar ): AnsiString;`
from Delphi5 - because D2 does not contain it.

`function` **CopyEnd**`( const S: KOLString; Idx: Integer ): KOLString;`
Returns copy of source string S starting from Idx up to the end of string S. Works correctly for case, when Idx > Length(S) (returns empty string for such case).

`function` **CopyTail**`( const S: KOLString; Len: Integer ): KOLString;`
Returns last Len characters of the source string. If Len > Length(S), entire string S is returned.

`procedure` **DeleteTail**`( var S: KOLString; Len: Integer );`
Deletes last Len characters from string.

`function` **IndexOfChar**`( const S: KOLString; Chr: KOLChar ): Integer;`
Returns index of given character (1..Length(S)), or -1 if a character not found.

`function` **IndexOfCharsMin**`( const S, Chars: KOLString ): Integer;`

© 2024 Carl Peeraer

Returns index (in string S) of those character, what is taking place in Chars string and located nearest to start of S. If no such characters in string S found, -1 is returned.

`function` **`IndexOfWideCharsMin`** `( const S, Chars: KOLWideString ): Integer;`
Returns index (in wide string S) of those wide character, what is taking place in Chars wide string and located nearest to start of S. If no such characters in string S found, -1 is returned.

`function` **`IndexOfStr`** `( const S, Sub: KOLString ): Integer;`
Returns index of given substring in source string S. If found, 1..Length(S)-Length(Sub), if not found, -1.

`function` **`Parse`** `( var S: KOLString; const Separators: KOLString ): KOLString;`
Returns first characters of string S, separated from others by one of characters, taking place in Separators string, assigning a tail of string (after found separator) to source string. If no separator characters found, source string S is returned, and source string itself becomes empty.

`function` **`WParse`** `( var S: KOLWideString; const Separators: KOLWideString ): KOLWideString;`
Returns first wide characters of wide string S, separated from others by one of wide characters, taking place in Separators wide string, assigning a tail of wide string (following found separator) to the source one. If there are no separator characters found, source wide string S is returned, and source wide
string itself becomes empty.

`function` **`ParsePascalString`** `( var S: KOLString; const Separators: KOLString ): KOLString;`
Returns first characters of string S, separated from others by one of characters, taking place in Separators string, assigning a tail of string (after the found separator) to source string. If there are no separator characters found, the source string S is returned, and the source string itself becomes empty. Additionally: if the first (after a blank space) is the quote "'" or '#', pascal string is assumung first and is converted to usual string (without quotas) before analizing of other separators.

`function` **`String2PascalStrExpr`** `( const S: KOLString ): KOLString;`
Converts string to Pascal-like string expression (concatenation of strings with quotas and characters with leading '#').

`function` **`StrEq`** `( const S1, S2: AnsiString ): Boolean;`
Returns True, if LowerCase 53 (S1) = LowerCase 53 (S2). I.e., if strings are equal to each other without caring of characters case sensitivity (ASCII only).

`function` **`WAnsiEq`** `( const S1, S2: KOLWideString ): Boolean;`
Returns True, if AnsiLowerCase 53 (S1) = AnsiLowerCase 53 (S2). I.e., if ANSI stringsare equal to each other without caring of characters case sensitivity.

`function` **`StrIn`** `( const S: AnsiString; const A: array of AnsiString ): Boolean;`

© 2024 Carl Peeraer

Returns True, if S is "equal" to one of strings, taking place in A array. To check equality, StrEq 56 function is used, i.e. comaprison is taking place without case sensitivity.

`function` **WStrIn**`( const S: KOLWideString; const A: array of KOLWideString ): Boolean;`
Returns True, if S is "equal" to one of strings, taking place in A array. To check equality, WAnsiEq 56 function is used, i.e. comaprison is taking place without case sensitivity.

`function` **CharIn**`( C: KOLChar; const A: TSetOfChar ): Boolean;`
To replace expressions like S[1] in ['0'..'z'] to CharIn(S[1], ['0'..'z']) (and to avoid problems with Unicode version of code).

`function` **StrIs**`( const S: AnsiString; const A: Array of AnsiString; var Idx: Integer ): Boolean;`
Returns True, if S is "equal" to one of strings, taking place in A array, and in such Case Idx also is assigned to an index of A element equal to S. To check equality, StrEq 56 function is used, i.e. comaprison is taking place without case sensitivity.

`function` **IntIn**`( Value: Integer; const List: array of Integer ): Boolean;`
Returns TRUE, if Value is found in a List.

`function` **_StrSatisfy**`( S, Mask: PKOLChar ): Boolean;`

`function` **_2StrSatisfy**`( S, Mask: PKOLChar ): Boolean;`

`function` **StrSatisfy**`( const S, Mask: KOLString ): Boolean;`
Returns True, if S is satisfying to a given Mask (which can contain wildcard symbols '*' and '?' interpeted correspondently as 'any set of characters' and 'single any character'. If there are no such wildcard symbols in a Mask, result is True only if S is maching to Mask string.)

`function` **StrReplace**`( var S: KOLString; const From, ReplTo: KOLString ): Boolean;`
Replaces first occurance of From to ReplTo in S, returns True, if pattern From was found and replaced.

`function` **KOLStrReplace**`( var S: KOLString; const From, ReplTo: KOLString ): Boolean;`
Replaces first occurance of From to ReplTo in S, returns True, if pattern From was found and replaced.

`function` **WStrReplace**`( var S: KOLWideString; const From, ReplTo: KOLWideString ): Boolean;`
Replaces first occurance of From to ReplTo in S, returns True, if pattern From was found and replaced. See also function StrReplace 57. This function is not available in Delphi2 (this version of Delphi does not support KOLWideString type).

`function` **StrRepeat**`( const S: KOLString; Count: Integer ): KOLString;`
Repeats given string Count times. E.g., StrRepeat('A', 5) gives 'AAAAA'.

`function` **WStrRepeat**`( const S: KOLWideString; Count: Integer ): KOLWideString;`

 © 2024 Carl Peeraer

Repeats given wide string Count times. E.g., StrRepeat57('A', 5) gives 'AAAAA'.

procedure **NormalizeUnixText**(var S: AnsiString);
In the string S, replaces all occurances of character #10 (without leading #13) to the character #13.

procedure **Koi8ToAnsi**(s: PAnsiChar);
Converts Koi8 text to Ansi (in place)

function **StrPCopy**(Dest: PAnsiChar; const Source: Ansistring): PAnsiChar;
Copy string into null-terminated.

function **StrLCopy**(Dest: PAnsiChar; const Source: PAnsiChar; MaxLen: Cardinal): PAnsiChar;
Copy first MaxLen characters of the Source string into null-terminated Dest.

function **DelimiterLast**(const Str, Delimiters: KOLString): Integer;
Returns index of the last of delimiters given by same named parameter among characters of Str. If there are no delimiters found, length of Str is returned. This function is intended mainly to use in filename parsing functions.

function **__DelimiterLast**(Str, Delimiters: PKOLChar): PKOLChar;
Returns address of the last of delimiters given by Delimiters parameter among characters of Str. If there are no delimeters found, position of the null terminator in Str is returned. This function is intended mainly to use in filename parsing functions.

function **W__DelimiterLast**(Str, Delimiters: PWideChar): PWideChar;

function **SkipSpaces**(P: PKOLChar): PKOLChar;
Skips all characters ' ' in a string.

function **CompareMem**(P1, P2: Pointer; Length: Integer): Boolean;
Fast compare of two memory blocks.

function **AllocMem**(Size: Integer): Pointer;
Allocates global memory and unlocks it.

procedure **DisposeMem**(var Addr: Pointer);
Locks global memory block given by pointer, and frees it. Does nothing, if the pointer is nil.

© 2024 Carl Peeraer

4.2 Working with long integers & Floating Point

I64 vs Int64

Delphi, starting with version 5, introduced the **Int64** data type for working with **8-byte integers**. But earlier versions of Delphi did not have this data type. In order to be able to work with them in older versions of Delphi, KOL introduces its own **I64 data type** and has developed a set of functions for working with this data type:

MakeInt64(lo, hi): I64 - generates a long integer from two ordinary integers;
Int2Int64(i): I64 - "converts" an integer data type to a long integer (equivalent to calling MakeInt64 (i, 0));
IncInt64(I, delta) - increases I: I64 by an integer delta;
DecInt64(I, delta) - decreases I: I64 by delta;
Add64(I1, I2) - adds two numbers like I64;
Sub64(I1, I2) - subtracts I2 from I1;
Neg64(I) - returns -I;
Mul64i(I, i) - multiplies the doubled integer I by the usual integer i;
Div64i(I, i) - divides a doubled whole into an ordinary whole;
Mod64i(I, i) - calculates the remainder of dividing I by i;
Sgn64i(I) - returns the "sign" of the number I (ie -1 if I is negative, 0 if I = 0, or 1 if I> 0);
Cmp64(I1, I2) - compares two doubled integers (also returns -1, 0, 1, depending on whether the first parameter of the second is less, they are equal or the first is greater than the second);
Int64_2Str(I) - converts a doubled integer to a string;
Str2Int64(s) - converts a number in string representation to a doubled integer;
Int64_2Double(I) - converts a doubled integer to a floating point number;
Double2Int64(d) - converts a floating point number to a double integer.

Nobody bothers, however, to use the **Int64** data type built into Delphi of lower versions, but to convert such numbers to a string and back, I still recommend using the **Int64_2Str**, **Str2Int64** functions, performing the appropriate data type conversions.
The use of the other above functions only makes sense if the project is being developed in Delphi 3 or 2.

Floating Point conversions. Floating Point math

In order to avoid the need to include the **SysUtils** module, a set of functions has been introduced in KOL to convert floating point numbers to a string and vice versa. (Normal floating point operations do not require special functions or the connection of the SysUtils module). These are the following functions: **Str2Double (s)**, **Double2Str (d)**, **Str2Extended (s)**, **Extended2Str (e)**.

© 2024 Carl Peeraer

In addition, KOL includes a couple of functions from the section of mathematics that are used in itself, these are **IntPower (i, n)**, and **IsNAN (d)**, as well as the **constant NAN**, which denotes an impossible floating point number (equal to 0/0 uncertainty) ...

Other mathematical functions (trigonometry, logarithms, finding the maximum, minimum number in an array, summation, static and economic functions), similar to the standard ones, are moved to a separate module **kolmath.pas** (when it is turned on, the **err.pas** module is also added to the project, which is used to support exception handling, and increases the weight of the application by about 6KB).

4.2.1 Long Integers & Floating Point - Syntax

```
type I64 = record
// 64 bit integer record. Use it and correspondent functions below in KOL projects to avoid dependancy from Delphi version (earlier versions of Delphi had no Int64 type).
    Lo, Hi: DWORD;
end;
```

```
type PI64 = ^ I64;
```

```
function MakeInt64( Lo, Hi: DWORD ): I64;
```

Generates a long integer from two ordinary integers.

```
function Int2Int64( X: Integer ): I64;
```

"Converts" an integer data type to a long integer (equivalent to calling MakeInt64 (i, 0))

```
procedure IncInt64( var I64: I64; Delta: Integer );
```

Increases I: I64 by an integer delta.
I64 := I64 + Delta;

```
procedure DecInt64( var I64: I64; Delta: Integer );
```

Decreases I: I64 by delta.
I64 := I64 - Delta;

```
function Add64( const X, Y: I64 ): I64;
```

Adds two numbers like I64.
Result := X + Y;

```
function Sub64( const X, Y: I64 ): I64;
```

Subtracts I2 from I1.
Result := X - Y;

© 2024 Carl Peeraer

function **Neg64**(const X: I64 60): I64 60;
Returns -I.
Result := -X;

function **Mul64i**(const X: I64 60; Mul: Integer): I64 60;
Multiplies the doubled integer I by the usual integer i.
Result := X * Mul;

function **Div64i**(const X: I64 60; D: Integer): I64 60;
Divides a doubled whole into an ordinary whole.
Result := X div D;

function **Mod64i**(const X: I64 60; D: Integer): Integer;
Calculates the remainder of dividing I by i.
Result := X mod D;

function **Sgn64**(const X: I64 60): Integer;
Result := sign(X); i.e.:
if X < 0 then -1
if X = 0 then 0
if X > 0 then 1

function **Cmp64**(const X, Y: I64 60): Integer;
Compares two doubled integers (also returns -1, 0, 1, depending on whether the first parameter of the second is less, they are equal or the first is greater than the second.
Result := sign(X - Y); i.e.
if X < Y then -1
if X = Y then 0
if X > Y then 1

function **Int64_2Str**(X: I64 60): AnsiString;
Converts a doubled integer to a string.

function **Int64_2Hex**(X: I64 60; MinDigits: Integer): KOLString;

function **Str2Int64**(const S: AnsiString): I64 60;
Converts a number in string representation to a doubled integer.

function **Int64_2Double**(const X: I64 60): Double;

function **IsNan**(const AValue: Double): Boolean;

© 2024 Carl Peeraer

Checks if an argument passed is NAN.

```
function IsInfinity( const AValue: Double ): Boolean;
```
Checks if an argument passed is Infinite.

```
function IntPower( Base: Extended; Exponent: Integer ): Extended;
```
Result := Base ^ Exponent;

```
function NextPowerOf2( n: DWORD ): DWORD;
```
0->1, 1->1, 2->2, 3->4, 4->4, 5->8, ...

```
function Str2Double( const S: KOLString ): Double;
```

```
function Str2Extended( const S: KOLString ): Extended;
```

```
function Double2Str( D: Double ): KOLString;
```

```
function Extended2Str( E: Extended ): KOLString;
```

```
function Extended2StrDigits( D: Double; n: Integer ): KOLString;
```
Converts floating point number to string, leaving exactly n digits following floating point.

```
function Double2StrEx( D: Double ): KOLString;
```
experimental, do not use

```
function GetBits( N: DWORD; first, last: Byte ): DWord;
```
Retuns bits starting from <first> and to <last> inclusively.

```
function GetBitsL( N: DWORD; from, len: Byte ): DWord;
```
Retuns len bits starting from index <from>.

4.3 Working with Date and Time

The SysUtils standard module from Delphi VCL declares the **TDateTime** data type. In fact, it is equivalent to a double precision floating point number. In its whole part the day is stored, in the fractional part - the time of the day as a fractional part of the day, considering the day as a unit. Similar to the TDateTime datatype in the VCL (the SysUtils module),
KOL introduces its own TDateTime datatype. With the difference that if SysUtils.TDateTime as a floating point number counts in its integer part the days from December 31, 1899, in KOL.TDateTime the countdown starts from January 1, 1 AD (era "from the birth of Christ") - by Gregorian calendar. I did this because I think this data type is convenient not only for communicating with SQL servers, the developers of the standards for which decided that before the 20th century there was nothing that could be counted.

© 2024 Carl Peeraer

If someone needs compatibility with SysUtils.TDatetime, then to convert from a KOL date to a VCL, it is enough to add the **VCLDate0** constant (equal to 693 594, i.e. the number of days from January 1, 1 to December 31, 1899) , and for the reverse transformation, subtract the same constant. For conversion convenience, such a constant is declared in KOL under the name **VCLDate0**.

The set of functions for working with dates and times is slightly different from that of SysUtils:

Now - returns the current system date and time;
Date - returns today's date (discarding the time);
DecodeDate(d, Y, M, D) - decodes the date;
DecodeDateFully(d, Y, M, DW, D) - decodes the date (and day of the week);
DayOfWeek(D) - decodes only the day of the week;
EncodeDate(Y, M, D, T) - encodes date and time;
SystemTime2DateTime(ST, D) - converts the TSystemTime structure to TDateTime;
DateTime2SystemTime(D, ST) - performs the inverse transformation;
Date2StrFmt(s, D) - formats the date into a string;
Time2StrFmt(s, D) - formats the time into a string;
DateTime2StrShort(D) - formats the date into a string using the default short system format;
Str2DateTimeFmt(s1, s2) - reads the date and time from the string according to the specified format;
Str2DateTimeShort(s) - similar to the previous function, but the system default format is used;
Str2DateTimeShortEx(s) - in addition to the previous function, uses separators (depending on the regional settings of the system) so as not to confuse the date with the time.

In addition to working with the TDateTime type, KOL has a number of functions for working with the **TSystemTime** structure directly through the API (floating point numbers are not used in this case):

CompareSystemTime(ST1, ST2) - compares two dates (structures of type TSystemTime), and returns -1, 0, or 1, depending on the result of the comparison;
IncDays(ST, n) - increases the date by the specified number of days (if n <0, then decreases);
IncMonths(ST, n) - increases the date by the specified number of months (for n <0, it decreases);
SystemDate2Str(ST, localeID, dfltDateFmt, s) - formats the date in accordance with the specified parameters;
SystemTime2Str(ST, localeID, flgs, s) - formats the time in accordance with the specified parameters.

4.3.1 Date and Time - Syntax

```
type TDateFormat =( dfShortDate, dfLongDate );
```

Date 64 formats available to use in formatting date/time to string.

© 2024 Carl Peeraer

```
type TTimeFormatFlag =( tffNoMinutes, tffNoSeconds, tffNoMarker, tffForce24 );
```
Additional flags, used for formatting time.

```
type TTimeFormatFlags = Set of TTimeFormatFlag;
```
64
Set of flags, used for formatting time.

```
var MonthDays: array[ Boolean ] of TDayTable =(( 31, 28, 31, 30, 31, 30, 31, 31, 30,
31, 30, 31 ),( 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 ) );
```
The MonthDays array can be used to quickly find the number of days in a month: MonthDays[IsLeapYear65(Y), M].

```
var SecsPerDay = 24* 60* 60;
```
Seconds per day.

```
var MSecsPerDay = SecsPerDay* 1000;
```
Milliseconds per day.

```
var VCLDate0 = 693594;
```
Value to convert VCL "date 0" to KOL "date 0" and back. This value corresponds to 30-Dec-1899, 0:00:00. So, to convert VCL date to KOL date, just subtract this value from VCL date. And to convert back from KOL date to VCL date, add this value to KOL date.

```
function Now: TDateTime;
```
Returns local date and time on running PC.

```
function Date: TDateTime;
```
Returns today local date.

```
procedure DecodeDateFully( DateTime: TDateTime; var Year, Month, Day, DayOfWeek:
WORD );
```
65
Decodes date string and day of the week.

```
procedure DecodeDate( DateTime: TDateTime; var Year, Month, Day: WORD );
```
Decodes date.

```
function EncodeDate( Year, Month, Day: WORD; var DateTime: TDateTime ): Boolean;
```
Encodes date and time.

```
function CompareSystemTime( const D1, D2: TSystemTime ): Integer;
```
Compares to TSystemTime records. Returns -1, 0, or 1 if, correspondantly, D1 < D2, D1 = D2 and D1 > D2.

© 2024 Carl Peeraer

`procedure` **IncDays**`( var SystemTime: TSystemTime; DaysNum: Integer );`
Increases/decreases day in TSystemTime record onto given days count (can be negative).

`procedure` **IncMonths**`( var SystemTime: TSystemTime; MonthsNum: Integer );`
Increases/decreases month number in TSystemTime record onto given months count (can be negative). Correct result is not garantee if day number is incorrect for newly obtained month.

`function` **IsLeapYear**`( Year: Integer ): Boolean;`
Returns True, if given year is "leap" (i.e. has 29 days in the February).

`function` **DayOfWeek**`( Date`64`: TDateTime ): Integer;`
Returns day of week (0..6) for given date.

`function` **SystemTime2DateTime**`( const SystemTime: TSystemTime; var DateTime: TDateTime ): Boolean;`
Converts TSystemTime record to XDateTime variable.

`function` **DateTime2SystemTime**`( const DateTime: TDateTime; var SystemTime: TSystemTime ): Boolean;`
Converts TDateTime variable to TSystemTime record.

`function` **DateTime_System2Local**`( DTSys: TDateTime ): TDateTime;`
Converts DTSys representing system time (+0 Grinvich) to local time.

`function` **DateTime_Local2System**`( DTLoc: TDateTime ): TDateTime;`
Converts DTLoc representing local time to system time (+0 Grinvich)

`function` **FileTime2DateTime**`( const ft: TFileTime; var DT: TDateTime ): Boolean;`

`function` **DateTime2FileTime**`( DT: TDateTime; var ft: TFileTime ): Boolean;`

`procedure` **DivMod**`( Dividend: Integer; Divisor: Word; var Result, Remainder: Word );`
Dividing of integer onto divisor with obtaining both result of division and remainder.

`function` **SystemDate2Str**`( const SystemTime: TSystemTime; const LocaleID: DWORD; const DfltDateFormat: TDateFormat`63`; const DateFormat: PKOLChar ): KOLString;`
Formats date, stored in TSystemTime record into string, using given locale and date/time formatting flags. (E.g.: GetUserDefaultLangID).

`function` **SystemTime2Str**`( const SystemTime: TSystemTime; const LocaleID: DWORD; const Flags: TTimeFormatFlags`64`; const TimeFormat: PKOLChar ): KOLString;`
Formats time, stored in TSystemTime record into string, using given locale and date/time formatting flags.

© 2024 Carl Peeraer

```
function Date2StrFmt( const Fmt: KOLString; D: TDateTime ): KOLString;
```
Represents date as a string correspondently to Fmt formatting string. See possible pictures in definition of the function Str2DateTimeFmt 66 (the first part). If Fmt string is empty, default system date format for short date string used.

```
function Time2StrFmt( const Fmt: KOLString; D: TDateTime ): KOLString;
```
Represents time as a string correspondently to Fmt formatting string. See possible pictures in definition of the function Str2DateTimeFmt 66 (the second part). If Fmt string is empty, default system time format for short date string used.

```
function DateTime2StrShort( D: TDateTime ): KOLString;
```
Formats date and time to string in short date format using current user locale.

```
function Str2DateTimeFmt( const sFmtStr, sS: KOLString ): TDateTime;
```
Restores date or/and time from string correspondently to a format string.

Date 64 and time formatting string can contain following pictures (case sensitive):

	DATE PICTURES
d	Day of the month as digits without leading zeros for single digit days.
dd	Day of the month as digits with leading zeros for single digit days
ddd	Day of the week as a 3-letter abbreviation as specified by a LOCALE_SABBREVDAYNAME value.
dddd	Day of the week as specified by a LOCALE_SDAYNAME value.
M	Month as digits without leading zeros for single digit months.
MM	Month as digits with leading zeros for single digit months
MMM	Month as a three letter abbreviation as specified by a LOCALE_SABBREVMONTHNAME value.
MMMM	Month as specified by a LOCALE_SMONTHNAME value.
y	Year represented only be the last digit.
yy	Year represented only be the last two digits.
yyyy	Year represented by the full 4 digits.
gg	Period/era string as specified by the CAL_SERASTRING value. The gg format picture in a date string is ignored if there is no associated era string. In Enlish locales, usual values are BC or AD.
	TIME PICTURES
h	Hours without leading zeros for single-digit hours (12-hour clock).

© 2024 Carl Peeraer

hh	Hours with leading zeros for single-digit hours (12-hour clock).
H	Hours without leading zeros for single-digit hours (24-hour clock).
HH	Hours with leading zeros for single-digit hours (24-hour clock).
m	Minutes without leading zeros for single-digit minutes.
mm	Minutes with leading zeros for single-digit minutes.
s	Seconds without leading zeros for single-digit seconds.
ss	Seconds with leading zeros for single-digit seconds.
t	One character–time marker string (usually P or A, in English locales).
tt	Multicharacter–time marker string (usually PM or AM, in English locales).
	E.g., 'D, yyyy/MM/dd h:mm:ss'. See also Str2DateTimeShort 67 function.

```
function Str2TimeFmt( const sFmtStr, sS: KOLString ): TDateTime;
```
Same as above but for time only

```
function Str2DateTimeShort( const S: KOLString ): TDateTime;
```
Restores date and time from string correspondently to current user locale.

```
function Str2DateTimeShortEx( const S: KOLString ): TDateTime;
```
Like Str2DateTimeShort 67 above, but uses locale defined date and time separators to avoid recognizing time as a date in some cases.

```
function Str2TimeShort( const S: KOLString ): TDateTime;
```
Like Str2DateTimeShort 67 but for time only.

4.4 Files and Folders

Low Level work with Files and Folders in KOL

Since KOL was developed to create projects primarily for the Windows environment, and the most efficient way to work with files in this environment is to work directly with the corresponding Windows API functions, a number of functions have been created for KOL to work with files at this level. I do not recommend using Pascal functions (Append, Rewrite, Reset, ...), even though they claim to be some kind of platform independence, due to their certain limitations. They also add a few kilobytes of completely useless code to the application. At the same time, working with Windows API functions directly is somewhat inconvenient due to too many parameters that must be specified for almost everything. In my opinion, it is more convenient to work with KOL functions:

FileCreate(s, flags) - creates or opens a file for reading or writing, depending on the flags (for example, **ofOpenRead** or **ofOpenExisting** or **ofShareDenyWrite** - opens an existing file for reading, disallowing writing to this file until the created descriptor is closed. must be concatenated with the or operation):

© 2024 Carl Peeraer

accessor group - only one flag needs to be selected	**ofOpenRead**, **ofOpenWrite**, **ofOpenReadWrite**
group of the way to **create or open** Existing file - only one flag should be selected	**ofOpenExisting**, **ofOpenAlways**, **ofCreateNew**, **ofTruncateExisting**
(optional) **sharing group**	**ofShareDenyWrite**, **ofShareDenyRead**, **ofShareDenyNone**
attribute group	**ofAttrReadOnly**, **ofAttrHidden**, **ofattrSystem**, **ofAttrTemp**, **ofAttrArchive**, **ofAttrOffline**

The result of calling the **FileCreate** function is a descriptor of type **hFile** (just an unsigned 32-bit number) that is used in other file functions to identify an open file object. Using the same descriptor, it is also possible to call API functions for working with files.

FileClose(f) - closes the file;
FileExists(s) - checks for the existence of a file at the specified path;
FileRead(f, buffer, n) - reads bytes from a file into memory;
FileWrite(f, buffer, n) - writes bytes from memory to a file;
FileEOF(f) - checks if the end of the file has been reached (while reading);
FileSeek(f, moveto, movemethod) - moves the read / write pointer in the file;
File2Str(f) - reads the rest of the file as a string.

In addition, there are a number of functions for opening, reading or writing, and closing a file - in one call (and other functions for working with a file by name, without creating a descriptor in the program):

StrSaveToFile(fname, s) - creates or overwrites a file from string s in RAM;
StrLoadFromFile(fname) - reads the entire file into a line;
Mem2File(fname, mem, n) - writes a piece of memory to a file;
File2Mem(fname, mem, n) - reads the entire file into a buffer in memory;
FileTimeCompare(fname1, fname2) - compares the time of the last modification of two files and returns, depending on the results, -1, 0 or 1;
FileSize(fname) - returns the file size (64-bit integer);
ChangeFileExt(fname, ext) - changes the extension of the specified file.

© 2024 Carl Peeraer

Also, there are a number of functions for working with directories (folders), the names of directories and temporary files, and with disks:

GetStartDir - returns the path to the directory in which the application was started (I recommend using this function, not GetWorkDir),
DirectoryExists(s) - checks the existence of a directory;
DirectoryEmpty(s) - checks for the presence of files (and subdirectories) in the specified directory (true is returned if the directory is empty, according to the function name);
DirectoryHasSubdirs(s) - checks for the presence of nested subdirectories in the specified directory;
CheckDirectoryContent(s, subdirsonly, mask) - checks for the presence of files and subdirectories specified by the mask;
CreateDir(s) - creates a directory;
ForceDirectories(s) - creates a directory, ensuring, if necessary, the creation of all overlying directories specified in the path s;
IncludeTrailingPathDelimiter(s)- returns the path s, adds a path separator (character '\') if necessary. In fact, most KOL functions that return a directory path provide a trailing slash ('\'), but sometimes the directory name can be obtained in another way (for example, as a result of manually entering the path by the user in the edit window;
ExcludeTrailingPathDelimiter(s) - in contrast to the previous function, discards the trailing backslash;
FilePathShortened(s, n) - formats the path to the file, shortening it to a maximum of n characters (intermediate directories are discarded from the middle and replaced with ellipsis '...');
FilePathShortenPixels(s, DC, n) - similar to the previous function, but the length of the textual representation of the path "fits" into the size of n pixels on the DC canvas;
ExtractFilePath(s) - extracts only the path to the file directory;
ExtractFileName(s) - extracts the name of the file with the extension;
ExtractFileNameWOExt(s) - extracts file name without extension;
ExtractFileExt(s) - extracts only the extension (the first character in the result string will be '.', except in the case of an empty extension);
GetSystemDir - returns the path to the Windows system directory (Windows \ System32 or another, depending on the Windows version);
GetWindowsDir - returns the path to the directory of Windows itself;
GetWorkDir - returns the path to the "working" directory;
GetTempDir - returns the path to the directory intended for storing temporary files (it is best to create your temporary files in this directory);
CreateTempFile(s, s1) - returns a string that can be used as the name of the temporary file (note that it does not create the file itself);
GetFileListStr(s) - returns a string containing a list (through the symbol with code # 13) of all files in the specified directory;
DeleteFile2Recycle(s) - deletes the specified file (or the list of files listed with delimiter # 13) to the trash (as opposed to the DeleteFile (s) API function);
DeleteFiles(s) - deletes files by mask (as usual, characters '*' and '?' are allowed when specifying a mask template;
CopyMoveFiles(s1, s2, move) - copies or moves the specified (in a list, through the # 13 symbol, template symbols are also allowed) files to the specified directory;

© 2024 Carl Peeraer

DiskFreeSpace(s) - returns the number of free bytes on the disk (result of type I64);
DirectorySize(s) - returns the size of the directory (along with all subdirectories, the result is also of type I64).

In addition to the low-level file access functions, KOL also contains tools for working with streams105, but more on that later (since streams are already objects, and now I'm not touching objects for now).

4.4.1 Files and Folders - Syntax

`var` **`ofOpenRead`** `= O_RDONLY $80000000;`
Use this flag (in combination with others) to open file for "read" only.

`var` **`ofOpenWrite`** `= O_WRONLY $40000000;`
Use this flag (in combination with others) to open file for "write" only.

`var` **`ofOpenReadWrite`** `= O_RDWR $C0000000;`
Use this flag (in combination with others) to open file for "read" and "write".

`var` **`ofShareExclusive`** `= $10 $00;`
Use this flag (in combination with others) to open file for exclusive use.

`var` **`ofShareDenyWrite`** `= $20 $01;`
Use this flag (in combination with others) to open file in share mode, when only attempts to open it in other process for "write" will be impossible. I.e., other processes could open this file simultaneously for read only access.

`var` **`ofShareDenyRead`** `= 0 $02;`
Use this flag (in combination with others) to open file in share mode, when only attempts to open it for "read" in other processes will be disabled. I.e., other processes could open it for "write" only access.

`var` **`ofShareDenyNone`** `= $30 $03;`
Use this flag (in combination with others) to open file in full sharing mode. I.e. any process will be able open this file using the same share flag.

`var` **`ofCreateNew`** `= O_CREAT or O_TRUNC $100;`
Default creation disposition. Use this flag for creating new file (usually for write access.

`var` **`ofCreateAlways`** `= O_CREAT $200;`
Use this flag (in combination with others) to open existing or creating new file. If existing file is opened, it is truncated to size 0.

© 2024 Carl Peeraer

var **ofOpenExisting** = 0 $300;
Use this flag (in combination with others) to open existing file only.

var **ofOpenAlways** = O_CREAT $400;
Use this flag (in combination with others) to open existing or create new (if such file is not yet exists).

var **ofTruncateExisting** = O_TRUNC $500;
Use this flag (in combination with others) to open existing file and truncate it to size 0.

var **ofAttrReadOnly** = 0 $10000;
Use this flag to create Read-Only file (?).

var **ofAttrHidden** = 0 $20000;
Use this flag to create hidden file.

var **ofAttrSystem** = 0 $40000;
Use this flag to create system file.

var **ofAttrTemp** = 0 $1000000;
Use this flag to create temp file.

var **ofAttrArchive** = 0 $200000;
Use this flag to create archive file.

var **ofAttrCompressed** = 0 $8000000;
Use this flag to create compressed file. Has effect only on NTFS, and only if ofAttrCompressed is not specified also.

var **ofAttrOffline** = 0 $10000000;
Use this flag to create offline file.

function **WFileCreate**(const FileName: KOLWideString; OpenFlags: DWord): THandle;

function **FileCreate**(const FileName: KOLString; OpenFlags: DWord): THandle;
Call this function to open existing or create new file. OpenFlags parameter can be a combination of up to three flags (by one from each group:

© 2024 Carl Peeraer

ofOpenRead 70, ofOpenWrite 70, ofOpenReadWrite 70	1st group. Here You decide wish You open file for read, write or read-and-write operations.
ofShareExclusive 70, ofShareDenyWrite 70, ofShareDenyRead 70, ofShareDenyNone 70	2nd group - sharing. Here You can mark out sharing mode, which is used to open file.
ofCreateNew 70, ofCreateAlways 70, ofOpenExisting 71, ofOpenAlways 71, ofTruncateExisting 71	3rd group - creation disposition. Here You determine, either to create new or open existing file and if to truncate existing or not.

```
function FileClose( Handle: THandle ): Boolean;
```
Call it to close opened earlier file.

```
function FileExists( const FileName: KOLString ): Boolean;
```
Returns True, if given file exists.
Note (by Dod): It is not documented in a help for GetFileAttributes, but it seems that under NT-based Windows systems, FALSE is always returned for files opened for excluseve use like pagefile.sys.

```
function WFileExists( const FileName: KOLWideString ): Boolean;
```
Returns True, if given file exists.
Note (by Dod): It is not documented in a help for GetFileAttributes, but it seems that under NT-based Windows systems, FALSE is always returned for files opened for excluseve use like pagefile.sys.

```
function FileSeek( Handle: THandle; const MoveTo: TStrmMove; MoveMethod: TMoveMethod
): TStrmSize;
```
Changes current position in file.

```
function FileRead( Handle: THandle; var Buffer; Count: DWord ): DWord;
```
Reads bytes from current position in file to buffer. Returns number of read bytes.

```
function File2Str( Handle: THandle ): AnsiString;
```
Reads file from current position to the end and returns result as ansi string.

```
function File2WStr( Handle: THandle ): KOLWideString;
```
Reads UNICODE file from current position to the end and returns result as unicode string.

```
function FileWrite( Handle: THandle; const Buffer; Count: DWord ): DWord;
```
Writes bytes from buffer to file from current position, extending its size if needed.

© 2024 Carl Peeraer

function **FileEOF**(Handle: THandle): Boolean;
Returns True, if EOF is achieved during read operations or last byte is overwritten or append made to extend file during last write operation.

function **FileFullPath**(const FileName: KOLString): KOLString;
Returns full path name for given file. Validness of source FileName path is not checked at all.

function **FileShortPath**(const FileName: KOLString): KOLString;
Returns short path to the file or directory.

function **FileIconSystemIdx**(const Path: KOLString): Integer;
Returns index of the index of the system icon correspondent to the file or directory in system icon image list.

function **FileIconSysIdxOffline**(const Path: KOLString): Integer;
The same as FileIconSystemIdx174, but an icon is calculated for the file as it were offline (it is possible to get an icon for file even if it is not existing, on base of its extension only).

function **DirIconSysIdxOffline**(const Path: KOLString): Integer;
The same as FileIconSysIdxOffline174, but for a folder rather then for a file.

procedure **LogFileOutput**(const filepath, str: KOLString);
Debug function. Use it to append given string to the end of the given file.

function **Str2File**(Filename: PKOLChar; Str: PAnsiChar): Boolean;
Save null-terminated string to file directly. If file does not exists, it is created. If it exists, it is overriden. If operation failed, FALSE is returned.

function **WStr2File**(Filename: PKOLChar; Str: PWideChar): Boolean;
Save null-terminated wide string to file directly. If file does not exists, it is created. If it exists, it is overriden. If operation failed, FALSE is returned.

function **StrSaveToFile**(const Filename: KOLString; const Str: AnsiString): Boolean;
Saves a string to a file without any changes. If file does not exists, it is created. If it exists, it is overriden. If operation failed, FALSE is returned.

function **StrLoadFromFile**(const Filename: KOLString): AnsiString;
Reads entire file and returns its content as a string. If operation failed, an empty strinng is returned.
by Sergey Shishmintzev: it is possible to pass Filename = 'CON' to read input from redirected console output.

function **WStrSaveToFile**(const Filename: KOLString; const Str: KOLWideString): Boolean;

© 2024 Carl Peeraer

Saves a string to a file without any changes. If file does not exists, it is created. If it exists, it is overriden. If operation failed, FALSE is returned.

`function` **`WStrLoadFromFile`**`( const Filename: KOLString ): KOLWideString;`
Reads entire file and returns its content as a string. If operation failed, an empty strinng is returned.
by Sergey Shishmintzev: it is possible to pass Filename = 'CON' to read input from redirected console output.

`function` **`Mem2File`**`( Filename: PKOLChar; Mem: Pointer; Len: Integer ): Integer;`
Saves memory block to a file (if file exists it is overriden, created new if not exists).

`function` **`File2Mem`**`( Filename: PKOLChar; Mem: Pointer; MaxLen: Integer ): Integer;`
Loads file content to memory.

`procedure` **`FileTime`**`( const Path: KOLString; CreateTime, LastAccessTime, LastModifyTime: PFileTime ); stdcall;`
Returns file times without opening it.

`function` **`GetUniqueFilename`**`( PathName: KOLString ): KOLString;`
If file given by PathName exists, modifies it to create unique filename in target folder and returns it. Modification is performed by incrementing last number in name (if name part of file does not represent a number, such number is generated and concatenated to it). E.g., if file aaa.aaa is already exist, the function checks names aaa1.aaa, aaa2.aaa, ..., aaa10.aaa, etc. For name abc123.ext, names abc124.ext, abc125.ext, etc. will be checked.

`function` **`FileTimeCompare`**`( const FT1, FT2: TFileTime ): Integer;`
Compares time of file (createing, writing, accessing. Returns -1, 0, 1 if correspondantly FT1<FT2, FT1=FT2, FT1>FT2.

`function` **`DirectoryExists`**`( const Name: KOLString ): Boolean;`
Returns True if given directory (folder) exists.

`function` **`DiskPresent`**`( const DrivePath: KOLString ): Boolean;`
Returns TRUE if the disk is present

`function` **`WDirectoryExists`**`( const Name: KOLWideString ): Boolean;`

`function` **`CheckDirectoryContent`**`( const Name: KOLString; SubDirsOnly: Boolean; const Mask: KOLString ): Boolean;`
Returns TRUE if directory does not contain files (or directories only) satisfying given mask.

`function` **`DirectoryEmpty`**`( const Name: KOLString ): Boolean;`
Returns True if given directory is not exists or empty.

© 2024 Carl Peeraer

`function` **`DirectoryHasSubdirs`**`( const Path: KOLString ): Boolean;`
Returns TRUE if given directory exists and has subdirectories.

`function` **`DirectorySize`**`( const Path: KOLString ): I64 60;`
Returns directory size in bytes as large 64 bit integer.

`function` **`GetStartDir`**`: KOLString;`
Returns path to directory where executable is located (regardless of current directory).

`function` **`ExePath`**`: KOLString;`
Returns the path to the exe-file (in case of dll hook, this is exe-file of the process in which context dll hook function is called).

`function` **`ModulePath`**`: KOLString;`
Returns the path to the module (exe, dll) itself.

`function` **`ExcludeTrailingChar`**`( const S: KOLString; C: KOLChar ): KOLString;`
If S is finished with character C, it is excluded.

`function` **`IncludeTrailingChar`**`( const S: KOLString; C: KOLChar ): KOLString;`
If S is not finished with character C, it is added.

`function` **`IncludeTrailingPathDelimiter`**`( const S: KOLString ): KOLString;`
by Edward Aretino. Adds '\' to the end if it is not present.

`function` **`ExcludeTrailingPathDelimiter`**`( const S: KOLString ): KOLString;`
by Edward Aretino. Removes '\' at the end if it is present.

`function` **`ExtractFileDrive`**`( const Path: KOLString ): KOLString;`
Returns only drive part from exact path to a file or a directory. For network paths, returns a computer name together with a following name of shared directory (like '\\compname\shared\').

`function` **`ExtractFilePath`**`( const Path: KOLString ): KOLString;`
Returns only path part from exact path to file.

`function` **`WExtractFilePath`**`( const Path: KOLWideString ): KOLWideString;`
Returns only path part from exact path to file.

`function` **`IsNetworkPath`**`( const Path: KOLString ): Boolean;`
Returns TRUE, if Path is starting from '\\'.

© 2024 Carl Peeraer

`function` **`ExtractFileName`**`( const Path: KOLString ): KOLString;`
Extracts file name from exact path to file.

`function` **`ExtractFileNameWOext`**`( const Path: KOLString ): KOLString;`
Extracts file name from path to file or from filename.

`function` **`ExtractFileExt`**`( const Path: KOLString ): KOLString;`
Extracts extention from file name (returns it with dot '.' first)

`function` **`ReplaceExt`**`( const Path, NewExt: KOLString ): KOLString;`
Returns Path to a file with extension replaced to a new extension. Pass a new extension started with '.', e.g. '.txt'.

`function` **`ForceDirectories`**`( Dir: KOLString ): Boolean;`
by Edward Aretino. Creates given directory if not present. All needed subdirectories are created if necessary.

`function` **`CreateDir`**`( const Dir: KOLString ): Boolean;`
by Edward Aretino. Creates given directory.

`function` **`ChangeFileExt`**`( FileName: KOLString; const Extension: KOLString ): KOLString;`
by Edward Aretino. Changes file extention.

`function` **`ReplaceFileExt`**`( const Path, NewExt: KOLString ): KOLString;`
Returns a path with extension replaced to a given one.

`function` **`ExtractShortPathName`**`( const Path: KOLString ): KOLString;`

`function` **`FilePathShortened`**`( const Path: KOLString; MaxLen: Integer ): KOLString;`
Returns shortened file path to fit MaxLen characters.

`function` **`FilePathShortenPixels`**`( const Path: KOLString; DC: HDC; MaxPixels: Integer ): KOLString;`
Returns shortened file path to fit MaxPixels for a given DC. If you pass Canvas.Handle of any control or bitmap object, ensure that font is valid for it (or call TCanvas.RequiredState(FontValid) method before. If DC passed = 0, call is equivalent to call FilePathShortened 76, and MaxPixels means in such case maximum number of characters.

`function` **`MinimizeName`**`( const Path: KOLString; DC: HDC; MaxPixels: Integer ): KOLString;`
Exactly the same as MinimizeName in FileCtrl.pas (VCL).

© 2024 Carl Peeraer

`function` **`GetSystemDir`**`: KOLString;`
Returns path to windows system directory.

`function` **`GetWindowsDir`**`: KOLString;`
Returns path to Windows directory.

`function` **`GetWorkDir`**`: KOLString;`
Returns path to application's working directory.

`function` **`GetTempDir`**`: KOLString;`
Returns path to default temp folder (directory to place temporary files).

`function` **`CreateTempFile`**`( const DirPath, Prefix: KOLString ): KOLString;`
Returns path to just created temporary file.

`function` **`GetFileListStr`**`( FPath, FMask: KOLString ): KOLString;`
List of files in string, separating each path from others with a character stored in FileOpSeparator variables (#13 by default). E.g.: 'c:\tmp\unit1.dcu'#13'c:\tmp\unit1.~pa' (for use with DeleteFile2Recycle77())

`function` **`DeleteFiles`**`( const DirPath: KOLString ): Boolean;`
Deletes files by file mask (given with wildcards '*' and '?').

`function` **`DoFileOp`**`( const FromList, ToList: KOLString; FileOp: UINT; Flags: Word; Title: PKOLChar ): Boolean;`
By Unknown Mystic. FileOp can be: FO_MOVE, FO_COPY, FO_DELETE, FO_RENAME. Flags can be a combination of values: FOF_MULTIDESTFILES, FOF_CONFIRMMOUSE, FOF_SILENT, FOF_RENAMEONCOLLISION, FOF_NOCONFIRMATION, FOF_WANTMAPPINGHANDLE, FOF_ALLOWUNDO, FOF_FILESONLY, FOF_SIMPLEPROGRESS, FOF_NOCONFIRMMKDIR, FOF_NOERRORUI. Title used only with FOF_SIMPLEPROGRESS.

`function` **`DeleteFile2Recycle`**`( const Filename: KOLString ): Boolean;`
Deletes file to recycle bin. This operation can be very slow, when called for a single file. To delete group of files at once (fast), pass a list of paths to files to be deleted, separating each path from others with a character stored in FileOpSeparator variable (by default #13, but in case when OLD_COMPAT symbol added - ';'). E.g.: 'unit1.dcu'#13'unit1.~pa'
FALSE is returned only in case when at least one file was not deleted successfully.
Note, that files are deleted not to recycle bin, if wildcards are used or not fully qualified paths to files.

`function` **`CopyMoveFiles`**`( const FromList, ToList: KOLString; Move: Boolean ): Boolean;`

© 2024 Carl Peeraer

4.5 Working with the Registry

In contrast to VCL, where in the Registry.pas module the work with the registry goes through objects, in KOL the main functionality for working with the registry is represented by a number of **functions-adapters to the corresponding API functions**. (If I'm not mistaken, there is also a volunteer-adapted TRegistry for KOL, but I use my own functions, and that's quite enough for me). These low-level functions, like the file access functions, operate on a descriptor like THandle, which is effectively an unsigned number.

A significant difference between the RegKeyXXXXX functions from working directly with the Windows registry API functions is that an incorrect or erroneous access to nonexistent or inaccessible registry keys, even in the absence of checks in the program for the success of the call, leads to idle call skips without any consequences. That is, in case of unsuccessful opening of the key, 0 is returned as a descriptor, and subsequent calls to other functions of this group with such a descriptor are simply ignored (and if it is necessary to return something, default values are returned, i.e. zeros and empty strings).

RegKeyOpenRead(k, s) - opens the registry key for reading;
RegKeyOpenWrite(k, s) - opens the key for writing;
RegKeyOpenCreate(k, s) - creates a key (if it has not been created yet) and opens it for writing;
RegKeyClose(r) - closes an open handle;
RegKeyDelete(r, s) - deletes the subkey with the given name;

RegKeyGetStr(r, s) - returns the value of a string value;
RegKeyGetStrEx(r, s) - the same as the previous function, but additionally understands values of the REG_EXPAND_SZ type (i.e. system variables like% TEMP% are replaced by their values from environment variables);
RegKeySetStr(r, s, s1) - writes a string value;
RegKeySetStrEx(r, s, s1, e) - the same as the previous function, but allows writing values of the REG_EXPAND_SZ type;

RegKeyGetDw(r, s) - returns the value of a numeric value (or a value that can be interpreted as numeric);
RegKeySetDw(r, s, i) - writes a numerical value;

RegKeyDeleteValue(r, s) - deletes the value;
RegKeyExists(r, s) - checks for the presence of a key;
RegKeyValExists(r,s) - checks for the presence of a value;
RegKeyValueSize(r, s) - returns the size of the value;
RegKeyGetBinary(r, s, buf, n) - reads a binary value into the buffer;
RegKeySetBinary(r, s, buf, n) - writes a binary value;
RegKeyGetDateTime(r, s) - reads a date / time value;
RegKeySetDateTime(r, s, d) - writes a value of the date / time type;
RegKeyGetValueTyp(r, s) - returns the type of the value;

© 2024 Carl Peeraer

RegKeyGetValueNames(r, list) - lists the names of all values in the specified list of type PStrList;
RegKeyGetSubKeys(r, list) - lists all subkeys in the specified PStrList.

4.5.1 Registry functions - Syntax

```
function RegKeyOpenRead( Key: HKey; const SubKey: KOLString ): HKey;
```
Opens registry key for read operations (including enumerating of subkeys). Pass either handle of opened earlier key or one of constans HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, HKEY_USERS as a first parameter. If not successful, 0 is returned.

```
function RegKeyOpenWrite( Key: HKey; const SubKey: KOLString ): HKey;
```
Opens registry key for write operations (including adding new values or subkeys), as well as for read operations too. See also RegKeyOpenRead79.

```
function RegKeyOpenCreate( Key: HKey; const SubKey: KOLString ): HKey;
```
Creates and opens key.

```
function RegKeyGetStr( Key: HKey; const ValueName: KOLString ): KOLString;
```
Reads key, which must have type REG_SZ (null-terminated string). If not successful, empty string is returned. This function as well as all other registry manipulation functions, does nothing, if Key passed is 0 (without producing any error).

```
function RegKeyGetStrEx( Key: HKey; const ValueName: KOLString; ExpandEnvVars:
Boolean ): KOLString;
```
Like RegKeyGetStr79, but accepts REG_EXPAND_SZ type, expanding all environment variables in resulting string.
Code provided by neuron, e-mailto:neuron@hollowtube.mine.nu

```
function RegKeyGetDw( Key: HKey; const ValueName: KOLString ): DWORD;
```
Reads key value, which must have type REG_DWORD. If ValueName passed is '' (empty string), unnamed (default) value is reading. If not successful, 0 is returned.

```
function RegKeySetStr( Key: HKey; const ValueName: KOLString; const Value: KOLString
): Boolean;
```
Writes new key value as null-terminated string (type REG_SZ). If not successful, returns False.

```
function RegKeySetStrEx( Key: HKey; const ValueName: KOLString; const Value:
KOLString; expand: Boolean ): Boolean;
```
Writes new key value as REG_SZ or REG_EXPAND_SZ. - by neuron, e-mailto:neuron@hollowtube.mine.nu

```
function RegKeySetDw( Key: HKey; const ValueName: KOLString; Value: DWORD ):
```


© 2024 Carl Peeraer

```
Boolean;
```
Writes new key value as dword (with type REG_DWORD). Returns False, if not successful.

```
procedure RegKeyClose( Key: HKey );
```
Closes key, opened using RegKeyOpenRead 79 or RegKeyOpenWrite 79. (But does nothing, if Key passed is 0).

```
function RegKeyDelete( Key: HKey; const SubKey: KOLString ): Boolean;
```
Deletes key. Does nothing if key passed is 0 (returns FALSE).

```
function RegKeyDeleteValue( Key: HKey; const SubKey: KOLString ): Boolean;
```
Deletes value. - by neuron, e-mailto:neuron@hollowtube.mine.nu

```
function RegKeyExists( Key: HKey; const SubKey: KOLString ): Boolean;
```
Returns TRUE, if given subkey exists under given Key.

```
function RegKeyValExists( Key: HKey; const ValueName: KOLString ): Boolean;
```
Returns TRUE, if given value exists under the Key.

```
function RegKeyValueSize( Key: HKey; const ValueName: KOLString ): Integer;
```
Returns a size of value. This is a size of buffer needed to store registry key value. For string value, size returned is equal to a length of string plus 1 for terminated null character.

```
function RegKeyGetBinary( Key: HKey; const ValueName: KOLString; var Buffer; Count:
Integer ): Integer;
```
Reads binary data from a registry, writing it to the Buffer. It is supposed that size of Buffer provided is at least Count bytes. Returned value is actul count of bytes read from the registry and written to the Buffer.
This function can be used to get data of any type from the registry, not only REG_BINARY.

```
function RegKeySetBinary( Key: HKey; const ValueName: KOLString; const Buffer;
Count: Integer ): Boolean;
```
Stores binary data in the registry.

```
function RegKeyGetDateTime( Key: HKey; const ValueName: KOLString ): TDateTime;
```
Returns datetime variable stored in registry in binary format.

```
function RegKeySetDateTime( Key: HKey; const ValueName: KOLString; DateTime:
TDateTime ): Boolean;
```
Stores DateTime variable in the registry.

© 2024 Carl Peeraer

```
function RegKeyGetSubKeys( const Key: HKEY; List: PKOLStrList ): Boolean;
```
The function enumerates subkeys of the specified open registry key. True is returned, if successful.

```
function RegKeyGetValueNames( const Key: HKEY; List: PKOLStrList ): Boolean;
```
The function enumerates value names of the specified open registry key. True is returned, if successful.

```
function RegKeyGetValueTyp( const Key: HKEY; const ValueName: KOLString ): DWORD;
```
The function receives the type of data stored in the specified value.
If the function fails, the return value is the Key value.
If the function succeeds, the return value return will be one of the following:
REG_BINARY , REG_DWORD, REG_DWORD_LITTLE_ENDIAN, REG_DWORD_BIG_ENDIAN, REG_EXPAND_SZ, REG_LINK , REG_MULTI_SZ, REG_NONE, REG_RESOURCE_LIST, REG_SZ

4.6 Working with Windows

System Functions and Working with Windows

This set of functions rather expands the API than just an adapter, and can be used for a variety of purposes (interaction between windows, including between different applications, identifying any characteristics of the operating system).

GetWindowChild(wnd, kind) - allows you to get a child window of this window with the specified characteristics (owning the keyboard input focus, carriage, seizing the mouse in exclusive use or having an activated menu);
GetFocusedChild(wnd) - a subspecies of the previous function, interested only in windows in the input focus;
FindWindowByThreadID(t) - Finds a window belonging to the specified flow of execution of instructions;

Stroke2Window(wnd, s) - sends a line to the window in focus, as if the user had typed this line on the keyboard;
Stroke2WindowEx(wnd, s, wait) - the same as the previous function, but allows you to "press" including control keys on the keyboard, such as arrows, page turning, etc .;

WindowsShutdown(s, force, reboot) - stops the session / shuts down / reboots the computer;

WinVer - returns the Windows version (the **TWindowsVersion** return type is defined as an ordered list of constants wv31, wv95, wv98, vwME *, wvNT, wvY2K, wvXP, wvVista, wvWin7);
IsWinVer(wv) - checks if the Windows version is one of the specified wv;

ExecuteWait(AppPath, CmdLine, DfltDirectory, Show, TimeOut, ProcID) - launches for execution and waits for completion (specified period of time) the specified application;

© 2024 Carl Peeraer

ExecuteIORedirect(AppPath, CmdLine, DfltDirectory, Show, ProcID, InPipe, OutPipeWr, OutPipeRd) - launches a console application, redirecting its input / output to the specified objects of the pipe type (pipe, literally, a kind of file streams in Windows);
ExecuteConsoleAppIORedirect(const AppPath, CmdLine, DfltDirectory, Show, InStr, OutStr, WaitTimeout) - the same as the previous function, but after the application is launched, the string InStr is "fed" to it, and upon completion, the contents of its console in the OutStr line are read at the output;

GetDesktopRect - returns a rectangle on the screen that is free for application windows (excluding, for example, the Windows taskbar, and other panels at the edges of the screen);
GetWorkArea - the same as the previous function, but the result is obtained in a slightly different way, through SystemParametersInfo. For various purposes, it is more correct to use either this function or the previous one. For Windows 7, the **GetDesktopRect** function is always redirected to the **GetworkArea** function, for example.

Perhaps, in the same section, it is worth adding a couple of functions from **KOL.pas**, which can be used to control the uniqueness of a running instance of an application (it may be necessary that an application does not allow the user to launch itself repeatedly):
JustOne(wnd, s) - returns true if only the application is launched in a single instance (if at the moment of application launch it is found that such is already among those running, false is returned);
JustOneNotify(wnd, s, onanother) - similar to the previous one, but in addition sets an event handler OnAnotherInstance, which is triggered in the first running application, and when the second is launched, when triggered, the event handler receives as a parameter the command line from which the second (and other) instance (s) of the application was launched. For example, if, when making a text editor at the beginning of work, when the main form is still invisible, make a call:

```
    if   not   JustOneNotify   (MainForm.Handle,   'My.Super.Puper.Text.Editor',
OnAnotherMyEditor) then
 MainForm.Close;
```

then when you restart it, the launch will not take place (the application will not even appear on the screen), and the handler in the first instance of the application will receive information about the command line of the second instance, and can load the requested text into a new tab, for example.

4.6.1 Working with Windows - Syntax

```
function ComputerName: KOLString;
```

Returns computer name.

```
function UserName: KOLString;
```

Returns user name (login).

```
function ListUsers: PStrList;
```


© 2024 Carl Peeraer

Returns a list of users currently logined to a system. Don't forget to free it when it is not more necessary!

`function` **`IsUserAdmin`**`( s: KOLString ): TUserRights;`
Returns TRUE if a user (given by s) has administrator rights on a computer.

`type` **`TWindowChildKind`** `=( wcActive, wcFocus, wcCapture, wcMenuOwner, wcMoveSize, wcCaret );`
Type of window child kind. Used in function GetWindowChild 83.

`function` **`GetWindowChild`**`( Wnd: HWnd; Kind: TWindowChildKind` 83 `): HWnd;`
Returns child of given top-level window, having given characteristics. For example, it is possible to get know for foreground window, which of its child window has focus. This function does not work in old Windows 95 (returns Wnd in that case). But for Windows 98, Windows NT/2000 this function works fine. To obtain focused child of the window, use GetFocusedWindow, which is independant from Windows version.

`function` **`GetFocusedChild`**`( Wnd: HWnd ): HWnd;`
Returns focused child of given window (which should be foreground and active, certainly). 0 is returned either if Wnd is not active or Wnd has no focused child window.

`function` **`ForceSetForegroundWindow`**`: Integer;`
Calls AllowSetForegroundWindow (if available) and changes SPI_SETFOREGROUNDLOCKTIMEOUT to 0, returning previus value got by SPI_GETFOREGROUNDLOCKTIMEOUT. If failed, -1 is returned

`var` **`TimeWaitFocus`**`: Byte = 10;`
Delay time while passing keys using Stroke2Window 83 and Stroke2WindowEx 83.

`function` **`Stroke2Window`**`( Wnd: HWnd; const S: AnsiString ): Boolean;`
Posts characters from string S to those child window of Wnd, which has focus now (top-level window Wnd must be foreground, and have focused edit-aware control to receive the stroke). This function allows only to post typeable characters (including such special symbols as #13 (Enter), #9 (Tab), #8 (BackSpace), etc.
See also function Stroke2WindowEx 83, which allows to post any key down and up events, simulating keyboard for given (automated) application.

`function` **`Stroke2WindowEx`**`( Wnd: HWnd; const S: AnsiString; Wait: Boolean ): Boolean;`
In addition to function Stroke2Window 83, this one can send special keys to given window, including functional keys and navigation keys. To post special key to target window, place a combination of names of such key together with keys, which should be passed simultaneously, between square or figure brackets. For example, [Ctrl F1], [Alt Shift Home], [Ctrl E]. For letters and usual characters, it is not necessary to simulate pressing it with determining all Shift combinations and it is sufficient to pass characters as is. (E.g., not '[Shift 1]', but '!').

© 2024 Carl Peeraer

`function SendCommands2Wnd( WndHandle: Hwnd; const s: KOLString ): Boolean;`
Sends commands to a window "as is" (e.g. #13 for Enter). Can pass up to 4K key commands at a time vewry fast.

`function FindWindowByThreadID( ThreadID: DWORD ): HWnd;`
Searches for window, belonging to a given thread.

`function DesktopPixelFormat: TPixelFormat;`
Returns the pixel format correspondent to current desktop color resolution. Use this function to decide which format to use for converting bitmap, planned to draw transparently using TBitmap.DrawTransparent or TBitmap.StretchDrawTransparent methods.

`function ListMonitors: TRectsArray;`
Lists all monitors in system, returns an array of rectangles with its coordinates and sizes.

`function MonitorAt( X, Y: Integer ): TRect;`
Returns monitor where given point (X,Y) is located. If not found, main monitor bounds is returned.

`function GetDesktopRect: TRect;`
Returns rectangle of screen, free of taskbar and other similar app-bars, which reduces size of available desktop when created.

`function GetWorkArea: TRect;`
The same as GetDesktopRect 84, but obtained calling SystemParametersInfo.

`function ExecuteWait( const AppPath, CmdLine, DfltDirectory: KOLString; Show: DWORD; TimeOut: DWORD; ProcID: PDWORD ): Boolean;`
Allows to execute an application and wait when it is finished. Pass INFINITE constant as TimeOut, if You sure that application is finished anyway. If another value passed as a TimeOut (in milliseconds), and application was not finished for that time, ExecuteWait is returning FALSE, and if ProcID is not nil, than ProcID^ contains started process handle (it can be used to wait it more, or to terminate it using TerminateProcess API function).
Launching application can be console or GUI - it does not matter. Pass SW_SHOW, SW_HIDE or other SW_XXX constant as Show parameter as appropriate.
True is returned only in case when application specified was launched successfully and finished for TimeOut specified. Otherwise, check ProcID^ variable: if it is 0, process could not be launched (and it is possible to get information about error using GetLastError API function in a such case). You can freely pass nil in place of ProcID parameter, but this is acually correct only when TimeOut is INFINITE.

`function ExecuteIORedirect( const AppPath, CmdLine, DfltDirectory: KOLString; Show: DWORD; ProcID: PDWORD; InPipe, OutPipeWr, OutPipeRd: PHandle ): Boolean;`

© 2024 Carl Peeraer

Executes an application with its console input and output redirection. Terminating of the application is not waiting, but if ProcID pointer is defined, it receives process Id launched, so it is possible to call WaitForSingleObject for it. InPipe is a pointer to THandle variable which receives a handle to input pipe of the console redirected. The same is for OutPipeWr and OutPipeRd, but for output of the console redirected. Before reading from OutPipeRd^, first close OutPipeWr^. If you run simple console application, for which you want to read results after its termination, you can use ExecuteConsoleAppIORedirect 85 instead.
Notes: if your application is not console and it does not create console using AllocConsole, this function will fail to redirect input-output.

```
function ExecuteConsoleAppIORedirect( const AppPath, CmdLine, DfltDirectory: KOLString; Show: DWORD; const InStr: KOLString; var OutStr: KOLString; WaitTimeout: DWORD ): Boolean;
```

Executes an application, redirecting its console input and output. After redirecting input and output and launching the application, content of InStr is written to input stream of the application, then the application is waiting for its termination (WaitTimeout milliseconds or INFINITE, as passed) and console output of the application is read to OutStr. TRUE is returned only in case, when all these tasks are completed successfully.
Notes: if your application is not console and it does not create console using AllocConsole, this function will fail to redirect input-output.

```
function WindowsShutdown( const Machine: KOLString; Force, Reboot: Boolean ): Boolean;
```

Shut down of Windows NT. Pass Machine = '' to shutdown this PC. Pass Reboot = True to reboot immediatelly after shut down.

```
function WindowsLogoff( Force: Boolean ): Boolean;
```

Logoff of Windows.

```
type TWindowsVersion =( wv31, wv95, wv98, wvME, wvNT, wvY2K, wvXP, wvServer2003, wvVista, wvSeven );
```

Windows versions constants.

```
type TWindowsVersions = Set of TWindowsVersion 85;
```

Set of Windows version (e.g. to define a range of versions supported by the application).

```
function WinVer: TWindowsVersion 85;
```

Returns Windows version.

```
function IsWinVer( Ver: TWindowsVersions 85 ): Boolean;
```

Returns True if Windows version is in given range of values.

```
function ParamStr( Idx: Integer ): KOLString;
```

Returns command-line parameter by index. This function supersides standard ParamStr function.

© 2024 Carl Peeraer

```
function ParamCount: Integer;
```
Returns number of parameters in command line.

```
type TOnAnotherInstance = procedure( const CmdLine: KOLString ) of object;
```
Event type to use in JustOneNotify 86 function.

```
function JustOne( Wnd: PControl; const Identifier: KOLString ): Boolean;
```
Returns True, if this is a first instance. For all other instances (application is already running), False is returned.

```
function JustOneNotify( Wnd: PControl; const Identifier: KOLString; const
aOnAnotherInstance: TOnAnotherInstance ): Boolean;
```
86
Returns True, if this is a first instance. For all other instances (application is already running), False is returned. If handler aOnAnotherInstance passed, it is called (in first instance) every time when another instance of an application is started, receiving command line used to run it.

4.7 Messageboxes

Several simple functions have also been added in KOL to display a simple **message box**. These are:

MsgBox, **MsgOK**, **ShowMsg**, **ShowMsgCentered**, **ShowMessage**, **SysErrorMessage**

In the the additional module **KOLadd.pas**, three additional features have been added: **ShowQuestion**, **ShowQuestionEx**, **ShowMsgModal**.

A feature was also provided to sound a **system beep** at a desired frequency and for a selected time: **SpeakerBeep**

4.7.1 Messageboxes - Syntax

```
function MsgBox( const S: KOLString; Flags: DWORD ): DWORD;
```
Displays message box with the same title as Applet.Caption. If applet is not running, and **Applet** global variable is not assigned, caption 'Error' is displayed (but actually this is not an error - the system does so, if nil is passed as a title).
Returns ID_... result (correspondently to flags passed (MB_OK, MBYESNO, etc. -> ID_OK, ID_YES, ID_NO, etc.)

```
procedure MsgOK( const S: KOLString );
```
Displays message box with the same title as Applet.Caption (or 'Error', if **Applet** is not running).

```
function ShowMsg( const S: KOLString; Flags: DWORD ): DWORD;
```


© 2024 Carl Peeraer

Displays message box like [MsgBox](86), but uses Applet.Handle as a parent (so the message has no button on a task bar).

`function` **ShowMsgCentered**`( Ctl:` **PControl**`; const S: KOLString; Flags: DWORD ): DWORD;`
Displays message box like [ShowMsg](86), but centers it on a control (or form) given by Ctl parameter.

`procedure` **ShowMessage**`( const S: KOLString );`
Like [ShowMsg](86), but has only styles MB_OK and MB_SETFOREGROUND.

`function` **SysErrorMessage**`( ErrorCode: Integer ): KOLString;`
Creates and returns a string containing formatted system error message. It is possible then to display this message or write it to a log file, e.g.:

```
  ShowMsg 86 ( SysErrorMessage( GetLastError ) );
```

`function` **ShowQuestion**`( const S: KOLString; Answers: KOLString ): Integer;`
Modal dialog like ShowMsgModal. It is based on KOL form, so it can be called also out of message loop, e.g. after finishing the application. Also, this function *must* be used in MDI applications in place of any dialog functions, based on MessageBox.
The the second parameter should be empty AnsiString or several possible answers separated by '/', e.g.: 'Yes/No/Cancel'. Result is a number answered, starting from 1. For example, if 'Cancel' was pressed, 3 will be returned.
User can also press ESCAPE key, or close modal dialog. In such case -1 is returned.

`function` **ShowQuestionEx**`( S: KOLString; Answers: KOLString; CallBack: TOnEvent ): Integer;`
Like ShowQuestion, but with CallBack function, called just before showing the dialog.

`procedure` **ShowMsgModal**`( const S: KOLString );`
This message function can be used out of a message loop (e.g., after finishing the application). It is always modal.
Actually, a form with word-wrap label (decorated as borderless edit box with btnFace color) and with OK button is created and shown modal. When a dialog is called from outside message loop, caption 'Information' is always displayed.
Dialog form is automatically resized vertically to fit message text (but until screen height is achieved) and shown always centered on screen. The width is fixed (400 pixels).
Do not use this function outside the message loop for case, when the Applet variable is not used in an application.

`procedure` **SpeakerBeep**`( Freq: Word; Duration: DWORD );`
On Windows NT, calls Windows.Beep. On Windows 9x, produces beep on speaker of desired frequency during given duration time (in milliseconds).

© 2024 Carl Peeraer

4.8 Clipboard Operations

KOL includes some functions for working with text on the **clipboard**. Deze functies zijn: **ClipboardHasText**, **Clipboard2Text**, **Clipboard2WText**, **Text2Clipboard**, **WText2Clipboard**

A **sample program** of working with the **clipboard** in **KOL** can be found here: https://www.artwerp.be/MultiClipboard/setup_multiclipboard.exe

Because the **MultiClipBoard** program is based on code by another author, the reader of this User Guide can register the program for free with the following info:

```
Hello KOL User Guide Reader

Your license for MultiClipBoard is:

###License###77025992687008F673A655D601670CD7KOL User Guide Reader

Copy the license string with CTRL+C to the clipboard to register MultiClipBoa

Best regards
Carl Peeraer - Artwerp.be
```

Further in this manual, clipboard functions are also described with graphics.

4.8.1 Clipboard Operations - Syntax

`function ClipboardHasText: Boolean;`
Returns true, if the clipboard contain text to paste from.

`function Clipboard2Text: AnsiString;`
If clipboard contains text, this function returns it for You.

`function Clipboard2WText: KOLWideString;`
If clipboard contains text, this function returns it for You (as Unicode string).

`function Text2Clipboard( const S: AnsiString ): Boolean;`
Puts given string to a clipboard.

`function WText2Clipboard( const WS: KOLWideString ): Boolean;`
Puts given Unicode string to a clipboard.

© 2024 Carl Peeraer

4.9 Arithmetics, geometry, utilities

Most of these functions are found in the units: **KolMath.pas**, **CplxMath.pas** and **Err.pas**. Check out these units for more information.
You can download KOL + MCK and these units from this link: https://www.artwerp.be/kol/kol-mck-master_3.23.zip.

4.9.1 Arithmetics, geometry, utilities - Syntax

```
function MulDiv( A, B, C: Integer ): Integer;
```
Returns A * B div C. Small and fast.

```
function MakeRect( Left, Top, Right, Bottom: Integer ): TRect; stdcall;
```
Use it instead of VCL Rect function

```
function RectsEqual( const R1, R2: TRect ): Boolean;
```
Returns True if rectangles R1 and R2 have the same bounds

```
function RectsIntersected( const R1, R2: TRect ): Boolean;
```
Returns TRUE if rectangles R1 and R2 have at least one common point. Note, that right and bottom bounds of rectangles are not their part, so, if such points are lying on that bounds, FALSE is returned.

```
function PointInRect( const P: TPoint; const R: TRect ): Boolean;
```
Returns True if point P is located in rectangle R (including left and top bounds but without right and bottom bounds of the rectangle).

```
function OffsetPoint( const T: TPoint; dX, dY: Integer ): TPoint;
```

```
function OffsetSmallPoint( const T: TSmallPoint; dX, dY: SmallInt ): TSmallPoint;
```

```
function Point2SmallPoint( const T: TPoint ): TSmallPoint;
```

```
function SmallPoint2Point( const T: TSmallPoint ): TPoint;
```

```
function MakePoint( X, Y: Integer ): TPoint;
```
Use instead of VCL function Point

```
function MakeSmallPoint( X, Y: Integer ): TSmallPoint;
```
Use to construct TSmallPoint

```
function MakeFlags( FlgSet: PDWORD; FlgArray: array of Integer ): Integer;
```

```
function MakeDateTimeRange( D1, D2: TDateTime ): TDateTimeRange;
```
Returns TDateTimeRange from two TDateTime bounds.

```
procedure Swap( var X, Y: Integer );
```


© 2024 Carl Peeraer

exchanging values

`function **Min**( X, Y: Integer ): Integer;`
minimum of two integers

`function **Max**( X, Y: Integer ): Integer;`
maximum of two integers

`function **Abs**( X: Integer ): Integer;`
absolute value

`function **Sgn**( X: Integer ): Integer;`
sign of X: if X < 0, -1 is returned, if > 0, then +1, otherwise 0.

`function **iSqrt**( X: Integer ): Integer;`
square root

`function **iCbrt**( X: DWORD ): Integer;`
cubic root

© 2024 Carl Peeraer

4.10 Sorting Data

Data sorting: quicksort implementation

This part contains implementation of 'quick sort' algorithm, based on following code:

- **TQSort** by **Mike Junkin** 10/19/95.
- **DoQSort** routine adapted from **Peter Szymiczek's** QSort procedure which was presented in issue#8 of The Unofficial Delphi Newsletter.
- **TQSort** changed by **Vladimir Kladov** (Mr.Bonanzas) to allow 32-bit sorting (of big arrays with more than 64K elements).
- Finally, this sort procedure is adapted to XCL (and then to KOL) requirements (no references to SysUtils, Classes etc. TQSort object is transferred to a single procedure call and DoQSort method is renamed to SortData - which is a regular procedure now).

The most efficient method for performing sorting is the so-called Quick Sort algorithm. The KOL library has an optimized (and assembled) version of this function called **SortData**. (And with version 3.00, the **SortArray** function was added, which provides slightly better performance for arrays and lists of 4-byte values, such as Integer numbers or in-memory string pointers.)

To use the **SortData** function, you need to set 4 parameters: an object for sorting (usually, it is some kind of list or array), the number of elements in the list, as well as a function for comparing two elements and a procedure for exchanging two elements of the array being sorted.

As an example of using the **SortData** and **SortArray** functions, it is recommended to study the implementation of the **SortIntegerArray** and **SortDwordArray** functions, which are also included in the library.

4.10.1 Sorting Data - Syntax

```
procedure SortData( const Data: Pointer; const uNElem: Dword; const CompareFun:
TCompareEvent; const SwapProc: TSwapEvent );
```

Call it to sort any array of data of any kind, passing total number of items in an array and two defined (regular) function and procedure to perform custom compare and swap operations. First procedure parameter is to pass it to callback function CompareFun and procedure SwapProc. Items are enumerated from 0 to uNElem-1.

```
procedure SortArray( const Data: Pointer; const uNElem: Dword; const CompareFun:
TCompareArrayEvent );
```

Like SortData91, but faster and allows to sort only contigous arrays of dwords (or integers or pointers occupying for 4 bytes for each item.

```
procedure SwapListItems( const L: Pointer; const e1, e2: DWORD );
```


© 2024 Carl Peeraer

Use this function as the last parameter for SortData 91 call when a PList object is sorting. SwapListItems just exchanges two items of the list.

```
procedure SortIntegerArray( var A: array of Integer );
```
procedure to sort array of integers.

```
procedure SortDwordArray( var A: array of DWORD );
```
Procedure to sort array of unsigned 32-bit integers.

4.11 Object Type Hierarchy

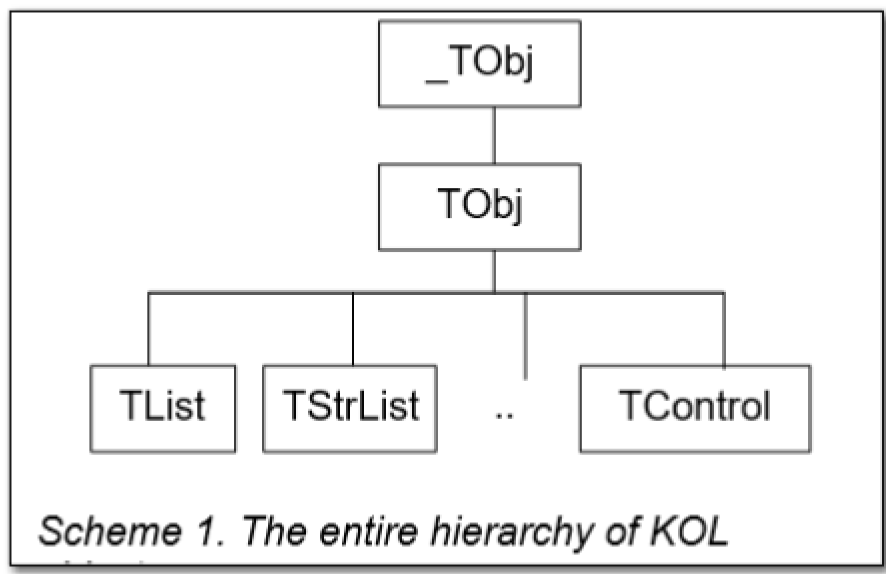
4.11.1 _TObj and TObj objects

At this point, I can end my review of sets of simple functions, and move on to describing the object part of **KOL**. Objects in KOL take advantage of almost all the basic delights of object programming, namely **encapsulation**, **inheritance**, and **polymorphism**, although sometimes somewhat limited.

For example, as I said before, inheritance should not be overused when building an object hierarchy, since each object type (or class) will require its own virtual method table in memory. Therefore, I took great care in building my hierarchy of objects.

The base object type for all objects in **KOL** is **TObj**. For some reason, later (in version 0.93 of 08/25/2001) the **_TObj** object type was introduced, from which **TObj** is inherited. The main reason was that with each modification of the **TObj** type, the pointer to the table of virtual methods vmt was shifted. Creation of a semi-dummy "ancestor" for **TObj** ensured the constancy of the vmt field in the object structure - at offset 0, and made it possible to create the **InstanceSize** function that returns the size of the field structure in memory for any object inherited from **TObj**. In addition, for each call to vmt, the code is shorter by 1 byte in this case. The author of this modification is Vyacheslav Gavrik, for which he is undoubtedly grateful.

© 2024 Carl Peeraer



Scheme 1. The entire hierarchy of KOL

So, what is **TObj** (I will consider its methods together with the methods of its ancestor **_TObj**). To some extent, this is an analogue of the **TObject** class in the **VCL**, and at the same time, it can also be considered an analogue of the **TComponent**. Almost all other object types, with a few exceptions, are derived directly from **TObj**. The **_TObj** object defines the only (first) virtual Init method, and there is one more function **VmtAddr**, and this is where the list of its methods ends (it has no fields of its own). Since **_TObj** is a helper object whose only purpose is to make code smaller, there is no need to directly use it in your program.

The **TObj** object is already more complicated, the virtual destructor **Destroy** already appears in it (but you should always call the Free method), it contains the **fAutoFree** list (of the **PList** type, by the way, the presence of a reference to **PList** in **TObj** already means that at least some methods of the object type **TList** 100 will be included in the code of any KOL program, but I decided to go for it, since it is difficult to do anything without lists at all, i.e. the list will still get into the code of even a minimal program). **fAutoFree** is a list of objects that will be automatically destroyed along with the data, there are methods for adding objects for self-destruction (Add2AutoFree and Add2AutoFreeEx) when the destructor is executed. The **TObj** type (and therefore all object types in KOL) also has an **OnDestroy** event.

There is a Tag field (yes, in KOL it is defined at the lowest level of the hierarchy, i.e. any object in KOL has this field a priori).

There is even an object usage counter, which allows you to prevent the destruction of the object while someone else needs it: calls to **RefInc** increase the counter, and for an object with a nonzero counter, calling the destructor will not lead to any consequences (except for marking that the destructor was called). When decreasing the usage counter by calling RefDec to zero, it is checked whether the destructor was called, and if it was, the object is destroyed, this time finally. The **RefCount** field is available for analysis from the program (the least significant bit of this field is used as an indication that the destructor was called, all the others are a counter that increases by 2 with each call to RefInc, and decreases by 2 with each RefDec).

© 2024 Carl Peeraer

In **KOL** itself, the **RefInc** and **RefDec** methods are applied "just in case" when processing messages for a visual object ("just in case" is that the object can be destroyed while any window message is being processed for it, and then it would be almost inevitable crash of the application). In fact, the **RefInc** methods can be used in **multithreaded applications** to protect temporary objects managed from different threads for a period of active use in some part of the code.

Sometimes it becomes necessary to "simultaneously" destroy an object and zero (assign nil) to the pointer to this object. The **VCL** has a function for this, in **KOL** the function is called **Free_And_Nil** for the same purpose. Moreover, in this function, the object pointer variable is first reset to zero, and only then the object is destroyed (by calling the Free method). Of course, this is almost equivalent to having the object first destroyed and then assigned nil to a pointer variable. But in a multithreaded application, the difference can be felt. It is enough to imagine a situation in which the object was destroyed (or began to decay, but the operation has not been completed yet), and the pointer is still not equal to nil, and at that moment the threads switched, and some operations with the same object through the same pointer. Even in the case of a single-threaded application, the fact that some global pointer continues to point to a non-existent object, or to an object, for which the destruction operation has already begun, presents a certain danger. So the need for the **Free_And_Nil** function is obvious.

In addition to the listed properties, **TObj** has a string field named Name, added by popular demand. But this field is optional, and the compiler knows that such a field exists only when the **Use_Names** 39 conditional compilation symbol is included in the project options. In this case, all named objects are remembered in the list of the parent object, and can be found by calling its FindObj (s) method.

The **TObj** object type is not intended to create its own instances, it was designed precisely as an ancestor for all inherited object types. Non-visual objects should be inherited from it, which must have a destructor, or can be passed as a parameter wherever a variable of type **PObj** is required. In all other cases, when the functionality of the **TObj** object is not required, you can create your own objects that do not derive from **TObj**. But the product of descendants from **TObj** is actually quite inexpensive, so my advice is to inherit all simple objects from **TObj** in general.

4.11.1.1 TObj - Syntax

```
TObj( unit KOL.pas ) ← _TObj
TObj = object( _TObj )
```

Prototype for all objects of KOL. All its methods are important to implement objects in a manner similar to Delphi TObject class.

TObj properties

```
property RefCount: Integer;
```


© 2024 Carl Peeraer

`property` **`Free`**`: Integer;`
Before calling destructor of object, checks if passed pointer is not nil - similar what is done in VCL for TObject. It is ALWAYS recommended to use Free instead of Destroy 95 - see also comments to RefInc 95, RefDec 95.

`property` **`Tag`**`: DWORD;`
Custom data field.

TObj methods

`destructor` **`Destroy`**`; virtual;`
Disposes memory, allocated to an object. Does not release huge strings, dynamic arrays and so on. Such memory should be freeing in overridden destructor.

`procedure` **`Final`**`;`
It is called in destructor to perform OnDestroy 96 event call and to released objects, added to fAutoFree 96 list.

`procedure` **`RefInc`**`;`
See comments below: RefDec 95.

`function` **`RefDec`**`: Integer;`
Decrements reference count. If it is becoming <0, and Free 95 method was already called, object is (self-) destroyed. Otherwise, Free 95 method does not destroy object, but only sets flag "Free 95 was called".
Use RefInc..RefDec to provide a block of code, where object can not be destroyed by call of Free 95 method. This makes code more safe from intersecting flows of processing, where some code want to destroy object, but others suppose that it is yet existing.
If You want to release object at the end of block RefInc..RefDec, do it immediately BEFORE call of last RefDec (to avoid situation, when object is released in result of RefDec, and attempt to destroy it follow leads to AV exception).
Actually, this "function" is a procedure and does not return any sensible value. It is declared as a function for internal needs (to avoid creating separate code for Free 95 method)

`function` **`InstanceSize`**`: Integer;`
Returns a size of object instance.

`constructor` **`Create`**`;`
Constructor. Do not call it. Instead, use New<objectname> function call for certain object, e.g., NewLabel(AParent, 'caption'); 346

© 2024 Carl Peeraer

```
function AncestorOfObject( Obj: Pointer ): Boolean;
```
Is intended to replace 'is' operator, which is not applicable to objects.

```
function VmtAddr: Pointer;
```
Returns addres of virtual methods table of object.

```
procedure Add2AutoFree( Obj: PObj );
```
Adds an object to the list of objects, destroyed automatically when the object is destroyed. Do not add here child controls of the TControl 203 (these are destroyed by another way). Only non-control objects, which are not destroyed automatically, should be added here.

```
procedure Add2AutoFreeEx( Proc: TObjectMethod );
```
Adds an event handler to the list of events, called in destructor. This method is mainly for internal use, and allows to auto-destroy VCL components, located on KOL form at design time (in MCK project).

```
procedure RemoveFromAutoFree( Obj: PObj );
```
Removes an object from auto-free list

```
procedure RemoveFromAutoFreeEx( Proc: TObjectMethod );
```
Removes a procedure from auto-free list

TObj events

```
property OnDestroy: TOnEvent;
```
This event is provided for any KOL object, so You can provide your own OnDestroy event for it.

TObj fields

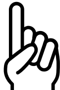
```
fAutoFree: PList;
```
Is called from a constructor to initialize created object instance filling its fields with 0. Can be overridden in descendant objects to add another initialization code there. (Main reason of intending is what constructors can not be virtual in poor objects).

```
fTag: DWORD;
```
Custom data.

4.11.2 Object inheritance from TObj

I'll make a point here: unlike classes, here you have to free all the resources belonging to the object yourself. Namely: all fields that are objects and created for the lifetime of this particular object (call of the Free method). All dynamic

© 2024 Carl Peeraer



chunks of memory allocated by AllocMemory or GetMem (call to the FreeMem function). All dynamic arrays (call SetLength with size parameter equal to 0). All variants (assignment Unassigned). All ANSI strings (empty string assignment). Pay attention to the last point (s): this is the most common source of memory leaks when working with simple objects in KOL projects.

If you inherit your object type directly from the TObj type, then there is no special need to call the inherited method in your implementation of the Init method (this method does nothing in the TObj object itself).

And even if there is a constructing function, it is desirable to transfer that part of the object initialization that does not depend on parameters into the Init method. And remember that when working with simple objects, the word override in your definition of the Init method should not be used: instead, you should use the word virtual again (see example on the right).

If you inherit your object type directly from the TObj type, then there is no special need to call the inherited method in your implementation of the Init method (this method does nothing in the TObj object itself).

And even if there is a constructing function, it is desirable to transfer that part of the object initialization that does not depend on parameters into the Init method. And remember that when working with simple objects, the word override in your definition of the Init method should not be used: instead, you should use the word virtual again (see example on the right).

```
type
 PmyType = ^ TmyType;
 TmyType = object (Tobj)
 Protected
 MyStrProp: string;
 MyStrList: PStrList;
 ...
 procedure Init; virtual;
     destructor     Destroy;
virtual;
 ...

// implementation:
procedure TmyType.Init;
begin
// inherited; - not needed
 MyStrProp: = 'OK';
 MyStrList: = NewStrList;
end;
```

Likewise, for the Destroy destructor: write virtual instead of override. But it is necessary to call inherited, and usually before exiting the destructor, when all its actions to release unnecessary resources have already been completed. Sometimes some actions can be performed after calling the inherited destructor, but in no case should this be a call to the fields of a remote object. Immediately after returning from inherited, the object no longer exists in memory, and an attempt to access them can lead to fatal consequences for the program.

```
destructor TmyType.Destroy;
begin
 MyStrProp: = '';
 MyStrList.Free;
inherited;              //
needednecessarily
end;
```


© 2024 Carl Peeraer

4.11.3 Event Handlers

Any object, except for methods, fields and properties, can also have some "events". An event is (for an object) a field of type a pointer to a function, procedure, or method. (Events outside objects may also exist, then it is just a global variable of the type of a pointer to a procedure, function or method). Most often, events are declared as properties (which allows you to work with them using a consistent syntax, regardless of whether assigning a handler to an event requires a special method call, or a pointer can be assigned as a regular field).

Most events are method pointers, i.e. their type is declared as procedure ... of object or function ... of object. For programmers, this means that this field is not just a pointer that stores the address of a procedure that will be called when an "event" occurs, but contains two pointers (occupying 8 bytes in memory): one points to an instance of an object that handles the event, and the other - on his method.

Handlers for such events should (but not necessarily) not be simple procedures and functions, but methods. For example, the TObj object type already contains an OnDestroy event that fires when the object begins to degrade. When an event is fired, it checks for the presence of an assigned event handler (that is, the nil inequality of a pointer to a procedure), and if there is one, the assigned method is called. The OnDestroy event for TObj objects is of type TOnEvent, declared as follows:

```
type TOnEvent = procedure (Sender: PObj) of object;
```

From the above description of this type of event, it follows that any method declared (in the body of the declaration of some object) as follows is allowed as an OnDestroy handler:
`procedure ObjDestroying (Sender: PObj);` (names are in italics, which can always be replaced with your own). If you try in your code to assign an ordinary procedure (i.e. not a method) to this event as a handler, or a method whose description differs more than using other names instead of ObjDestroying and Sender, the compiler will not compile such code, issuing error message.

Fortunately, the Pascal language, in spite of its seeming strictness, allows to perform the so-called "data type casting". Operation **type_name** (...) tells the compiler that what is written in parentheses is of a data type **type_name**, no matter what data type the expression is being cast. (Of course, you cannot convert any data type to any other in this way, and the main criterion for the possibility of converting one data type to another is that the sizes of the variable before and after the casting must be the same).

Thus, there is a legal opportunity to bypass the requirement that the event handlers are always methods, and not simple procedures and functions. KOL has a special function MakeMethod that allows you to "construct" a method from two pointers - an object pointer (which can be nil, and a simple procedure or function pointer). In order for a method of type procedure of object constructed in this way to be assigned as an event handler, the same OnDestroy, it is enough to cast it to the event type when assigned. For example:

```
MyObj.OnDestroy: = TOnEvent (MakeMethod (nil, @ MyObjDestroying));
```


© 2024 Carl Peeraer

Note that in order to remove the event handler, in any case, it is enough to assign the value nil to the event property - the compiler perfectly understands such an operator as assigning a nil value to both a pointer to a method and a pointer to an object in the event field.

Of course, in this code the compiler will no longer check the correspondence between the procedure type MyObjDestroy and the event type. On the one hand, this is good, as it allows you to compile such code. On the other hand, this is not good at all, since anything can be passed as a procedure pointer. The programmer must now ensure the correct operation of the event handler.

Pay Attention!

But not all programmers know how a simple procedure (or function) differs from a method (this is bad, but it's never too late to learn). The essential difference between a method and simple procedures and functions is that when the method is called, it receives one more parameter. Namely, the pointer of the object itself is passed as the (first, and this is important) invisible parameter, which can be accessed in the method code either explicitly using the reserved name Self, or implicitly, simply by referring to the methods, fields and properties of the object to which it belongs this method.

The conclusion from the above is the following: in order for a simple procedure to be used as an event handler instead of a method, it needs to add the first parameter of type PObj. You can call it whatever is convenient, for example, _Self_, or Dummy (this name is often used to indicate that the parameter is not actually used, and is only needed so that other parameters are passed each in its place).

That is, the following description of the MyObjDestroying procedure will be erroneous:

```
procedure MyObjDestroying (Sender: PObj);
```

while the description would be correct:

```
procedure MyObjDestroying (Dummy: PObj; Sender: PObj);
```

In the first case, when calling the procedure, instead of the Sender parameter, nil, specified when constructing the method as an object, would be passed, and the pointer to the object (Sender) for which the event occurred is lost. Whereas in the second case, it is transmitted correctly. The program, however, is executed, and there is no problem with the violation of the stack pointer, because in Pascal, by default, the first three parameters are passed not through the stack, but through the processor registers. However, if the event handler tries to use Sender, then in the first case it will always "see" the value nil. Looks discouraging, doesn't it?

Let me finish with this educational program, and I hope that if you want to use a simple procedure as an event handler, then you will act correctly.

© 2024 Carl Peeraer

4.12 TList Object (Generic List)

So, the first object that descends from TObj (and is already used in the TObj object itself to store a list of objects and methods for automatic destruction) is TList. It can store arbitrary pointers or 32-bit numbers (which is why it is called a "universal" list).

In Delphi (both VCL and KOL), a list is not just a single or doubly linked list of pointers, but rather an array of pointers. The advantages of an array over a linked list are obvious: great speed of work when you need to quickly access the elements of the list by index. Memory is also consumed more economically: in the case, for example, of a doubly linked list, along with each pointer, you would have to store pointers to the previous and next elements in the list, and allocate your own memory fragment in the heap for this triplet, adding 8 more bytes of overhead per item.

Unfortunately, the array list also has disadvantages when compared to a simple doubly linked list. Namely, since the number of pointers for storage is usually unknown in advance, increasing the size of the array leads to its reallocation in memory, and very often - to moving the entire accumulated array to a new location. With small sizes of lists, this circumstance can be neglected, but if the number of elements reaches several thousand, you need to think about optimizing performance.

For this, there is the Capacity property, which determines how many items in the list will be allocated memory. If the size of the list (the Count property is the current size of the list) exceeds the Capacity value, it is recalculated according to some simple algorithm, which is chosen as a reasonable compromise between saving reserved memory and optimizing program speed by reducing the number of memory reallocations for an array of pointers. By default, the recalculation consists in increasing the reserved size by AddBy, initially equal to 4, but if the AddBy property is set to 0, then the increase occurs immediately by 25%, but not more than 1000. Of course, this algorithm cannot be good for all occasions. ,

The main property of the TList object is Items [i], which provides access to list items by index. For example, a typical loop of enumerating all the elements of a list from first to last is not much different from what is done in the VCL:

```
var L: PList; i: Integer; P: Pointer;
...
for i: = 0 to L. Count-1 do
begin
 P: = L. Items [i];
 ... // working with P
end;
```

Note (once again, it was already mentioned above) that the short form P: = L [i] is not available, because objects object, unlike classes, cannot have default properties (which is a pity, for example, I do not see any point in such a restriction, except for the lack of desire on the part of Dephi developers to provide this convenient service).

© 2024 Carl Peeraer

Another significant difference from the VCL is how the TList object type is instantiated. If in VCL we wrote:

```
var L: TList;
...
L: = TList.Create;
```

then in KOL you should write differently:

```
var L: PList;
...
L: = NewList;
```

Now I will give the main set of methods and properties of TList, in addition to those already mentioned:

Add(P) - adds a pointer to the end of the list;
Insert(i, P) - inserts a pointer at position i (all previous elements starting from i, if any, are shifted one position up);
Delete(i) - removes one element with index i (all elements with index i + 1, if any, are shifted one position down);
DeleteRange(i, n) - fast deletion of n elements from position i (it is allowed to specify as n a greater value than there are elements starting from position i, i.e. DeleteRange (i, MaxInt) - will delete all elements starting from index i);
Remove(P) - finds and removes the first occurrence of the pointer P;
Clear - clears the list, removing all pointers from it;
IndexOf(P) - finds the first occurrence of the pointer P and returns its index (or -1 if there is no such pointer in the list);
Last - returns the last pointer in the list, equivalent to Items [Count-1];
Swap(i1, i2) - swaps pointers with indices i1 and i2;
MoveItem(i1, i2) - removes an element from position with index i1 and inserts it into position with index i2;
Release - can be used to destroy the list of pointers to memory areas allocated in the heap (by the GetMem or AllocMem, ReallocMem functions), beforehand for all non-null pointers, FreeMem is called;
ReleaseObjects - similar to the previous procedure, but used for a list of pointers to objects: all objects in the list are destroyed by calling the Free method;
AddItems(a) - allows you to add an array of pointers at once;
Assign(L) - assigns to the list the elements of another list, i.e. simply copies an array of pointers;
DataMemory - returns the current pointer to the internal array of pointers that make up the list (you should use it with caution, and only if you need to significantly optimize the speed of access to the elements of the list).

Comment: *KOL also has an object type **TListEx** (but it is moved to the additional module **KOLadd.pas**, for reasons of saving the number of lines in the main module **KOL.pas**, and because there is no strict need to use it). In addition to the properties and methods of **TList**, it has a property **Objects [i]**, methods **AddObject (P, o)**, **InsertObject (i, P, o)** and others - allowing*

© 2024 Carl Peeraer

*you to associate another "object" -number with each element of the list, or pointer. In fact, there is no special need for such an object, and it is often enough to use the same **TList**, storing pairs of values in it and assuming that each even element, together with the next odd one, form an inseparable bundle.*

4.12.1 Speeding up work with large Lists

Speeding up work with large lists and lists of strings. Conditional compilation symbol TLIST_FAST.

In addition, with the conditional compilation symbol **TLIST_FAST**, the structure of the list and the algorithms for working with elements are changed in such a way as to provide a faster operation of adding new elements. At the same time, the additional **UseBlocks property** becomes available, which allows you to control the use of new list methods. The increase in speed occurs primarily due to the fact that the number of memory reallocation operations (with copying accumulated pointers to a new location in memory) decreases. More precisely, the reallocation of memory is no longer required at all, only new blocks of 256 elements per block are allocated as needed.

Unfortunately, the increase in performance at the stage of adding elements turns into a decrease in the speed of accessing the elements of the list. In the case when all the elements are used in the blocks, except for the last one, the reduction in the access speed is insignificant: the block index is calculated by the index (by simple division by 256), and after obtaining the pointer to the block from the continuous list of blocks, the required element can be retrieved. The worst result is obtained if, as a result of deleting or inserting elements somewhere in the middle or at the beginning of the list, incomplete blocks are formed. In this case, a longer algorithm is used to find the desired block and the index of the desired element in it, which requires enumeration of all blocks from the initial to the required block. It is somewhat optimized for the case of sequential access to elements,

To speed up the work with the "quick" list at the stage when its initial filling is completed, and then it is used only for accessing elements, it is required to ensure its "dense" filling, in which all blocks are completely filled in it (except, perhaps, the last). To do this, just call the method OptimizeForReadcompaction of blocks.

Note that in the case when the **TLIST_FAST** symbol is defined in the project options, it also affects the string lists. **TStrList**, **TStrListEx**, **TWStrList**, **TWStrListEx**... In order to speed up their work after the initial filling, you should call their method OptimizeForReadwhich refers to the corresponding method of the inner list.

4.12.2 TList Object - Syntax

```
TList( unit KOL.pas ) ← TObj ← _TObj
TList = object( TObj )
```


© 2024 Carl Peeraer

Simple list of pointers. It is used in KOL instead of standard VCL TList to store any kind data (or pointers to these ones). Can be created calling function NewList.

TList properties

`property **Count**: Integer;`
Returns count of items in the list. It is possible to delete a number of items at the end of the list, keeping only first Count items alive, assigning new value to Count property (less then Count it is).

`property **Capacity**: Integer;`
Returns number of pointers which could be stored in the list without reallocating of memory. It is possible change this value for optimize usage of the list (for minimize number of reallocating memory operations).

`property **Items**[ Idx: Integer ]: Pointer; default;`
Provides access (read and write) to items of the list. Please note, that TList is not responsible for freeing memory, referenced by stored pointers.

`property **AddBy**: Integer;`
Value to increment capacity when new items are added or inserted and capacity need to be increased.

`property **DataMemory**: PPointerList;`
Raw data memory. Can be used for direct access to items of a list. Do not use it for **TLIST_FAST** 38 !

TList methods

`destructor **Destroy**; virtual;`
Destroys list, freeing memory, allocated for pointers. Programmer is resposible for destroying of data, referenced by the pointers.

`procedure **Clear**;`
Makes Count 103 equal to 0. Not responsible for freeing (or destroying) data, referenced by released pointers.

`procedure **Add**( Value: Pointer );`
Adds pointer to the end of list, increasing Count 103 by one.

`procedure **Insert**( Idx: Integer; Value: Pointer );`
Inserts pointer before given item. Returns Idx, i.e. index of inserted item in the list. Indeces of items, located after insertion point, are increasing. To add item to the end of list, pass Count 103

© 2024 Carl Peeraer

as index parameter. To insert item before first item, pass 0 there.

`function **IndexOf**( Value: Pointer ): Integer;`
Searches first (from start) item pointer with given value and returns its index (zero-based) if found. If not found, returns -1.

`procedure **Delete**( Idx: Integer );`
Deletes given (by index) pointer item from the list, shifting all follow item indeces up by one.

`procedure **DeleteRange**( Idx, Len: Integer );`
Deletes Len items starting from Idx.

`procedure **Remove**( Value: Pointer );`
Removes first entry of a Value in the list.

`function **Last**: Pointer;`
Returns the last item (or nil, if the list is empty).

`procedure **Swap**( Idx1, Idx2: Integer );`
Swaps two items in list directly (fast, but without testing of index bounds).

`procedure **MoveItem**( OldIdx, NewIdx: Integer );`
Moves item to new position. Pass NewIdx >= Count 103 to move item after the last one.

`procedure **Release**;`
Especially for lists of pointers to dynamically allocated memory. Releases all pointed memory blocks and destroys object itself.

`procedure **ReleaseObjects**;`
Especially for a list of objects derived from TObj. Calls **Free** for every of the object in the list, and then calls **Free** for the object itself.

`procedure **Assign**( SrcList: PList );`
Copies all source list items.

`procedure **AddItems**( const AItems: array of Pointer );`
Adds a list of items given by a dynamic array.

`function **ItemAddress**( Idx: Integer ): Pointer;`

© 2024 Carl Peeraer

Returns an address of memory occupying by the item with index Idx. (If the item is a pointer, returned value is a pointer to a pointer). Item with index requested must exist.

4.13 Data Streams in KOL

I have already described working with files in KOL, at a low level 67. The set of functions for working with files does not require the use of objects. Working with data stream objects provides a higher level both for working with files and with any data sets, for example, in memory. Without the use of objects, it would be quite difficult to provide an acceptable level of encapsulation for this functionality, so the TStream object type is introduced in KOL, in much the same way as in the VCL. Just like in the VCL, it has methods for reading (Read) and writing (Write) data, to change the current position in the stream (Seek).

But that's where the similarities end right there. Instead of inheriting the required data stream classes from the base TStream class, KOL uses a mechanism of function pointers. In the "constructors" of instances of data streams (that is, in the NewXXXXXStream functions) these pointers are assigned certain sets of functions, as a result objects of the same object type TStream are obtained (constructors, of course, return pointers of created streams, of type PStream) , but these objects provide different functionality based on which constructor is called.

So, the following "constructors" of data streams are defined in the KOL module itself:
NewReadFileStream(s) - creates a stream for reading a file (an existing file is opened in "read-only" mode);
NewWriteFileStream(s) - creates a stream for writing a file (a new file is created, or, if it already exists, the file is opened for writing);
NewReadWriteFileStream(s) - a stream is created for writing and reading a file;
NewFileStream(s, options) - allows you to create a file stream with a more detailed listing of the opening and creation modes (these are the same options that are used in the FileCreate function);

NewMemoryStream - creates a stream in memory (for writing and reading);
NewExMemoryStream(P, n)- also creates a thread in memory, but this time in existing memory. If in the previous "constructor" a stream was created that initially does not contain data and grows as it is written to it by methods like Write, then this function creates a stream on an existing contiguous piece of memory (with address P and length n bytes), and the size of this stream does not may change while working with a stream. This memory is not considered to be "owned" by such a thread, and when the data stream object is destroyed, it is not freed in any way (to free it, if, for example, it was dynamically allocated, the code or object that allocated it should).
The benefits of creating this kind of flow are obvious. Let's say you already have some structured data in memory, and there is a method that can read this data from the stream. Instead of creating a regular stream in memory (**NewMemoryStream**), writing this data to it, and then reading it, we simply create a stream in existing memory (NewExMemoryStream), and immediately read the data using the available method. At the same time, at least the allocation

© 2024 Carl Peeraer

of memory for a new stream and copying of this data are saved, which in the case of a large data size also has a very positive effect on the performance of the application.
NewMemBlkStream(blksize) and **NewMemBlkStream_WriteOnly (blksize)** - These two constructors allow you to create a stream of data in memory, but continuity is guaranteed only for the chunk of data written by a single call to the Write method. What is important is that it is guaranteed that the recorded data is not transferable in the future. This type of data flow is very convenient to use to improve the efficiency of the memory manager, providing a single large block of data allocation at once. That is, memory is allocated less frequently, but in large portions (and subsequently freed up faster). Usually, it makes sense to use this kind of stream in write-only mode, getting the address of the next written memory block through the fJustWrittenBlockAddress field. In KOL itself, such a stream is used by the TDirList object type to improve performance.
NewExFileStream(hFile) - similar to the previous one, creates a stream for reading or writing to a file, but based on the existing descriptor of an already open file. Note that the descriptor can also refer to an object of type pipe (pipe), and there is no other way to create a stream for working with a pipe.

In addition to this set of "constructors" of streams, it is possible to create your own types of data streams based on **TStream**. (For example, the DIUCL package defines the stream constructors **NewUCLCStream** and **NewUCLDStream**, which compress and decompress data in a way that works with streams.)

The set of methods on the **TStream** object in KOL provides everything you need to read and write data. When working with KOL data streams, unlike VCL, you need to remember that all methods and properties, including those that are not typical for this type of data flow, remain open for use. But, for example, it makes no sense to try to write to a read-only file stream, or the Handle property has no meaning for the in-memory data stream (Handle provides access to the file descriptor, but only matters for file streams). In VCL, additional control is provided by the compiler at the stage of writing the code; in KOL, you need a little more care, but it achieves a more compact size of the application, with the same functionality.

Here is a list of the **main methods** and properties of **TStream**:

Read(buf, n) - reads a maximum of n bytes from the current position in the stream into the buffer, returns the number of bytes read (it may be less if the end of the data has been reached);
Write(buf, n) - writes n bytes from the buffer in memory to the stream;
Seek(n, method) - moves a position in the stream, returns a new position;
Position - current position in the stream;
Size - stream size (for some stream types the stream size may not be known);
Memory - a pointer to the memory in which the stream data is located in memory (for other types of streams it is always nil);
Capacity - memory reserve for streams in memory (as well as for TList, it can be changed externally in order to optimize the speed of memory allocation);
Handle - file stream descriptor (you can analyze it for the inequality of the constant INVALID_HANDLE_VALUE immediately after opening to make sure that the connection with the

© 2024 Carl Peeraer

file is established normally, for example, or use other low-level functions for working with files that allow an open file descriptor as a parameter, but with a certain caution);
SaveToFile(s) - saves all the contents of the stream to a file named s.

This set is extended with **additional methods** for working with **strings in a stream**:

WriteStr(s) - writes the specified string to the stream (neither the terminating byte with the code # 0, nor the length of the string is written, it is assumed that the "reader" of the stream will subsequently know this length: either it is written in a different way to the same stream, or it is constant, or is calculated somehow);
WriteStrZ(s) - writes a string and a terminating null byte to the stream;
ReadStrZ - reads a null-terminated string from the stream;
ReadStr - reads a string from the stream, ending with one of the combination of characters: # 0, # 13 # 10, # 13, # 10;
ReadStrLen(n) - reads a string of length n bytes from the stream;
WriteStrEx(s) - writes to the stream first the length of the string (4 bytes), and then the string itself - without the terminating null byte;
ReadStrEx - reads from the stream first the length of the string, then the string itself (the inverse of the previous write function);
ReadStrExVar(s) - the same as the previous method, but reads a string into the s parameter, and returns the number of bytes read;
WriteStrPas(s) - writes a short string (such strings up to 255 bytes in length were used in the first versions of the Pascal language, if you remember, the size of such a string is stored in the 0th byte of the string), while the length of the string is written first (1 byte);
ReadStrPas - reads a Pascal string (first read the byte storing the length of the Pascal string, from 0 to 255, then the string itself).

And one more set of methods is used to work with **threads in asynchronous mode**, when the program, having issued a request for a read or write operation, can continue without stopping to wait for the operation to complete, and then, when the result of the operation is already definitely needed by the program, the Wait method is called to completion of the current operation:

SeekAsync(n, method) - the same as Seek, but asynchronously;
ReadAsync(buf, n) - the same as Read (the essential difference is that, since the operation has just begun, but not yet completed, this procedure cannot return the number of bytes read, therefore it is framed as a procedure);
WriteAsync(buf, n) - the same as Write, but asynchronously;
Busy - returns true if the thread has not yet completed the operation;
Wait - permanently waiting for the completion of the last asynchronous operation.

© 2024 Carl Peeraer

Quite often it is required to **transfer a portion of data from one data stream to another**, for this there are global functions:

Stream2Stream(dst, src, n)- reads n bytes from the src stream (source - source) and writes them to the dst stream. In the case when one of the streams (or both) is a stream in memory, it performs optimization and does not create an intermediate buffer up to n bytes in size, but uses the memory in the stream in memory as a buffer;
Stream2StreamEx(dst, src, n) - the same as above, but does not optimize for streams in memory, but easily copes with very large data streams (since it sends data in portions through a 64 Kbyte buffer);
Stream2StreamExBufSz(dst, src, n, bufsz)- the same as the previous function, but allows you to set your own size of the intermediate buffer for data transfer. It is likely that allocating a 1 MB buffer will significantly speed up the transfer of large amounts of data, but at the same time allocating an even larger portion of memory for the buffer can only reduce performance if there is not enough memory in the system.

In the case when the **resources in the application** contain some kind of data that is easy to read through a stream, the following global function comes in handy:

Resource2Stream(dst, inst, s, restype) - allows you to read a resource of any restype type into the stream (not only from the application module, but also from any executable file for which the inst descriptor is obtained).

Among other things, the **TStream** type has Methods and Data properties for developers of new flavors of data streams. To create a new kind of data stream, you need to define your own "constructor", and in this constructor you specify your set of methods (using the Methods property) for reading, writing and changing the position in the stream. These methods can use the Data structure to place their service data (the usual set should be enough, but, as a last resort, it is always possible to allocate an additional block of memory and use one of the fields of this structure to refer to its structure).

4.13.1 Data Streams - Syntax

```
type TStream = object( TObj 92 )
```

Simple stream object. Can be opened for file, or as memory stream (see NewReadFileStream 109, NewWriteFileStream 109, NewMemoryStream 109, etc.). And, another type of streaming object can be derived (without inheriting new object type, just by writing another New...Stream method, which calls _NewStream 108 and pass methods record to it).

```
function _NewStream( const StreamMethods: TStreamMethods ): PStream;
```

Use this method only to define your own stream type. See also declared below (in KOL.pas) methods used to implement standard KOL streams. You can use it in your code to create streams, which are partially based on standard methods.

© 2024 Carl Peeraer

function **NewFileStream**(const FileName: KOLString; Options: DWORD): PStream;
Creates file stream for read and write. Exact set of open attributes should be passed through Options parameter (see **FileCreate** where those flags are listed).

function **NewFileStreamWithEvent**(const FileName: KOLString; Options: DWORD): PStream;
Creates file stream for read and write. Exact set of open attributes should be passed through Options parameter (see **FileCreate** where those flags are listed). Also, resulting stream is supporting OnChangePos event.

function **NewReadFileStream**(const FileName: KOLString): PStream;
Creates file stream for read only.

function **NewReadFileStreamWithEvent**(const FileName: KOLString): PStream;
Creates file stream for read only, supporting OnChangePos event.

function **NewWriteFileStream**(const FileName: KOLString): PStream;
Creates file stream for write only. Truncating of file (if needed) is provided automatically.

function **NewWriteFileStreamWithEvent**(const FileName: KOLString): PStream;
Creates file stream for write only. Truncating of file (if needed) is provided automatically. Created stream supports OnChangePos event.

function **NewReadWriteFileStream**(const FileName: KOLString): PStream;
Creates stream for read and write file. To truncate file, if it is necessary, change Size property.

function **NewReadFileStreamW**(const FileName: KOLWideString): PStream;
Creates file stream for read only.

function **NewWriteFileStreamW**(const FileName: KOLWideString): PStream;
Creates file stream for write only. Truncating of file (if needed) is provided automatically.

function **NewReadWriteFileStreamW**(const FileName: KOLWideString): PStream;
Creates stream for read and write file. To truncate file, if it is necessary, change Size property.

function **NewExFileStream**(F: HFile): PStream;
Creates read only stream to read from opened file or pipe from the current position. When stream is destroyed, file handle still not closed (your code should do this) and file position is not changed (after the last read operation).

function **NewMemoryStream**: PStream;
Creates memory stream (read and write).

function **NewMemoryStreamWithEvent**: PStream;

© 2024 Carl Peeraer

Creates memory stream (read and write). Created stream support OnChangePos event.

`function NewExMemoryStream( ExistingMem: Pointer; Size: DWORD ): PStream;`

Creates memory stream on base of existing memory. It is not possible to write out of top bound given by Size (i.e. memory can not be resized, or reallocated. When stream object is destroyed this memory is not freed.

`function NewMemBlkStream( BlkSize: Integer ): PStream;`

Creates memory stream which consists from blocks of given size. Contrary to a memory stream, contents of the blocks stream should not be accessed directly via fMemory but therefore it is possible to access its parts by portions written to blocks still those were written contigously. To do so, get an address of just written portion for further usage via field fJustWrittenBlkAddress. It is guarantee that blocks of memory allocated during write process never are relocated until destruction the stream.

`function NewMemBlkStream_WriteOnly( BlkSize: Integer ): PStream;`

Same as NewMemoryStream 109

`function NewConcatStream( Stream1, Stream2: PStream ): PStream;`

Creates a stream which is a concatenation of two source stream. After the call, both source streams are belonging to the resulting stream and these will be destroyed together with the resulting stream. (So forget about it). After the call, first stream will not be changed in size via methods of concatenated stream (and it is not recommended to use further Stream1 and Stream2 methods too). But Stream2 can still be increased, if it allows doing so when some data are appended or Size of resulting stream is changed (but not less then Stream1.Size). Nature and physical location of Stream1 and Stream2 are not important and can be absolutely different. But it is supposed that both streams are not compressed and its Size is known always and Seek operation is valid. This function accepts recursive (multi-level) usage: resulting concatenation stream can be used as a left or right parameter to create another concatenation stream later, so it is possible to build a tree of streams concatenated, concatenating this way several different streams and use it as a single data streaming object.

`function NewSubStream( BaseStream: PStream; const FromPos, Size: TStrmSize ): PStream;`

Creates a stream which is a subpart of BaseStream passes, starting from FromPos and with given Size. Like in function NewConcatStream 110, passes BaseStream become owned by newly created sub-stream object, and will be destroyed automatically together with a sub-stream. If you want to provide more long life time for a base stream (e.g. if you plan to use it after a sub-stream based on it is destroyed), use method RefInc for base stream once to prevent it from destroying when the sub-stream is destroyed. Note: be careful and avoid direct calling methods and properties of the base stream, while you have a sub-stream created on base it, since the sub-stream actually redirects all the requests to the parent base stream. Sub-stream accepts setting Size to greater value later, and if some data are written to it, it is written actually to the base stream, and when it is written beyond the end position, this will increase size of the base stream too (and if it is a file stream, this also will increase size of the file on which the base

© 2024 Carl Peeraer

stream was created). This function accepts recursive (multi-level) usage: it is possible to create later another sub-stream on base of existing sub-stream, still it is actully can be treated as usual stream.

```
function Stream2Stream( Dst, Src: PStream; const Count: TStrmSize ): TStrmSize;
```
Copies Count (or less, if the rest of Src is not sufficiently long) bytes from Src to Dst, but with optimizing in cases, when Src or/and Dst are memory streams (intermediate buffer is not allocated).

```
function Stream2StreamEx( Dst, Src: PStream; const Count: TStrmSize ): TStrmSize;
```
Copies Count bytes from Src to Dst, but without any optimization. Unlike Stream2Stream 111 function, it can be applied to very large streams. See also Stream2StreamExBufSz 111.

```
function Stream2StreamExBufSz( Dst, Src: PStream; const Count: TStrmSize; BufSz:
DWORD ): TStrmSize;
```
Copies Count bytes from Src to Dst using buffer of given size, but without other optimizations. Unlike Stream2Stream 111 function, it can be applied to very large streams

```
function Resource2Stream( DestStrm: PStream; Inst: HInst; ResName: PKOLChar;
ResType: PKOLChar ): Integer;
```
Loads given resource to DestStrm. Useful for non-standard resources to load it into memory (use memory stream for such purpose). Use one of following resource types to pass as ResType:

RT_ACCELERATOR	Accelerator table
RT_ANICURSOR	Animated cursor
RT_ANIICON	Animated icon
RT_BITMAP	Bitmap resource
RT_CURSOR	Hardware-dependent cursor resource
RT_DIALOG	Dialog box
RT_FONT	Font resource
RT_FONTDIR	Font directory resource
RT_GROUP_CURSOR	Hardware-independent cursor resource
RT_GROUP_ICON	Hardware-independent icon resource
RT_ICON	Hardware-dependent icon resource

© 2024 Carl Peeraer

RT_MENU	Menu resource
RT_MESSAGETABLE	Message-table entry
RT_RCDATA	Application-defined resource (raw data)
RT_STRING	String-table entry
RT_VERSION	Version resource

For example:

```
var MemStrm: PStream;
    JpgObj: PJpeg;
......
 MemStrm := NewMemoryStream;
 JpgObj := NewJpeg;
......
 Resource2Stream( MemStrm, hInstance, 'MYJPEG', RT_RCDATA );
 MemStrm.Position := 0;
 JpgObj.LoadFromStream( MemStrm );
 MemStrm.Free;
......
```

TStream properties

`property **Size**: TStrmSize;`
Returns stream size. For some custom streams, can be slow operation, or even always return undefined value (-1 recommended).

`property **Position**: TStrmSize;`
Current position

`property **Memory**: Pointer;`
Only for memory stream.

`property **Handle**: THandle;`
Only for file stream. It is possible to check that Handle <> INVALID_HANDLE_VALUE to ensure that file stream is created OK.

`property **Methods**: PStreamMethods;`
Pointer to TStreamMethods record. Useful to implement custom-defined streams, which can access its fCustom field, or even to change methods when necessary.

`property **Data**: TStreamData;`

© 2024 Carl Peeraer

Pointer to TStreamData record. Useful to implement custom-defined streams, which can access Data fields directly when implemented.

`property **Capacity**: TStrmSize;`
Amound of memory allocated for data (MemoryStream).
Properties, inherited from TObj 92

TStream methods

`function **Read**( var Buffer; const Count: TStrmSize ): TStrmSize;`
Reads Count bytes from a stream. Returns number of bytes read.

`function **Seek**( const MoveTo: TStrmMove; MoveMethod: TMoveMethod ): TStrmSize;`
Allows to change current position or to obtain it. Property Position 112 uses this method both for get and set position.

`function **Write**( var Buffer; const Count: TStrmSize ): TStrmSize;`
Writes Count bytes from Buffer, starting from current position in a stream. Returns how much bytes are written.

`function **WriteVal**( Value: DWORD; Count: DWORD ): DWORD;`
Writes maximum 4 bytes of Value to a stream. Allows writing constants easier than via Write 113.

`function **WriteStr**( S: AnsiString ): DWORD;`
Writes string to the stream, not including ending #0. Exactly Length(S) characters are written.

`function **WriteStrZ**( S: AnsiString ): DWORD;`
Writes string, adding #0. Number of bytes written is returned.

`function **WriteWStrZ**( S: KOLWideString ): DWORD;`
Writes string, adding #0. Number of bytes written is returned.

`function **ReadStrZ**: AnsiString;`
Reads string, finished by #0. After reading, current position in the stream is set to the byte, follows #0.

`function **ReadWStrZ**: KOLWideString;`
Reads string, finished by #0. After reading, current position in the stream is set to the byte, follows #0.

`function **ReadStr**: AnsiString;`
Reads string, finished by #13, #10 or #13#10 symbols. Terminating symbols #13 and/or #10 are

© 2024 Carl Peeraer

not added to the end of returned string though stream positioned follow it.

```
function ReadStrLen( Len: Integer ): AnsiString;
```
Reads string of the given length Len.

```
function WriteStrEx( S: AnsiString ): DWord;
```
Writes string S to stream, also saving its size for future use by ReadStrEx* functions. Returns number of actually written characters.

```
function ReadStrExVar( var S: AnsiString ): DWord;
```
Reads string from stream and assigns it to S. Returns number of actually read characters. Note: String must be written by using WriteStrEx 114 function. Return value is count of characters READ, not the length of string.

```
function ReadStrEx: AnsiString;
```
Reads string from stream and returns it.

```
function WriteStrPas( S: AnsiString ): DWORD;
```
Writes a string in Pascal short string format - 1 byte length, then string itself without trailing #0 char. S parameter length should not exceed 255 chars, rest chars are truncated while writing. Total amount of bytes written is returned.

```
function ReadStrPas: AnsiString;
```
Reads 1 byte from a stream, then treat it as a length of following string which is read and returned. A purpose of this function is reading strings written using WriteStrPas 114.

```
procedure SeekAsync( MoveTo: TStrmMove; MoveMethod: TMoveMethod );
```
Changes current position asynchronously. To wait for finishing the operation, use method Wait 115.

```
procedure ReadAsync( var Buffer; Count: DWord );
```
Reads Count bytes from a stream asynchronously. To wait finishing the operation, use method Wait 115.

```
procedure WriteAsync( var Buffer; Count: DWord );
```
Writes Count bytes from Buffer, starting from current position in a stream - asynchronously. To wait finishing the operation, use method Wait 115.

```
function Busy: Boolean;
```
Returns TRUE until finishing the last asynchronous operation started by calling SeekAsync 114, ReadAsync 114, WriteAsync 114 methods.

© 2024 Carl Peeraer

```
procedure Wait;
```
Waits for finishing the last asynchronous operation.

```
procedure SaveToFile( const Filename: KOLString; const Start, CountSave: TStrmSize );
```
Methods, inherited from TObj 92

TStream events

```
property OnChangePos: TOnEvent;
```
To allow using this event, create stream with special constructing function like **NewMemoryStreamWithEvent** or **NewReadFileStreamWithEvent**, or replace reading / writing methods to certain supporting **OnChangePos** event.

4.14 List of Strings

Lists of strings in KOL (TStrList, TStrListEx and others)

Of course, string lists are a very handy sort of object for storing randomly sized strings. They are also present in KOL, and are called **TStrList** and **TStrListEx**. But in KOL, these lists are not used to virtualize access to strings in Memo or RichEdit. An important difference from **TStrings** in VCL: strings cannot contain the # 0 character, since strings are stored exactly as character strings, terminated by a byte with code # 0. This is dictated by considerations of speed of work with large texts. Loading text (for example, from a file) into a **TStrList** object, or saving text to a file or stream is extremely fast and is instantaneous, even for megabytes and tens of megabytes.

The object **TStrListEx** differs from its **TStrList** ancestor (one of the rare cases in KOL when the object type is not directly inherited from TObj) in that it has an Objects property that maps each string to a 32-bit number, or a pointer (in fact, **TStrListEx** is built as a union of **TStrList** and **TList**, and operations on their elements are performed synchronously).

The "constructor" functions are used to create lists of strings:
NewStrList - creates an object of type TStrList, returns a pointer to it of type PStrList;
NewStrListEx - creates a TStrListEx object, returns a pointer of the PStrListEx type.

The main set of **methods and properties** typical for TStrList:

Count - the number of lines in the list;
Add(s) - adds a line to the end of the list;
Insert(i, s) - inserts a line at position i;
AddStrings(SL) - adds to the end of the list all lines from another object of the PStrList type;

© 2024 Carl Peeraer

Assign(SL) - assigns to the given list of strings the contents of the list of strings specified by the parameter;
Clear - clears the list, freeing the memory occupied by lines;
Delete(i) - deletes the line with index i;
DeleteLast - deletes the last line in the list;
IndexOf(s) - finds the string s in the list (case-sensitive), and returns the index of the found string, or -1 if the string is not found;
IndexOf_NoCase(s) - the same as the previous method, but the search for the specified string is case-insensitive (meaning the case of the Ascii encoding, the case of national characters cannot be ignored by this method);
IndexOfStrL_NoCase(s, n) - the same as the previous method, but only the first n characters of the string are compared;
Find(s, i) - performs a search in a sorted list (the method of halving is used, which increases the speed of searching in large lists of strings);
Items[i] - this property provides access to individual list items as Ansi-strings, and allows them to be read or modified;
Last - property for accessing the last line in the list (equivalent to Items [Count-1])
ItemPtrs[i]- this property, unlike Items, allows you to get the address of the beginning of the list line by its index. For the purpose of reading strings or modifying them in place, this method is preferable in terms of performance, since there is no need to allocate memory on the heap for a copy of the string. Of course, when modifying strings "in place", it is necessary to control the possible overstepping of the string when writing to it, otherwise unpleasant consequences are guaranteed. Up to the immediate crash of the application or the occurrence of a memory access exception - Access Violation, or, worse, to the corruption of the service fields of the heap manager, and the subsequent crash of the application, the causes of which are much more difficult to identify and fix;
Sort(casesensitive) - sorts strings (after which you can use the Find method, for example);
AnsiSort(casesensitive) - sorts strings as ANSI (i.e., the order of national characters is also taken into account);
Swap(i1, i2) - swaps strings in the list;
Move(i1, i2) - moves the line with index i1 to position i2;
Text - provides the ability to work with all lines of the list as one line. When reading this property, all lines - quickly - are combined into one text, consisting of lines of text separated by characters # 13 # 10, when assigning a value to this property, the original large line - quickly - is split into separate lines based on the presence of combinations of characters # 13 # 10 , # 13, # 0 at the end of each substring. An important detail: immediately after assigning a value to this property, all lines in the list are stored in a contiguous piece of memory, one after another, each ending with byte # 0. This circumstance can be used in order to perform fast processing of large texts (having received a pointer to the first line, with index 0, through the ItemPtrs [0] property, then you can "run" all lines with the pointer,
SetText(s, append) - an additional method that allows you to quickly add text from a single line, breaking it into lines in much the same way as it is done when assigning to the Text property;
SetUnixText(s, append) - similar to the previous one, but single characters # 10 (standard for Unix systems) are also considered as separators;

© 2024 Carl Peeraer

Join(s) - a function that returns concatenation of strings like the Text property does, but the line separator is specified by a parameter.

A special group can be divided into a set of **methods for exchanging data with files and data streams:**

LoadFromFile(s) - loads a list of strings from a text file;
SaveToFile(s) - saves a list of lines as text in a file;
LoadFromStream(strm) - loads a list of lines from the stream (reading it from the current position to the end of the stream);
SaveToStream(strm) - saves the list of strings to the stream (writing it from the current position in the stream);
AppendToFile(s) - adds lines from the list to the end of the specified file;
MergeFromFile(s) - adds lines from the specified file to the list of lines;

A list of strings can be used in a special way as a set of named values of the form <value_name> = <value> (similar to Ini files), and not only the '=' symbol can be used as a sign separating the value name from the value itself, but also, for example , the ':' character, and any other character. When creating the list, the symbol specified by the global variable DefaultNameDelimiter is used to initialize the delimiter, which by default stores the value '='. You can either change the value of this variable before creating the list, or you can change the NameDelimiter property of each individual list. Next, I will list the methods and properties for working with a list of strings as with a list of values:
Values[s] - this property allows you to read or change the value named s (if the name is not found during reading, an empty string is returned, if the name does not exist during writing, it is added);

IndexOfName(s) - returns the index of the string containing the value with the specified name;
LineName[i] - returns or changes (when writing) the name in the string with index i;
LineValue[i] - similarly for the value in the string with index i.

The **TStrListEx** object type, being an inheritor of the TStrList type, retains all these capabilities, and adds to them the ability to associate each line in the list with a numeric value, or a pointer - at will. Due to the fact that the policy of inheritance and abuse of virtual methods is not welcomed in KOL, you should not use object parameter polymorphism, and work with TStrListEx as with a TStrList object. Nothing terrible will happen (most likely), but when deleting lines, changing their order and other operations, the agreement between the lines and their associated "objects" is likely to be broken. Conclusion from the above: if you already use an object of type TStrListEx, then this object in any code that works with it must be declared exactly as TStrListEx.

All of the above methods of the **TStrList** object also hold for **TStrListEx**, but if the specialized analogs developed for **TStrListEx** are not used, then the object is assumed to be null (i.e. when

© 2024 Carl Peeraer

calling Insert (i, s)a row will be inserted at position i, and the value 0 will be inserted as its object, also at position i). Below is a list of methods and properties specific to TStrListEx:
Objects[i] - access to objects associated with strings, by the string index;
LastObj - access to the last "object", equivalent to Objects [Count-1];
Assign(SLex) - assigns strings and objects from the specified extended list of strings;
AddObject(s, o) - adds a line immediately with the "object" -number associated with it;
InsertObject(i, s, o) - similar to the previous one, inserts a line together with the object, at position i;
IndexOfObj(o) - finds the first object o, and returns its index (or -1 if no such "object" is found).

In addition to the given object types for storing lists of strings, KOL also has others (but they are moved to the additional module **KOLadd.pas**): **TFastStrListEx**, **TWStrList**, **TWStrListEx**.

The **TFastStrListEx** object is similar in functionality to the **TStrListEx** type, but is optimized for fast addition of lines. In order not to clutter up this text, I will not give its detailed description here, you can always look into the source code and familiarize yourself with the set of its properties and methods.

The **TWStrList** and **TWStrListEx** objects are similar to the TStrList and TStrListEx objects, but are focused on working with Unicode strings (WideString), consisting of double-byte characters (WideChar). Therefore, it does not make much sense to describe them in detail as well, almost everything said for ordinary lists of strings is also true for these lists, except for the type of strings stored in them.

Unlike many other objects in KOL, when the **UNICODE_CTRL** directive is used, the **TStrList** and **TStrListEx** string lists do not automatically become UNICODE string lists. In order not to insert conditional compilation directives of the form {$ IFDEF UNICODE_CTRLS}... {$ ELSE}... {$ ENDIF} into your code, the **TKOLStrList** / **TKOLStrListEx** types are declared in KOL, which are equivalent to **TStrList** / **TStrListEx** in the case of using Ansi string types, and are replaced **TWStrList** / **TWStrListEx** in case of adding the **UNICODE_CTRLS** conditional compilation symbol. This is necessary because sometimes in Ansi applications you need to work with UNICODE text and (even more often) on the contrary - in a UNICODE project you need to process a "clean" Ansi list of strings.

Therefore, when creating a project that can compile with or without this option, you should use the **TKOLStrList** data type and the **NewKOLStrList** function to create it.

4.14.1 List of Strings - Syntax

```
function NewStrList: PStrList;
```
Creates string list object.

```
function WStrLen( W: PWideChar ): Integer;
```


© 2024 Carl Peeraer

Returns Length of null-terminated Unicode string.

`function` **NewStrListEx**`: PStrListEx;`
Creates extended string list object.

`procedure` **WStrCopy**`( Dest, Src: PWideChar );`
Copies null-terminated Unicode string (terminated null also copied).

`procedure ( Dest, Src: PWideChar; MaxLen: Integer );`
Copies null-termin**WStrLCopy**ated Unicode string (terminated null also copied).

`function` **WStrCmp**`( W1, W2: PWideChar ): Integer;`
Compares two null-terminated Unicode strings.

`function` **WStrCmp_NoCase**`( W1, W2: PWideChar ): Integer;`
Compares two null-terminated Unicode strings.

`type` **PWStrList** `= ^ TWstrList;`

`function` **NewWStrList**`: PWStrList` 119`;`
Creates new TWStrList 124 object and returns a pointer to it.

`function` **NewWStrListEx**`: PWStrListEx;`
Creates new TWStrListEx objects and returns a pointer to it.

`function` **NewKOLStrList**`: PKOLStrList;`

`function` **NewKOLStrListEx**`: PKOLStrListEx;`

TStrList

TStrList`( unit KOL.pas )` ← `TObj` 92 ← `_TObj` 92
TStrList `= object( TObj` 92 `)`

Easy string list implementation (non-visual, just to store string data). It is well improved and has very high performance allowing to work fast with huge text files (more then megabyte of text data). Please note that #0 charaster if stored in string lines, will cut it preventing reading the rest of a line. Be careful, if your data contain such characters.

© 2024 Carl Peeraer

TStrList properties

`property Values[ const AName: Ansistring ]: Ansistring;`
by Dod. Returns right side of a line starting like Name=...

`property Count: integer;`
Number of strings in a string list.

`property Items[ Idx: integer ]: Ansistring; default;`
Strings array items. If item does not exist, empty string is returned. But for assign to property, string with given index *must* exist.

`property ItemPtrs[ Idx: Integer ]: PAnsiChar;`
Fast access to item strings as PChars.

`property Text: Ansistring;`
Content of string list as a single string (where strings are separated by characters $0D,$0A).

TStrList methods

`function IndexOfName( AName: Ansistring ): Integer;`
by Dod. Returns index of line starting like Name=...

`function Add( const S: Ansistring ): integer;`
Adds a string to list.

`procedure AddStrings( Strings: PStrList );`
Merges string list with given one. Very fast - more preferrable to use than any loop with calling Add 120 method.

`procedure Assign( Strings: PStrList );`
Fills string list with strings from other one. The same as AddStrings 120, but Clear 120 is called first.

`procedure Clear;`
Makes string list empty.

`procedure Delete( Idx: integer );`
Deletes string with given index (it *must* exist).

 © 2024 Carl Peeraer

```
procedure DeleteLast;
```
Deletes the last string (it *must* exist).

```
function IndexOf( const S: AnsiString ): integer;
```
Returns index of first string, equal to given one.

```
function IndexOf_NoCase( const S: Ansistring ): integer;
```
Returns index of first string, equal to given one (while comparing it without case sensitivity).

```
function IndexOfStrL_NoCase( Str: PAnsiChar; L: Integer ): integer;
```
Returns index of first string, equal to given one (while comparing it without case sensitivity).

```
function Find( const S: AnsiString; var Index: Integer ): Boolean;
```
Returns Index of the string, equal or greater to given pattern, but works only for sorted TStrList object. Returns TRUE if exact string found, otherwise nearest (greater then a pattern) string index is returned, and the result is FALSE. And in such _case Index is returned negated when the S string is less then the string found.

```
function FindFirst( const S: AnsiString; var Index: Integer ): Boolean;
```
Like above but always returns Index of the first string, equal or greater to given pattern. Also works only for sorted TStrList object. Returns TRUE if exact string found, otherwise nearest (greater then a pattern) string index is returned, and the result is FALSE.

```
procedure Insert( Idx: integer; const S: Ansistring );
```
Inserts string before one with given index.

```
procedure Move( CurIndex, NewIndex: integer );
```
Moves string to another location.

```
procedure SetText( const S: Ansistring; Append2List: Boolean );
```
Allows to set strings of string list from given string (in which strings are separated by $0D,$0A or $0D characters). Text 120 must not contain #0 characters. Works very fast. This method is used in all others, working with text arrays (LoadFromFile 122, MergeFromFile 122, Assign 120, AddStrings 120).

```
procedure SetUnixText( const S: AnsiString; Append2List: Boolean );
```
Allows to assign UNIX-style text (with #10 as string separator).

```
function Last: AnsiString;
```
Last item (or '', if string list is empty).

© 2024 Carl Peeraer

```
procedure Swap( Idx1, Idx2: Integer );
```
Swaps to strings with given indexes.

```
procedure Sort( CaseSensitive: Boolean );
```
Call it to sort string list.

```
procedure AnsiSort( CaseSensitive: Boolean );
```
Call it to sort ANSI string list.

```
procedure SortEx( const CompareFun: TCompareEvent );
```
Call it to sort via your own compare procedure

```
function Join( const sep: AnsiString ): AnsiString;
```
by Sergey Shishmintzev

```
function LoadFromFile( const FileName: KOLString ): Boolean;
```
Loads string list from a file. (If file does not exist, nothing happens). Very fast even for huge text files.

```
procedure LoadFromStream( Stream: PStream; Append2List: Boolean );
```
Loads string list from a stream (from current position to the end of a stream). Very fast even for huge text.

```
procedure MergeFromFile( const FileName: KOLString );
```
Merges string list with strings in a file. Fast.

```
function SaveToFile( const FileName: KOLString ): Boolean;
```
Stores string list to a file.

```
procedure SaveToStream( Stream: PStream );
```
Saves string list to a stream (from current position).

```
function AppendToFile( const FileName: KOLString ): Boolean;
```
Appends strings of string list to the end of a file.

TStrListEx

TStrListEx(unit KOL.pas) ← TStrList 119 ← TObj 92 ← _TObj 92

© 2024 Carl Peeraer

`TStrListEx = object( TStrList 119 )`
Extended string list object. Has additional capability to associate numbers or objects with string list items.

TStrListEx properties

`property Objects[ Idx: Integer ]: DWORD;`
Objects are just 32-bit values. You can treat and use it as pointers to any other data in the memory. But it is your task to free allocated memory in such case therefore.
If the last item of a string list is deleted vis DeleteLast 120 method (but not via Delete 120 method), it's object still is preserved. As well, it is possible to set Objects[idx] for idx >= Count 120. To get know object's count, rather then strings count, use ObjectCount 123 property.

`property ObjectCount: Integer;`
Returns number of objects available. This value can differ from Count 120 after some operations: objects are stored in the independant list and only synchronization is provided while using methods Delete 120, Insert 121, Add 120, AddObject 124, InsertObject 124 while changing the list.
Properties, inherited from TStrList 119

TStrListEx methods

`destructor Destroy; virtual;`

`procedure AddStrings( Strings: PStrListEx );`
Merges string list with given one. Very fast - more preferrable to use than any loop with calling Add 120 method.

`procedure Assign( Strings: PStrListEx );`
Fills string list with strings from other one. The same as AddStrings 120, but Clear 120 is called first.

`procedure Clear;`
Makes string list empty.

`procedure Delete( Idx: integer );`
Deletes string with given index (it *must* exist).

`procedure DeleteLast;`
Deletes the last string and correspondent object in the list.

`procedure Move( CurIndex, NewIndex: integer );`
Moves string to another location.

`procedure Swap( Idx1, Idx2: Integer );`
Swaps to strings with given indexes.

© 2024 Carl Peeraer

```
procedure Sort( CaseSensitive: Boolean );
```
Call it to sort string list.

```
procedure AnsiSort( CaseSensitive: Boolean );
```
Call it to sort ANSI string list.

```
function LastObj: DWORD;
```
Object assotiated with the last string.

```
function AddObject( const S: AnsiString; Obj: DWORD ): Integer;
```
Adds a string and associates given number with it. Index of the item added is returned.

```
procedure InsertObject( Before: Integer; const S: AnsiString; Obj: DWORD );
```
Inserts a string together with object associated.

```
function IndexOfObj( Obj: Pointer ): Integer;
```
Returns an index of a string associated with the object passed as a parameter. If there are no such strings, -1 is returned.

TWStrList

TWStrList(unit KOL.pas) ← TObj 92 ← _TObj 92
TWStrList = object(TObj 92)
String list to store Unicode (null-terminated) strings.

TWStrList properties

```
property Items[ Idx: Integer ]: KOLWideString;
```
See also TStrList.Items 120
```
property ItemPtrs[ Idx: Integer ]: PWideChar;
```
See also TStrList.ItemPtrs 120
```
property Count: Integer;
```
See also TStrList.Count 120
```
property Text: KOLWideString;
```
See also TStrList.Text 120
Properties, inherited from TObj 92

TWStrList methods

```
procedure SetText( const Value: KOLWideString );
```
See also TStrList.SetText 121

```
destructor Destroy; virtual;
```

```
procedure Clear;
```
See also TStrList.Clear 120

```
function Add( const W: KOLWideString ): Integer;
```
See also TStrList.Add 120

```
procedure Insert( Idx: Integer; const W: KOLWideString );
```
See also TStrList.Insert 121

```
procedure Delete( Idx: Integer );
```
See also TStrList.Delete 120

```
procedure AddWStrings( WL: PWStrList 119 );
```
See also TStrList.AddStrings 120

```
procedure Assign( WL: PWStrList 119 );
```
See also TStrList.Assign 120

```
function LoadFromFile( const Filename: KOLString ): Boolean;
```
See also TStrList.LoadFromFile 122

```
procedure LoadFromStream( Strm: PStream; AppendToList: Boolean );
```
See also TStrList.LoadFromStream 122

```
function MergeFromFile( const Filename: KOLString ): Boolean;
```
See also TStrList.MergeFromFile 122

```
procedure MergeFromStream( Strm: PStream );
```
See also TStrList.MergeFromStream 125

```
function SaveToFile( const Filename: KOLString ): Boolean;
```
See also TStrList.SaveToFile 122

```
procedure SaveToStream( Strm: PStream );
```
See also TStrList.SaveToStream 122

```
function AppendToFile( const Filename: KOLString ): Boolean;
```
See also TStrList.AppendToFile 122

```
procedure Swap( Idx1, Idx2: Integer );
```
See also TStrList.Swap 121

© 2024 Carl Peeraer

```
procedure Sort( CaseSensitive: Boolean );
```
See also TStrList.Sort 122

```
procedure Move( IdxOld, IdxNew: Integer );
```
See also TStrList.Move 121

```
function IndexOf( const s: KOLWideString ): Integer;

function IndexOf_NoCase( const s: KOLWideString ): Integer;

function Last: KOLWideString;

procedure Put( Idx: integer; const Value: KOLWideString );
```

4.15 List of Files and Directories

List of Files and Directories - TDirList

Since I started talking about lists, it is natural to continue with a special kind of list - a list of file names. KOL has an object type **TDirList**, which greatly simplifies the work with directories. It encapsulates a call to API functions that view the contents of a folder and constitutes its directory. All you need to start working with files in the entire directory is to call one of the constructors (**NewDirList** or **NewDirListEx**), and get an object that stores the "dossier" for all ordered files located in the specified path (and meeting the stated requirements).

So, "constructors":

NewDirList(path, filter, attr)- creates a list of directories, reading files and / or directories (this depends on the parameter that sets the attributes of the searched file-director objects, for example, FILE_ATTRIBUTE_DIRECTORY will read only subdirectories, FILE_ATTRIBUTE_ARCHIVE - only files, and 0 - all names indiscriminately). The filter can only be a single filter, but it accepts the wildcard characters '*' and '?'.
NewDirListEx(path, filters, attr) - differs from the previous constructor in that it allows you to use several patterns, separated by the ';' character, and patterns prefixed with '^' are considered anti-filters - to exclude names that match such patterns.

© 2024 Carl Peeraer

The created object of type TDirList has the following methods and properties:

Path - a string that stores the path to the directory (always terminated by the '\' character, that is, you can always concatenate DL.Path + DL.Names [i] to get the full path to the i-th file);
Count - returns the number of file names and subdirectories in the list;
Names[i] - returns file names by their index;
IsDirectory[i] - checks that the object with index i is a directory;
Items[i] - complete structure with information about the file / directory (it contains all the information provided by the system, including the file size, date of its creation / modification / last access, short name, attributes);
Clear - clears the list;
ScanDirectory(path, filter, attr) - allows you to scan the contents of another (or rescan the contents of the same) directory, similar to the **NewDirList** constructor;
ScanDirectoryEx(path, filters, attr) - similar to the previous method, but according to advanced rules, similar to the **NewDirListEx** constructor;
Sort(rules) - sorts names according to the specified rules. A whole array of rules is specified, which are applied sequentially during comparison until the first rules detect differences between names. For example, the rule **sdrFoldersFirst** does not distinguish between files and files, and between directories and directories, but distinguishes only directories from files. Some rules are used as modifiers to apply other rules, for example **sdrCaseSensitive**;
FileList(separator, dirs, fullpaths) - returns a list of files as a string, in which the files are separated by the specified separator;
OnItem - an event that is triggered for each read item when scanning a directory, and allows you to make a decision about including or not including a name in the list in accordance with the algorithm specified in the event handler. Of course, in order for the handler for this event to be assigned before scanning the directory, when calling the constructor, pass an empty string as a path, then assign your handler to the object, and only then call the scanning method **ScanDirectory** or **ScanDirectoryEx**.

Although the object described here does not allow scanning the contents of a directory for all subdirectories, nevertheless, organizing a recursive traversal of the entire folder tree is a completely solvable task even for a beginner. An example of such a recursive traversal can be found in the KOL.pas module itself, in the implementation of the **DirectorySize** function, which just uses the **TDirList** object to scan the contents of directories.

4.15.1 List of Files and Directories - Syntax

```
type TSortDirRules =( sdrNone, sdrFoldersFirst, sdrCaseSensitive, sdrByName,
sdrByExt, sdrBySize, sdrBySizeDescending, sdrByDateCreate, sdrByDateChanged,
sdrByDateAccessed, sdrInvertOrder );
```
List of rules (options) to sort directories. Rules are passed to Sort method in an array, and first placed rules are applied first.

```
function NewDirList( const DirPath, Filter: KOLString; Attr: DWORD ): PDirList;
```
Creates directory list object using easy one-string filter. If Attr = FILE_ATTRIBUTE_NORMAL, only

© 2024 Carl Peeraer

files are scanned without directories. If Attr = 0, both files and directories are listed.

```
function NewDirListEx( const DirPath, Filters: KOLString; Attr: DWORD ): PDirList;
```
Creates directory list object using several filters, separated by ';'. Filters starting from '^' consider to be anti-filters, i.e. files, satisfying to those masks, are skept during scanning.

```
var DefSortDirRules: array[ 0 . . 3 ] of TSortDirRules =( sdrFoldersFirst,
sdrByName, sdrBySize, sdrByDateCreate );
```
TSortDirRules 127

Default rules to sort directory entries.

```
function DirectorySize( const Path: KOLString ): I64;
```
I64 59

Returns directory size in bytes as large 64 bit integer.

TDirList

```
TDirList( unit KOL.pas ) ← TObj ← _TObj
TDirList = object( TObj )
```
TObj 92 _TObj 92 TObj 92

Allows easy directory scanning. This is not visual object, but storage to simplify working with directory content.

TDirList properties

```
property Items[ Idx: Integer ]: PFindfileData; default;
```
Full access to scanned items (files and subdirectories).

```
property IsDirectory[ Idx: Integer ]: Boolean;
```
Returns TRUE, if specified item represents a directory, not a file.

```
property Count: Integer;
```
Number of items.

```
property Names[ Idx: Integer ]: KOLString;
```
Full long names of directory items.

```
property Path: KOLString;
```
Path of scanned directory.

© 2024 Carl Peeraer

TDirList methods

```
destructor Destroy; virtual;
```
Destructor. As usual, call **Free** method to destroy an object.

```
procedure Clear;
```
Call it to clear list of files.

```
procedure ScanDirectory( const DirPath, Filter: KOLString; Attr: DWord );
```
Call it to rescan directory or to scan another directory content (method Clear 129 is called first). Pass path to directory, file filter and attributes to scan directory immediately.
Note: Pass FILE_ATTRIBUTE_... constants or-combination as Attr parameter. If 0 passed, both files and directories are listed.

```
procedure ScanDirectoryEx( const DirPath, Filters: KOLString; Attr: DWord );
```
Call it to rescan directory or to scan another directory content (method Clear 129 is called first). Pass path to directory, file filter and attributes to scan directory immediately.
Note: Pass FILE_ATTRIBUTE_... constants or-combination as Attr parameter.

```
procedure Sort( Rules: array of TSortDirRules );
```
(TSortDirRules: 127)
Sorts directory entries. If empty rules array passed, default rules array DefSortDirRules 128 is used.

```
function FileList( const Separator: KOLString; Dirs, FullPaths: Boolean ):
KOLString;
```
Returns a string containing all names separated with Separator. If Dirs=FALSE, only files are returned.

```
procedure DeleteItem( Idx: Integer );
```
Allows to delete an item from the directory list (not from the disk!)

```
procedure AddItem( FindData: PFindFileData );
```
Allows to add arbitrary item to the list.

```
procedure InsertItem( idx: Integer; FindData: PFindFileData );
```
Allows to add arbitrary item to the list.

TDirList events

```
property OnItem: TOnDirItem;
```


© 2024 Carl Peeraer

This event is called on reading each item while scanning directory. To use it, first create PDirList object with empty path to scan, then assign OnItem event and call ScanDirectory 129 with correct path.

4.16 Tracking Changes on Disk

To track events such as changes to the contents of a directory, or modification of files in a directory, KOL has a special **TDirChange** object (located in the **KOLadd.pas** module). Its purpose is to trigger a designated event every time one of the changes specified by the watch filter occurs.

Constructor:

NewDirChangeNotifier(s, filter, watchsubtree, onchange)- creates and returns an object of type **PDirChange** (however, if the parameters are specified incorrectly, nil is returned). As a filter, you can specify the names of files, directories, attributes, file size, as well as the time of creation, modification and last access to files.

This object does not contain any essential properties or methods (perhaps you can mark the **Path** property, which stores the path to the monitored directory). The event that is called when a change is detected in the tracking area is set in the object's constructor. When it is triggered, the handler gets the path to the directory in which the change was detected (useful for the case when the object monitors the directory at once along with all subdirectories). In order to determine exactly what changes have occurred, the program must store information about the previous state of the directory, when changes occur, rescan the directory, and perform a comparison with its own code.

4.16.1 Tracking Changes on Disk - Syntax

```
type PDirChange = ^TDirChange;
```

```
TOnDirChange = procedure (Sender: PDirChange; const Path: KOLString) of object;
```

Event type to define OnChange event for folder monitoring objects.

```
TFileChangeFilters = (fncFileName, fncDirName, fncAttributes, fncSize, fncLastWrite,
fncLastAccess, fncCreation, fncSecurity);
```

Possible change monitor filters.

```
TFileChangeFilter = set of TFileChangeFilters;
```

Set of filters to pass to a constructor of TDirChange object.

```
TDirChange = object(TObj)
```

Object type to monitor changes in certain folder.

 © 2024 Carl Peeraer

TDirChange Properties

```
property Handle: THandle;
```
Handle of file change notification object.

```
property Path: KOLString;
```
Path to monitored folder (to a root, if tree of folders is under monitoring).

```
property OnChange: TOnDirChange;
```

```
property OnExecute: TOnEvent;
```

```
function NewDirChangeNotifier( const Path: KOLString; Filter: TFileChangeFilter;
WatchSubtree: Boolean; ChangeProc: TOnDirChange; OnExecuteProc: TOnEvent) :
PDirChange;
```
Creates notification object **TDirChange** 130. If something wrong (e.g., passed directory does not exist), nil is returned as a result. When change is notified, **ChangeProc** is called always in main thread context.
(Please note, that **ChangeProc** can not be nil).
If empty filter is passed, default filter is used: [**fncFileName..fncLastWrite**].

4.17 INI Files

The program can store its settings in the **registry** or **ini-files**. Or it can store its settings in both the **registry** and **ini-files**. Storing such configuration data in the registry allows each computer user to have their own settings, as well as to save the settings for an application that runs in conditions where there is no permission to directly change the contents of the (disk) media. For example, it can be a compact disc, or a media with a blocking of the possibility of changing information. Storing settings in an **ini-file** has its advantages. For example, it allows all computer users to have the same settings that are not lost when reinstalling the operating system. In addition, the use of ini files by applications slightly "lightens" the size of the operating system registry.

Conclusion from the above: it is sometimes necessary to work with **ini-files** when developing applications. And there is such an object in KOL.

© 2024 Carl Peeraer

Its constructor:

OpenIniFile(s)- creates an object of type **TIniFile**, returning a pointer to it of type **PIniFile**. The newly created object is either linked to an existing **ini-file** (named s), or if such a file does not exist at the time of the call, it is created automatically.
This KOL object for working with **ini-files** has a feature that allows in some cases to use the same procedure in order to ensure both loading and saving of settings. Both reading and writing of key values are performed by the same methods of the object. What exactly to do, read or write, determines the mode of work with the settings file (the Mode property).

Properties and methods of the TIniFile object:

Mode - operating mode: **ifmRead** - read, **ifmWrite** - write;
Filename - the name of the settings file (read-only);
Section - section of the ini-file (in the settings file, the section begins with a line containing the section name in square brackets);
ValueInteger(key, i)- in read mode, it returns the value of the key, while the value i is used as the default value, which is returned if there is no key in the current section; in write mode, the same method writes a new value i for key;
ValueString(key, s) - similar to the previous method, but for the string value of the key;
ValueBoolean(key, s) - the same for a boolean value;
ValueData(key, buf, i) - similar to the previous methods, but the work is carried out with a data block of length i bytes;
ValueDouble(key, d) - the same for a real number of type Double;
ClearAll - complete cleaning of the settings file (all sections are deleted together with all keys);
ClearKey(s) - removes the key s in the current section;
GetSectionNames(SL) - reads into the object of the list of lines (PStrList) the names of all sections from the settings file;
GetSectionData(SL) - reads the entire contents of the current section into the object of the list of strings.

© 2024 Carl Peeraer

I will give a small example of how it is possible with the same code to ensure both recording and saving of settings. Suppose, for example, we want to save the coordinates of the application window in the settings file at the end of the work, and restore them from there at the beginning of work. Let's create a method that will perform both of these operations:

```
procedure MyObj.ReadWriteIni (write: boolean);
var ini: PIniFile;
begin
 ini: = OpenIniFile (GetStartDir + 'my.ini');
 if write then ini.Mode: = ifmWrite;
 ini.Section = 'position';
 form.Left: = ini.ValueInteger ('Left', form.Left);
 form.Top: = ini.ValueInteger ('Top', form.Top);
 ini.Free;
end;
```

Now, when starting the application, we will organize a call to this method with the false parameter, and when closing - with the true parameter. I leave it to you to figure out why this code will do everything that is required of it, although the same operators work in both cases.

4.17.1 INI Files - Syntax

`type TIniFileMode =( ifmRead, ifmWrite );`
ifmRead is default mode (means "read" data from ini-file. Set mode to ifmWrite to write data to ini-file, correspondent to TIniFile133.

`function OpenIniFile( const FileName: KOLString ): PIniFile;`
Opens ini file, creating TIniFile133 object instance to work with it.

TIniFile

TIniFile(unit KOL.pas) ← TObj92 ← _TObj92
TIniFile = object(TObj92)
Ini file incapsulation. The main feature is what the same block of read-write operations could be defined (difference must be only in Mode133 value).

TIniFile properties

`property Mode: TIniFileMode;` 133
ifmWrite, if write data to ini-file rather than read it.

`property FileName: KOLString;`

© 2024 Carl Peeraer

Ini file name.

`property Section: KOLString;`
Current ini section.

TIniFile methods

`destructor Destroy; virtual;`
Destructor

`function ValueInteger( const Key: KOLString; Value: Integer ): Integer;`
Reads or writes integer data value.

`function ValueString( const Key: KOLString; const Value: KOLString ): KOLString;`
Reads or writes string data value.

`function ValueDouble( const Key: KOLString; const Value: Double ): Double;`
Reads or writes Double data value.

`function ValueBoolean( const Key: KOLString; Value: Boolean ): Boolean;`
Reads or writes Boolean data value.

`function ValueData( const Key: KOLString; Value: Pointer; Count: Integer ): Boolean;`
Reads or writes data from/to buffer. Returns True, if success.

`procedure ClearAll;`
Clears all sections of ini-file.

`procedure ClearSection;`
Clears current Section 134 of ini-file.

`procedure ClearKey( const Key: KOLString );`
Clears given key in current section.

`procedure GetSectionNames( Names: PKOLStrList );`
Retrieves section names, storing it in string list passed as a parameter. String list does not cleared before processing. Section 134 names are added to the end of the string list.

`procedure SectionData( Names: PKOLStrList );`
Read/write current section content to/from string list. (Depending on current Mode 133 value).

© 2024 Carl Peeraer

4.18 An Array of Bit Flags

Sometimes a program needs to organize an array of boolean flags of a large (and sometimes very large) dimension. A specially designed **TBits** object (**in the KOLadd module**) will help you to accomplish this task. Flags are stored in it as a single continuous array of bits (packed by 8 bits per byte). At the same time, a sufficiently high speed of work with flags is provided, and there are a number of quick operations that can increase the efficiency of work when performing some traditional tasks for which such arrays are usually used.

Constructor:

NewBits - creates an empty object to store a dynamic array of flags, returns a pointer of the **PBits** type to this object.

Methods and properties:

Bits[i]- access to individual flags of the array. Reading outside the array always returns false. Writing outside the array ensures that the array grows automatically (if written to true);
Count - the number of flags in the array (read-only);
Size - the size of the flags array in bytes (also read-only);
Capacity - the maximum number of flags for the storage of which memory is reserved. If the number of flags when adding new ones begins to exceed the reserved memory, then this value increases, and the memory for storing the flags is reallocated. In this case, if required, the accumulated flags are moved to a new location - in the same way as for lists. To prevent too frequent reallocation of memory, you should set the value of the Capacity property in the program, sufficient to work for a long time without the need to reallocate memory;
IndexOf(b) - returns the index of the first flag with the specified boolean value;
Openbit - similar to the previous one, but it returns the index of the first flag false in the array;
InstallBits(i, j, b) - for a continuous group of flags starting from index i and length j sets the value b;
Clear - clears the array of flags, freeing the occupied memory. If the array is assumed to be infinitely expanding to the right, then this operation is semantically equivalent to writing false to all the flags of the array;
Copy(i, j) - creates a new object TBits on the basis of this (returning a pointer to it of the **PBits** type), copying j flags into it, starting from index i;
Range(i, j) - a function equivalent to the previous one;
AssignBits(to_i, from_bits, from_i, j) - copies j flags from another **TBits** object from position from_i;
SaveToStream(strm)- saves an array of flags to the stream from the current position. First, the number of bits in the array is written, then the bits themselves;
LoadFromStream(strm)- loads an array of bits from a stream. the data must have been saved to the stream earlier by the **SaveToStream** method.

© 2024 Carl Peeraer

4.18.1 An Array of Bit Flags - Syntax

```
function NewBits: PBits;
```
Creates variable-length bits array object.

```
type
PBits = ^TBits;
```

```
TBits = object( TObj )
```
92
Variable-length bits array object. Created using function **NewBits** 136.

```
destructor Destroy; virtual;
```

```
property Bits[ Idx: Integer ]: Boolean;
```

```
property Size: Integer;
```
Size in bytes of the array. To get know number of bits, use property **Count** 136.

```
property Count: Integer;
```
Number of bits an the array.

```
property Capacity: Integer;
```
Number of bytes allocated. Can be set before assigning bit values to improve performance (minimizing amount of memory allocation operations).

```
function Copy( From, BitsCount: Integer ): PBits;
```
Use this property to get a sub-range of bits starting from given bit and of **BitsCount** bits count.

```
function IndexOf( Value: Boolean ): Integer;
```
Returns index of first bit with given value (True or False).

```
function OpenBit: Integer;
```
Returns index of the first bit not set to true.

```
procedure Clear;
```
Clears bits array. **Count** 136, **Size** 136 and **Capacity** 136 become 0.

```
function LoadFromStream( strm: PStream ): Integer;
```
Loads bits from the stream. Data should be stored in the stream earlier using **SaveToStream** 137 method. While loading, previous bits data are discarded and replaced with new one totally. In part, Count of bits also is changed. Count of bytes read from the stream while loading data is

© 2024 Carl Peeraer

returned.

`function SaveToStream( strm: PStream ): Integer;`
Saves entire array of bits to the stream. First, Count of bits in the array is saved, then all bytes containing bits data.

`function Range( Idx, N: Integer ): PBits;`
Creates and returns new **TBits** 136 object instance containing N bits starting from index Idx. If you call this method, you are responsible for destroying returned object when it become not necessary.

`procedure AssignBits( ToIdx: Integer; FromBits: PBits; FromIdx, N: Integer );`
Assigns bits from another bits array object. N bits are assigned starting at index ToIdx.

`procedure InstallBits( FromIdx, N: Integer; Value: Boolean );`
Sets new Value for all bits in range [FromIdx, FromIdx+Count-1].

`function CountTrueBits: Integer;`
Returns count of bits equal to TRUE.

4.19 Tree in Memory

In order to be able to efficiently store and process data organized in the form of trees in memory, the object type **TTree** was created (**in the KOLadd module**).

Constructor:

NewTree(Parent) or **NewTree (Parent, Name)** - creates a new node subordinate to the "parent" node Parent (can be nil to create the top-level node). A pointer to the created object, of type **PTree**, is returned. The first constructor should be used in projects that have the **TREE_NONAME** conditional compilation symbol. In this case, the tree nodes do not have a Name property and are not intended to hold the string as the main element of the node. In the second case, the constructor (and the corresponding Name property) uses regular strings, or Unicode strings (WideString) can be used.
To switch to using **Unicode strings**, you must include the **TREE_WIDE** conditional compilation symbol in your project.

Methods and properties:

© 2024 Carl Peeraer

Name - node name. As noted, this property does not exist if the project uses the conditional compilation symbol **TREE_NONAME**;
Data - node data. Generally speaking, any pointer or 32-bit number;
Count - the number of subordinate nodes of the next level (i.e. the number of nodes, the immediate parent of which is this node);
Total - the total number of subordinate nodes of all lower levels (but the node itself is not counted, as for Count);
Items[i] - access to the list of subordinate nodes (the TList object is used in the implementation to store the list of child nodes);
Add(Node) - Adds the specified node to the end of the list of subordinate nodes. If at this moment the node Node was subordinate (to the same, go to some other) node, then it is preliminarily excluded from the list of its previous parent. Thus, you can combine individual trees, or move nodes in the tree together with bunches of child nodes strung on them;
Insert(i, Node)- inserts the specified node into the list of child nodes, at position i. As with the Add method, detaches the inserted node from its previous parent;
SwapNodes(i, j) - swaps nodes with indices i and j places;
SortByName - sorts nodes by name (the Name field must exist);
Parent - parent node;
Index - own index of the node in the list of child nodes of the parent node (Parent);
PrevSibling - returns the pointer of the previous node in the list of child nodes of its parent (or nil if there is no such node);
NextSibling - returns the pointer of the next node in the list of child nodes of its parent (or nil);
Root - returns the pointer of the parent node of the topmost level (for a node that does not have a parent, it itself is returned);
Level - returns the level of the node, i.e. the number of ancestors of the node in the tree hierarchy (0 is returned for the root node);
IsParentOfNode(Node) - checks if the given node is the ancestor of the specified Node in the tree hierarchy;
IndexOf(Node) - returns the "total" index of the child node of any nesting level Node. Note: the total or general index is an index in the array, which is conventionally built as a sequence of all child nodes, each of which is counted along with all its subordinate nodes. It should not be confused with the index of a node in a sibling list;

4.19.1 Tree in Memory - Syntax

```
type
PTree = ^TTree;

TTree = object( TObj 92 )
```

Object to store tree-like data in memory (non-visual).

```
function NewTree( AParent: PTree ): PTree;
```

Nameless version (for case when **TREE_NONAME** symbol is defined).
Constructs tree node, adding it to the end of children list of the AParent. If AParent is nil, new root tree node is created.

© 2024 Carl Peeraer

`function` **`NewTree`**`( AParent: PTree; const AName: WideString ): PTree;`
WideString version (for case when **TREE_WIDE** symbol is defined).
Constructs tree node, adding it to the end of children list of the AParent. If AParent is nil, new root tree node is created.

`function` **`NewTree`**`( AParent: PTree; const AName: AnsiString ): PTree;`
Constructs tree node, adding it to the end of children list of the AParent. If AParent is nil, new root tree node is created.

`constructor` **`CreateTree`**`( AParent: PTree; const AName: AnsiString );`

`destructor` **`Destroy`**`; virtual;`

`procedure` **`Clear`**`;`
Destoyes all child nodes.

TTree properties

If **TREE_WIDE** symbol is defined:
`property` **`Name:`** `WideString read fNodeName write fNodeName;`

Default:

`property` **`Name:`** `AnsiString read fNodeName write fNodeName;`
Optional node name.

`property` **`Data:`** `Pointer;`
Optional user-defined pointer.

`property` **`Count:`** `Integer;`
Number of child nodes of given node.

`property` **`Items`**`[ Idx: Integer ]: PTree;`
Child nodes list items.

`procedure` **`Add`**`( Node: PTree );`
Adds another node as a child of given tree node. This operation as well as Insert can be used to move node together with its children to another location of the same tree or even from another tree.
Anyway, added Node first correctly removed from old place (if it is defined for it).

© 2024 Carl Peeraer

But for simplest task, such as filling of tree with nodes, code should looking as follows:

```
Node := NewTree 138 ( nil, 'test of creating node without parent' );
RootOfMyTree.Add( Node );
```

Though, this code gives the same result as:

```
Node := NewTree 138 ( RootOfMyTree, 'test of creatign node as a child' );
```

`procedure Insert( Before, Node: PTree );`

Inserts earlier created 'Node' just before given child node 'Before' as a child of given tree node. See also Add method.

`property Parent: PTree;`

Returns parent node (or nil, if there is no parent).

`property Index: Integer;`

Returns an index of the node in a list of nodes of the same parent (or -1, if Parent is not defined).

`property PrevSibling: PTree;`

Returns previous node in a list of children of the Parent. Nil is returned, if given node is the first child of the Parent or has no Parent.

`property NextSibling: PTree;`

Returns next node in a list of children of the Parent. Nil is returned, if given node is the last child of the Parent or has no Parent at all.

`property Root: PTree;`

Returns root node (i.e. the last Parent, enumerating parents recursively).

`property Level: Integer;`

Returns level of the node, i.e. integer value, equal to 0 for root of a tree, 1 for its children, etc.

`property Total: Integer;`

Returns total number of children of the node and all its children counting its recursively (but node itself is not considered, i.e. Total for node without children is equal to 0).

`procedure SortByName;`

Sorts children of the node in ascending order. Sorting is not recursive, i.e. only immediate children are sorted.

`procedure SwapNodes( i1, i2: Integer );`

© 2024 Carl Peeraer

Swaps two child nodes.

```
function IsParentOfNode( Node: PTree ): Boolean;
```
Returns true, if Node is the tree itself or is a parent of the given node on any level.

```
function IndexOf( Node: PTree ): Integer;
```
Total index of the child node (on any level under this node).

4.20 Elements of Graphics

Elements of graphics. Graphics Tools (TGraphicTool 151) and Drawing Canvas (TCanvas 147)

When creating projects on Windows, there is at least one task that is extremely difficult to program using a pure API. This is drawing on a DC (DC - Device Context) window or on a temporary image in memory (bitmap, i.e. a bitmap, a common name for a bitmap) using drawing tools - fonts, pencils and brushes ... Without object programming, this task becomes very difficult, the code is confusing and cumbersome, it is extremely easy to make mistakes in it, or simply "forget" to destroy any GDI tool (GDI - Graphic Device Interface). As a result, a so-called "resource leak" can appear in the program (which can lead to the fact that all resources in the system, the number of which is limited, run out,

To do this, both VCL and KOL create a **TCanvas** 147 object (only in KOL this is an object type, and in VCL - a class), and a set of tools encapsulating a **font** (font), a **brush** (brush) and a **pencil** (pen). The VCL uses a standard approach for these three graphics tools: there is a base class TGraphicsObject 151, which inherits the **TFont**, **TBrush** and **TPen** classes. In the KOL library, heirs are saved, and all three types of graphic tools are represented by the same object type **TGraphicTool** 151. Of course, they have different constructors, and the functionality of the object and the set of supported properties are different. For this reason, you should not try, for example, to change the FontName property for a brush - it still does nothing.

Constructors:

NewCanvas(DC) - creates a canvas object (if DC is specified, then this canvas is bound to an existing device context, usually for an in-memory image). In real programming, it is almost never necessary to create a canvas on your own, instead, you should use the Canvas property of the corresponding object to draw on a bitmap or in a window, the same applies to the following graphic tool constructors: usually you should use the Font, Brush and Pen properties of the canvas itself or a visual object;
NewFont - creates a font (returns a PGraphicTool);
NewBrush - creates a brush (returns PGraphicTool);
NewPen - creates a pencil (returns a PGraphicTool).

© 2024 Carl Peeraer

Now about the properties and methods of the canvas. They are basically the same as in the VCL. But there are also differences. The most important difference is how the GDI resources used by the application are prevented from growing too much. Dealing with the subtle and complex mechanism that implements this task in the VCL, I once spent more than one evening trying to figure out how it works there. This is somewhat similar to automatic garbage collection, which is sometimes used by memory managers. I decided to use a simpler algorithm for KOL. These are all internal implementation details, and I cannot dwell on them in more detail now, but I note that as a result, the canvas in the KOL library has a little more restrictions on its use.

For example, you should not take the canvas of a window object at an arbitrary point in time and start drawing something on it. You should draw exactly when processing of the OnPaint message begins (that is, when the system allows you to do it). What should you do if you need to render some animation and update the image in the window at certain intervals? The correct solution is to "tell" the system that the window is "damaged" and needs to be redrawn after the next time interval (for example, by the **OnTimer** event in the "clock" object) (for example, call the Invalidate method of the corresponding window object). After that, the system itself will send the message **WM_ERASEBKGND** (erase the background) and **WM_PAINT** (draw the content) to the window, and the **OnPaint** event handler will be called,

Graphical tool objects have several properties in common:

Handle - graphic object descriptor;
HandleAllocated - checks that the descriptor has been created (if you just refer to the Handle property, the descriptor will be created, so it makes no sense to check if it is equal to zero for this purpose);
OnChange - an event that is triggered when any properties of a graphic instrument change;
ReleaseHandle - takes ownership of the handle from the tool, returning the previous handle (Handle);
Assign(GT) - assigns all the properties of the specified tool to this one (the type of tool must be the same, i.e. a brush can be assigned to a brush, etc.);
Color - color.
All other properties are different, including by name.

Brush properties:

BrushBitmap - a bitmap (i.e. a picture) that is used for filling when a brush is used to fill;
BrushStyle - brush style (especially interesting are the bsSolid style - the main style for filling, and bsClear - when the brush does not work, this style of the "transparent" brush allows you to keep the matte intact in all drawing operations on the canvas);
BrushLineColor - line color for brush with styles (BrushStyle) of hatching with lines.

© 2024 Carl Peeraer

Pencil properties:

PenWidth - the width of the pencil in pixels;
PenStyle - pencil style (the degree of hatching of the line, there is also psClear, which allows you to ignore the pencil when drawing);
PenMode - pencil drawing mode (black, white, color, inverse, etc.);
GeometricPen - sets the so-called "geometric" pencil (in contrast to the non-geometric pencil, it allows you to set the type of outline of the line ends, see PenJoin and PenEndCap);
PenBrushStyle - brush style for a geometric pencil with shading;
PenBrushBitmap - bitmap for filling when drawing with a geometric pencil;
PenEndCap - the shape of the end of the line for a geometric pencil (round, square, flat);
PenJoin - way of connecting lines (round, boundary, middle).

Properties for the font (Font):

FontHeight - font height in pixels (an exception for a rich edit object: for it, the font height is set in special units called twips (literally: twentieth), and equal to 1/20 of the height of a point on the printer, or 1/1440 inch, or 1 / 10 pixels - approximate, depending on display resolution);
FontWidth - font width in pixels, if 0, then the standard font width for the current height is used, in KOL this can be changed by narrowing or thickening the fonts to your liking;
FontPitch - font style (monospaced, proportional or default);
FontStyle - a set of font styles (bold - see also FontWeight, oblique, underlined, strikethrough);
FontCharset - character set (forced selection of one or another national character set);
FontQuality - the quality of the font drawing;
FontOrientation - the angle of rotation of the font in grades, i.e. in 1/10 degree. A value of 900 corresponds to the rotation of the font 90 degrees counterclockwise. This property works only for TrueType fonts (for example, Arial or Times);
FontWeight - sets the exact value for font thickening. If set to a nonzero value, then the fsBold indication, i.e. bold is ignored in font styles. The value 700 corresponds to the fsBold style, the value 400 - to the fsNormal style, the others are in accordance with the obtained calibration scale;
FontName - font name;
IsFontTrueType - checks if the font is a TrueType font (ie "truly scalable").

In addition, a little more detailed information on the use of fonts in visual objects. In a KOL application (including MCK projects) it is possible not to specify a font at all, in this case the system font (extremely large, FixedSys) will be used. In general, MCK projects use the "default" font as the default font, the characteristics of which are recorded in the global **DefFont** structure. Initially, this is MS Sans Serif with a height of 0, i.e. the height depends on the default settings for the desktop (and the default font color is taken from the global variable **DefFontColor**, originally **clWindowText**). If you change these variables before creating the first visual, then all fonts (assigned by default) in the application will change.

As for KOL applications (i.e., those written without using MCK, if you don't change them at all and don't refer to the Font property, then the **FixedSys system font** will be used, and if you

© 2024 Carl Peeraer

refer to at least one font property of the visual object, then both this visual object and all its child objects will have their **DefFont** applied first, and then the specified property will be modified (if it is modified).

And finally, I will give a list of canvas properties and methods that you may need when programming drawing:

Handle - handle to the device context (the same DC with which the canvas object is associated). It is provided so that it remains possible to perform any low-level operations using API functions if there is no equivalent for them in the TCanvas object. Also, the canvas descriptor can always be passed as a parameter to functions that are focused on working with the DC device context, and may not know anything about the canvas, and not use its methods;
PenPos - the position of the pencil, remembers the last coordinate used in the MoveTo and LineTo methods;
Pen - a property that provides a "pencil" object;
Brush - a property that provides a brush object;
Font - a property that provides a "font" object;
Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4) - draws an elliptical arc - along the curve of the ellipse, limiting the ellipse to points (X1, Y1) and (X2, Y2) and drawing the curve counterclockwise starting with point (X3, Y3) and up to point (X4, Y4), use a pencil (Pen) for drawing;
Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4) - Draws a shape bounded by an arch and a chord connecting the ends of the arch. The bounding line is drawn using a pencil (Pen), the interior of the resulting shape is filled with a brush (Brush);
DrawFocusRect(R) - draws a focus frame along the specified rectangle (Pen is used in XOR mode, i.e. repeated call of the same method returns the image to its original state);
Ellipse(X1, Y1, X2, Y2) - draws an ellipse bounded by a rectangle specified by two vertices at points (X1, Y1) and (X2, Y2). To draw the border, use a pencil (Pen), fill the inside of the ellipse with a brush (Brush);
FillRect(R) - fills rectangle R using a brush (Brush);
FillRgn(rgn) - fills the specified region using a brush (Brush);
Floodfill(X, Y, Color, FillStyle) - filling the area either filled with Color, or vice versa, up to the border of Color, depending on the FillStyle, with a brush (Brush);
FrameRect(R) - draws the border of the specified rectangle using a brush (Brush);
MoveTo(X, Y) - moves the pencil to point (X, Y);
LineTo(X, Y) - draws a straight line from the current position of the pencil to point (X, Y) with the pencil tool (Pen);
Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4) - draws a sector based on an elliptical arc and centered in the center of an ellipse inscribed in a rectangle (X1, Y1), (X2, Y2), the arc is is drawn counterclockwise from point (X3, Y3) to point (X4, Y4), the border of the resulting shape is drawn with a pencil (Pen), and the inner part of the sector is painted over with a brush (Brush);
Polygon(pts) - draws a polygon based on an array of specified points (the last point in the array is connected to the first one), the border is drawn with a pencil, and the inner part is filled with a brush;
Polyline(pts) - draws a polyline along a given array of points using a pencil;

© 2024 Carl Peeraer

Rectangle(X1, Y1, X2, Y2) - draws a rectangle with vertices at points (X1, Y1), (X2, Y2), the border is drawn with a pencil, the interior is filled with a brush;
RoundRect(X1, Y1, X2, Y2, X3, Y3) - draws a rectangle with rounded corners, using an ellipse of height Y3 and width X3 for rounding;
TextOut(X, Y, s) - draws text s from point (X, Y) with the current font (Font) and filling the background with a brush (Brush);
ExtTextOut(X, Y, options, s, spacing) - draws text in a given rectangle using additional options and an array of letter spacing (spacing), for more details, see the description of the ExtTextOut API function, which this method calls;
DrawText(s, R, flags) - draws text in a rectangle using the DrawText API function and allows you to format the text in accordance with the specified flags;
TextRect(R, X, Y, s) - draws text, limiting the drawing area to rectangle R;
TextExtent(s) - calculates the size of the text in pixels;
TextArea(s, sz, pt) - calculates the size of the rectangle and the starting point for the given text and the current font (taking into account its orientation, i.e. rotation angle, and other properties) and the starting point for setting in text rendering methods;
TextWidth(s) - calculates the width of the text;
TextHeight(s) - calculates the height of the text;
ClipRect - returns the current bounding rectangle of the output area;
ModeCopy - the current copy mode (for the CopyRect method);
CopyRect(Rdst, srcDC, Rsrc) - copies a rectangle from another (or the same) canvas, possibly performing stretching / compressing or even flipping horizontally or vertically along the way, depending on the specified coordinates of the source and destination rectangles;
OnChange - an event that fires as soon as the content of the canvas changes;
Assign(srcCanvas) - assigns the content and parameters of the specified canvas to this canvas, including copying graphic tools;
RequiredState(i) - the method is mainly for internal use, ensures that the descriptors of the required graphic tools are ready for drawing, the input parameter can be a combination (OR combination) of the HandleValid, FontValid, BrushValid, PenValid and ChangingCanvas flags;
DeselectHandles - detaches all instruments from the canvas. It makes sense to use this method if you made direct changes to the font, brush or pencil parameters through API functions, and you need to ensure that the descriptors for these tools are recreated and attached to the canvas with the corrected descriptors before further drawing. This function is also mainly intended for internal use;
Pixels[X, Y] - slow access to canvas pixels (for fast pixel-by-pixel drawing on a bitmap in memory, it is recommended to use the Scanline [] property of the TBitmap object).
As you can see, the main set of canvas properties and methods is just as rich as in the VCL. In addition, there are a number of additions that allow you to work with Unicode text from a non-UNICODE application (in a UNICODE application, all the usual functions for working with text are automatically translated into their UNICODE compatible counterparts):
WTextOut(X, Y, s) - displays a Unicode string at the specified coordinates (analogous to TextOut);
WExtTextOut(X, Y, options, s, spacing) - similar to ExtTextOut, but for Unicode text;
WDrawText(s, R, flags) - analogue of DrawText for Unicode;
WTextRect(R, X, Y, s) - analogue of TextRec for Unicode;
WTextExtent(s) - similar to TextExtent, calculates the size of text in pixels;

© 2024 Carl Peeraer

WTextWidth(s) - the width of the Unicode text in pixels;
WTextHeight(s) - the height of the Unicode text in pixels.

And to wrap up my discussion of canvas and the tools for painting on canvas, here are the KOL functions for working with color. The color, just like in the VCL, is stored in a 32-bit integer variable (of the TColor type), in which a sign (less than zero) means that this is a system color constant corresponding to one of the system elements in the desktop setting, and the lower three bytes - in all other cases, the values of the red, green and blue color channels are stored in the usual color coding system R, G, B (R is the least significant byte, B is the most significant byte in the triplet).

There are a number of color conversion functions:

Color2RGB(C) - converts the system color to RGB-encoding (if the color is already specified by the RGB-code, then it is also returned as a result);
ColorsMix(C1, C2) - mixes two colors (arithmetic mean for each of the R, G, B channels), both colors are preliminarily converted to RGB;
Color2RGBQuad(C) - for a given color, returns a TRGBQuad structure (used in 32-bit bitmaps to store individual pixels, for example);
Color2Color16(C) - Returns the color as represented for a 16-bit color palette with 64K colors;
Color2Color15(C) - similar, but for a palette of 32K colors.

4.20.1 Elements of Graphics - Syntax

Constructors

```
function NewCanvas( DC: HDC ): PCanvas;
```
Use to construct Canvas on base of memory DC.

```
function NewFont: PGraphicTool;
```
151
Creates and returns font graphic tool object.

```
function NewBrush: PGraphicTool;
```
151
Creates and returns new brush object.

```
Function NewPen: PGraphicTool;
```
151
Creates and returns new pen object.

© 2024 Carl Peeraer

4.20.2 TCanvas - Syntax

TCanvas (unit KOL.pas) 🡐 TObj 92 🡐 _TObj 92
TCanvas = object(TObj 92)
Very similar to VCL's TCanvas object. But with some changes, specific for KOL: there is no necessary to use canvases in all applications. And graphic tools objects are not created with canvas, but only if really accessed in program. (Actually, even if paint box used, only programmer decides, if to implement painting using Canvas or to call low level API drawing functions working directly with DC). Therefore TCanvas has some powerful extensions: rotated text support, geometric pen support - just by changing correspondent properties of certain graphic tool objects (Font.FontOrientation, Pen.GeometricPen). See also additional Font 147 properties (Font.FontWeight, Font.FontQuality, etc.

type **PCanvas** = ^ TCanvas 221;

type **TFillStyle** =(fsSurface, fsBorder);
Available filling styles. For more info see Win32 or Delphi help files.

type **TFillMode** =(fmAlternate, fmWinding);
Available filling modes. For more info see Win32 or Delphi help files.

TCanvas properties

property **Handle**: HDC;
GDI device context object handle. Never created by Canvas 147 itself (to use Canvas with memory bitmaps, always create DC by yourself and assign it to the Handle property of Canvas 147 object, or use property Canvas 147 of a bitmap).

property **PenPos**: TPoint;
Position of a pen.

property **Pen**: PGraphicTool 151;
Pen of Canvas 147 object. Do not change its Pen. OnChange event value.

property **Brush**: PGraphicTool;
Brush of Canvas 147 object. Do not change its Brush.OnChange event value.

property **Font**: PGraphicTool;
Font of Canvas 147 object. Do not change its Font.OnChange event value.

property **ModeCopy**: TCopyMode;

© 2024 Carl Peeraer

Current copy mode. Is used in CopyRect 150 method.

```
property Pixels[ X, Y: Integer ]: TColor;
```
Obvious.

TCanvas methods

```
destructor Destroy; virtual;
```

```
procedure OffsetAndRotate( Xoff, Yoff: Integer; Angle: Double );
```
Transforms world coordinates so that Xoff and Yoff become the coordinates of the origin (0,0) and all further drawing is done rotated around that point by the Angle (which is given in radians)

```
procedure Arc( X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer ); stdcall;
```
Draws arc. For more info, see Delphi TCanvas help.

```
procedure Chord( X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer ); stdcall;
```
Draws chord. For more info, see Delphi TCanvas help.

```
procedure DrawFocusRect( const Rect: TRect );
```
Draws rectangle to represent focused visual object. For more info, see Delphi TCanvas help.

```
procedure Ellipse( X1, Y1, X2, Y2: Integer );
```
Draws an ellipse. For more info, see Delphi TCanvas help.

```
procedure FillRect( const Rect: TRect );
```
Fills rectangle. For more info, see Delphi TCanvas help.

```
procedure FillRgn( const Rgn: HRgn );
```
Fills region. For more info, see Delphi TCanvas help.

```
procedure FloodFill( X, Y: Integer; Color: TColor; FillStyle: TFillStyle 147 );
```
Fills a figure with givien color, floodfilling its surface. For more info, see Delphi TCanvas help.

```
procedure FrameRect( const Rect: TRect );
```
Draws a rectangle using Brush 147 settings (color, etc.). For more info, see Delphi TCanvas help.

```
procedure MoveTo( X, Y: Integer );
```
Moves current PenPos 147 to a new position. For more info, see Delphi TCanvas help.

© 2024 Carl Peeraer

`procedure` **`LineTo`**`( X, Y: Integer );`
Draws a line from current PenPos 147 up to new position. For more info, see Delphi TCanvas help.

`procedure` **`Pie`**`( X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer ); stdcall;`
Draws a pie. For more info, see Delphi TCanvas help.

`procedure` **`Polygon`**`( const Points: array of TPoint );`
Draws a polygon. For more info, see Delphi TCanvas help.

`procedure` **`Polyline`**`( const Points: array of TPoint );`
Draws a bound for polygon. For more info, see Delphi TCanvas help.

`procedure` **`Rectangle`**`( X1, Y1, X2, Y2: Integer );`
Draws a rectangle using current Pen 147 and/or Brush 147. For more info, see Delphi TCanvas help.

`procedure` **`RoundRect`**`( X1, Y1, X2, Y2, X3, Y3: Integer );`
Draws a rounded rectangle. For more info, see Delphi TCanvas help.

`procedure` **`TextOutA`**`( X, Y: Integer; const Text: AnsiString ); stdcall;`
Draws an ANSI text. For more info, see Delphi TCanvas help.

`procedure` **`TextOut`**`( X, Y: Integer; const Text: KOLString ); stdcall;`
Draws a text. For more info, see Delphi TCanvas help.

`procedure` **`ExtTextOut`**`( X, Y: Integer; Options: DWORD; const Rect: TRect; const Text: AnsiString; const Spacing: array of Integer );`

`procedure` **`TextRect`**`( const Rect: TRect; X, Y: Integer; const Text: Ansistring );`
Draws a text, clipping output into given rectangle. For more info, see Delphi TCanvas help.

`procedure` **`DrawText`**`( Text: AnsiString; var Rect: TRect; Flags: DWord );`

`function` **`TextExtent`**`( const Text: KOLString ): TSize;`
Calculates size of a Text, using current Font 147 settings. Does not need in Handle 147 for Canvas 147 object (if it is not yet allocated, temporary device context is created and used.

`procedure` **`TextArea`**`( const Text: KOLString; var Sz: TSize; var P0: TPoint );`
Calculates size and starting point to output Text, taking into considaration all Font 147 attributes,

© 2024 Carl Peeraer

including Orientation (only if GlobalGraphics_UseFontOrient flag is set to True, i.e. if rotated fonts are used). Like for TextExtent 149, does not need in Handle 147 (and if this last is not yet allocated/assigned, temporary device context is created and used).

```
procedure WTextArea( const Text: KOLWideString; var Sz: TSize; var P0: TPoint );
```
Calculates size and starting point to output Text, taking into considaration all Font 147 attributes, including Orientation (only if GlobalGraphics_UseFontOrient flag is set to True, i.e. if rotated fonts are used). Like for TextExtent 149, does not need in Handle 147 (and if this last is not yet allocated/assigned, temporary device context is created and used).

```
function TextWidth( const Text: KOLString ): Integer;
```
Calculates text width (using TextArea 149).

```
function TextHeight( const Text: KOLString ): Integer;
```
Calculates text height (using TextArea 149).

```
function ClipRect: TRect;
```
returns ClipBox. by Dmitry Zharov.

```
procedure WTextOut( X, Y: Integer; const WText: KOLWideString ); stdcall;
```
Draws a Unicode text.

```
procedure WExtTextOut( X, Y: Integer; Options: DWORD; const Rect: TRect; const
WText: KOLWideString; const Spacing: array of Integer );
```

```
procedure WDrawText( WText: KOLWideString; var Rect: TRect; Flags: DWord );
```

```
procedure WTextRect( const Rect: TRect; X, Y: Integer; const WText: KOLWideString );
```
Draws a Unicode text, clipping output into given rectangle.

```
function WTextExtent( const WText: KOLWideString ): TSize;
```
Calculates Unicode text width and height.

```
function WTextWidth( const WText: KOLWideString ): Integer;
```
Calculates Unicode text width.

```
function WTextHeight( const WText: KOLWideString ): Integer;
```
Calculates Unicode text height.

```
procedure CopyRect( const DstRect: TRect; SrcCanvas: PCanvas; const SrcRect:
TRect );
```


© 2024 Carl Peeraer

Copyes a rectangle from source to destination, using StretchBlt.

```
function Assign( SrcCanvas: PCanvas ): Boolean;
```

```
function RequiredState( ReqState: DWORD ): HDC; stdcall;
```

It is possible to call this method before using Handle 147 property to pass it into API calls - to provide valid combinations of pen, brush and font, selected into device context. This method can not provide valid Handle 147 - You always must create it by yourself and assign to TCanvas.Handle property manually. To optimize assembler version, returns Handle 147 value.

```
procedure DeselectHandles;
```

Call this method to deselect all graphic tool objects from the canvas 147.

TCanvas events

```
property OnGetHandle: TOnGetHandle;
```

For internal use only.

```
property OnChange: TOnEvent;
```

TCanvas fields

```
fIsPaintDC: Boolean;
```

TRUE, if DC obtained during current WM_PAINT (or WM_ERASEBKGND?) processing for a control. This affects a way how Handle 147 is released.

```
fIsAlienDC: Boolean;
```

TRUE if Canvas was created on base of existing DC, so DC is not beloning to the Canvas and should not be deleted when the Canvas object is destroyed.
Fields, inherited from TObj 92

4.20.3 TGraphicTool - Syntax

TGraphicTool (unit KOL.pas) ← TObj 92 ← _TObj 92
TGraphicTool = object(TObj 92)
Incapsulates all GDI objects: Pen, Brush and Font.

```
type PGraphicTool = ^ TGraphicTool;
```
151

```
type TBrushStyle =( bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal,
bsBDiagonal, bsCross, bsDiagCross );
```


© 2024 Carl Peeraer

Available brush styles.

```
type TPenStyle =( psSolid, psDash, psDot, psDashDot, psDashDotDot, psClear,
psInsideFrame );
```
Available pen styles. For more info see Delphi or Win32 help files.

```
type TPenMode =( pmBlack, pmNotMerge, pmMaskNotPen, pmNotCopy, pmMaskPenNot, pmNot,
pmXor, pmNotMask, pmMask, pmNotXor, pmNop, pmMergePenNot, pmCopy, pmMergeNotPen,
pmMerge, pmWhite );
```
Available pen modes. For more info see Delphi or Win32 help files.

```
type TPenEndCap =( pecRound, pecSquare, pecFlat );
```
Avalable (for geometric pen) end cap styles.

```
type TPenJoin =( pjRound, pjBevel, pjMiter );
```
Available (for geometric pen) join styles.

```
type TFontStyles =( fsBold, fsItalic, fsUnderline, fsStrikeOut );
```
Available font styles.

```
type TFontStyle = set of TFontStyles;
```
152
Font style is representing as a set of XFontStyles.

```
type TFontPitch =( fpDefault, fpFixed, fpVariable );
```
Availabe font pitch values.

```
type TFontName = type string;
```
Font name is represented as a string.

```
type TFontCharset = 0 . . 255;
```
Font charset is represented by number from 0 to 255.

```
type TFontQuality =( fqDefault, fqDraft, fqProof, fqNonAntialiased, fqAntialiased,
fqClearType );
```
Font quality.

TGraphicTool properties

```
property Handle: THandle;
```


© 2024 Carl Peeraer

Every time, when accessed, real GDI object is created (if it is not yet created). So, to prevent creating of the handle, use HandleAllocated 155 instead of comparing Handle with value 0.

`property **Color**: TColor;`
Color is the most common property for all Pen, Brush and Font objects, so it is placed in its common for all of them.

`property **BrushBitmap**: HBitmap;`
Brush bitmap. For more info about using brush bitmap, see Delphi or Win32 help files.

`property **BrushStyle**: TBrushStyle 151;`
Brush style.

`property **BrushLineColor**: TColor;`
Brush line color, used to represent lines in hatched brush. Default value is clBlack.

`property **FontHeight**: Integer;`
Font height. Value 0 (default) says to use system default value, negative values are to represent font height in "points", positive - in pixels. In XCL usually positive values (if not 0) are used to make appearance independent from different local settings.

`property **FontWidth**: Integer;`
Font width in logical units. If FontWidth = 0, then as it is said in Win32.hlp, "the aspect ratio of the device is matched against the digitization aspect ratio of the available fonts to find the closest match, determined by the absolute value of the difference."

`property **FontPitch**: TFontPitch 152;`
Font pitch. Change it very rare.

`property **FontStyle**: TFontStyle 152;`
Very useful property to control appearance.

`property **FontCharset**: TFontCharset 152;`
Do not change it if You do not know what You do.

`property **FontQuality**: TFontQuality 152;`
Font quality.

`property **FontOrientation**: Integer;`
It is possible to rotate text in KOL just by changing this property of a font (tenths of degree, i.e.

© 2024 Carl Peeraer

value 900 represents 90 degree - text written from bottom to top).

`property` **`FontWeight:`** `Integer;`
Additional font weight for bold fonts (must be 0..1000). When set to value <> 0, fsBold is added to FontStyle 153. And otherwise, when set to 0, fsBold is removed from FontStyle 153. Value 700 corresponds to Bold, 400 to Normal.

`property` **`FontName:`** `KOLString;`
Font face name.

`property` **`PenWidth:`** `Integer;`
Value 0 means default pen width.

`property` **`PenStyle:`** `TPenStyle` 152`;`
Pen style.

`property` **`PenMode:`** `TPenMode` 152`;`
Pen mode.

`property` **`GeometricPen:`** `Boolean;`
True if Pen is geometric. Note, that under Win95/98 only pen styles psSolid, psNull, psInsideFrame are supported by OS.

`property` **`PenBrushStyle:`** `TBrushStyle` 151`;`
Brush style for hatched geometric pen.

`property` **`PenBrushBitmap:`** `HBitmap;`
Brush bitmap for geometric pen (if assigned Pen is functioning as its style = BS_PATTERN, regadless of PenBrushStyle 154 value).

`property` **`PenEndCap:`** `TPenEndCap` 152`;`
Pen end cap mode - for GeometricPen 154 only.

`property` **`PenJoin:`** `TPenJoin` 152`;`
Pen join mode - for GeometricPen 154 only.

`property` **`LogFontStruct:`** `TLogFont;`
by Alex Pravdin: a property to change all font structure items at once.

© 2024 Carl Peeraer

TGraphicTool methods

```
procedure Changed;
```

```
function GetHandle: THandle;
```

```
destructor Destroy; virtual;
```

```
function HandleAllocated: Boolean;
```
Returns True, if handle is allocated (i.e., if real GDI objet is created.

```
function ReleaseHandle: Integer;
```
Returns Handle 152 value (if allocated), releasing it from the object (so, it is no more knows about this handle and its HandleAllocated 155 function returns False.

```
function Assign( Value: PGraphicTool ): PGraphicTool;
```
Assigns properties of the same (only) type graphic object, excluding Handle 152. If assigning is really leading to change object, procedure Changed 155 is called.

```
procedure AssignHandle( NewHandle: Integer );
```
Assigns value to Handle 152 property.

```
function IsFontTrueType: Boolean;
```
Returns True, if font is True Type. Requires of creating of a Handle 152, if it is not yet created.

TGraphicTool events

```
property OnChange: TOnGraphicChange;
```
Called, when object is changed.

4.20.4 Color Conversion - Syntax

```
function Color2RGB( Color: TColor ): TColor;
```
Function to get RGB color from system color. Parameter can be also RGB color, in that case result is just equal to a parameter.

```
function RGB2BGR( Color: TColor ): TColor;
```
Converts RGB color to BGR

```
function ColorsMix( Color1, Color2: TColor ): TColor;
```
Returns color, which RGB components are build as an (approximate) arithmetic mean of correspondent RGB components of both source colors (these both are first converted from

© 2024 Carl Peeraer

system to RGB, and result is always RGB color). Please note: this function is fast, but can be not too exact.

```
function Color2RGBQuad( Color: TColor ): TRGBQuad;
```
Converts color to RGB, used to represent RGB values in palette entries (actually swaps R and B bytes).

```
function Color2Color16( Color: TColor ): WORD;
```
Converts Color to RGB, packed to word (as it is used in format pf16bit).

```
function Color2Color15( Color: TColor ): WORD;
```
Converts Color to RGB, packed to word (as it is used in format pf15bit).

4.21 Image in Memory

Image in Memory - TBitmap

It is natural to continue the discussion of graphics with objects for representing images in RAM. The first and one of the most important of these is the TBitmap object type. It is intended for loading raster images from files (files with the .bmp extension), resources or other sources, for storing them in memory and modification, for drawing (on the canvas) and for saving - to files, streams. In the Windows environment, bitmaps (i.e., images that are stored in memory point by point, pixel by pixel, without any compression) play an important role due to their high processing speed. There are many API functions on Windows that are specifically designed to work with such images. The KOL library organizes a convenient object interface to these functions, enriching it with its extensions.

Bitmap object of type **TBitmap** is created by constructors
NewBitmap(W, H) - creates a "device-dependent" bitmap (DDB - Device Dependent Bitmap) of width W and height H pixels;
NewDIBBitmap(W, H, PixelFormat) - creates a "device independent" image (DIB - Device Independent Bitmap), specifying the pixel format (i.e., color depth, one of the values in the list: pf1bit, pf4bit, pf8bit, pf15bit, pf16bit , pf24bit, pf32bit). The pixel format affects how many different colors can be represented in an image and how much memory you have to allocate to store the image.

For pf1bit, pf4bit and pf8bit formats, images with a "palette" are created, i.e. a pixel value of 1, 4, or 8 bits is actually a corresponding bit number representing the color index of a point in the image's palette. The peculiarity of KOL when working with such images is that the palette is not assigned by default, and if you just create such an image and draw something on its canvas, then you will not see anything but a black square (I hope Malevich did not use KOL to become famous?). In order for the palette to appear in the image, you must either assign it with your

© 2024 Carl Peeraer

own code (**DIBPalEntries** property), or load the original image from some source, for example, from a file or from a resource.

With other formats, the situation is simpler: the pixel no longer stores the index in the palette, but the packed color R, G, B in a certain way (in this case, unlike the structure of the TColor variable, the B channel is stored in the low-order bits of the pixel, and R in the high-order ones) ... An object of type **TBitmap** has properties that allow you to directly access pixels in memory and modify them at will (ScanLine), this is the fastest possible way to modify images.

In any case, immediately after the image is created, it is a black rectangle of width W and height H pixels (since memory for the image is initialized with zeros). To fill the entire image with the desired color, it is convenient to use the canvas, and having assigned the desired brush color (Brush.Color: = clGreen, for example), call the canvas method **FillRect**.

4.21.1 The methods and properties of the TBitmap object

I will list the methods and properties of the TBitmap object (it has no events), grouping them into several main categories:

- Pixel descriptor and format 157
- Dimensions 158
- Loading and Saving 158
- Drawing an Image in a different Context 159
- Canvas and modification of your own image through it 159
- Direct access to pixels and image modification without canvas 160
- DIB image parameters 161

4.21.1.1 Pixel descriptor and format

HandleType - the type of the bitmap (**bmDDB or bmDIB**, depending on whether the image is device dependent or independent);
PixelFormat - pixel format (used for **HandleType = bmDIB, for bmDDB** type stores a special value **pfDevice**);
BitsPerPixel - calculates the number of bits required to represent one pixel (including for device-specific images);
Handle - descriptor of the system graphic object of the hBitmap type. Device independent images do not need such a descriptor, and in general, all work can potentially be done without allocating such a descriptor. However, if you are working with an image through a canvas, a descriptor for the image is created automatically. It is allowed to assign this property as a value a descriptor of a bitmap (of the hBitmap type) obtained in any way, including from API functions - in this case, the old image is lost and replaced with the assigned one, and the object becomes the "owner" of the assigned descriptor (that is, the descriptor will be automatically destroyed along with the object in its destructor);
HandleAllocated - checks if the handle has been allocated;
ReleaseHandle - separates the descriptor from the image, freeing it (the device-independent image continues to exist in memory, and if it is necessary to perform any operations requiring

© 2024 Carl Peeraer

the presence of the descriptor, it will be allocated again). The descriptor that was detached as a result of such an operation is freed in the sense that it is not known (or of interest) to the **TBitmap** object from that point on. And then the calling code is responsible for its further fate. For example, it can be deleted by the **DeleteObject** API function, or used in some other way. It is only important to ensure that there are no leaks of such resources: all allocated GDI resources, which include hBitmap, must be removed when they are no longer needed;
Dormant - detaches the handle from the image, and destroys it (which is equivalent to calling **DeleteObject** (ReleaseHandle)), and also releases the canvas of the object (RemoveCanvas). This method is recommended to prevent the application from allocating too many GDI resources at the same time if you need to work alternately with a large number of bitmaps in memory;
Assign(srcBmp) - assigns the specified image to the given image, copying it physically (unlike VCL, where copying is postponed until the content of one of the images changes);

4.21.1.2 Dimensions

Width - width in pixels (must be greater than zero, otherwise the image is considered empty);
Height - height in pixels (similar to Width, must be greater than zero);
The width and height of the image for the **TBitmap** object can be changed dynamically, while the previous image is saved (copied from the old one), when the size is increased, the new space is filled with the color that is set for the brush of its canvas, and when the image is reduced, it is cropped to the new size;
BoundsRect - returns a rectangle with coordinates (0, 0, Width, Height). This function is convenient to use for passing parameters to those methods where a rectangle is required: obviously, notation of the form

```
Bmp.Canvas.FillRect (Bmp.BoundsRect);
```

both shorter and clearer than the equivalent construction

```
Bmp.Canvas.FillRect (MakeRect (0, 0, Bmp.Width, Bmp.Height));
```

Empty - checks if the image is empty (the image is empty if its width or height is zero);
Clear - makes the image empty, freeing the resources occupied by the image;

4.21.1.3 Loading and Saving

LoadFromFile(s)- loads an image from a BMP file. This method cannot download compressed (RLE-encoded) images;
LoadFromFileEx(s) - loads an image from a BMP file, similar to the previous method, but understands, among other things, loading RLE-encoded images;
SaveToFile(s) - saves the image to a file in BMP format;
LoadFromStream(strm)- loads an image from the stream (from the current position in the stream, and to the end of the image). Loading of RLE-encoded images by this method is not performed;
LoadFromStreamEx(strm) - the same as the previous method, but also loads RLE-encoded images;
SaveToStream(strm) - writes an image to a stream in BMP format;

© 2024 Carl Peeraer

LoadFromResourceID(inst, resID) - loads an image from an application resource or another executable file (as determined by the inst parameter), by the numeric resource identifier resID;
LoadFromResourceName(inst, s) - loads an image from a resource by resource name;
CopyToClipboard - copies the image to the Windows clipboard;
PasteFromClipboard - pastes an image from the clipboard.

4.21.1.4 Drawing an Image in a different Context

Draw(DC, X, Y) - draws its image on the specified DC context (Device Context of hDC type) from the specified coordinate (X, Y), without changing the scale;
StretchDraw(DC, R) - draws its image on the specified DC device context, fitting it (scaling) into the rectangle R. Additional information: in order for the image scaling to be smooth, it is necessary for the DC to use the API function SetStretchBltMode (DC, halftone);
DrawTransparent(DC, X, Y, C) - draws its image on the DC context similarly to the Draw method, but the C color is considered "transparent" and skipped (that is, the previous image on the device remains untouched at the corresponding points);
StretchDrawTransparent(DC, X, Y, C) - similar to StretchDraw, but assuming the color C is "transparent";
DrawMasked(DC, X, Y, maskBmp) - another version of transparent drawing, in which the maskBmp parameter of type hBitmap is used as a mask (black color in the mask corresponds to transparent areas that will not fall into the target context). This method is faster, and allows you to optimize performance for multiple rendering of the same image, if the mask is prepared in advance;
StretchDrawMasked(DC, R, maskBmp) - similar to the previous one, but after applying the mask, the image is scaled to fit the rectangle R before being displayed on the target context;
Convert2Mask(C) - converts the image into a monochrome mask, assuming C to be a transparent color (the corresponding areas become black, all other pixels in the mask - white);

Note that the approach taken in KOL is completely contrary to what is done in this regard in the VCL. But this is not in KOL everything is done upside down, but in the VCL, everything is inside out. There the canvas "draws" graphical objects like TBitmap, ie. it must "know" in advance, at least, about the existence of some abstract progenitor TPicture for all such objects, and refer to its virtual (and in fact, abstract Draw method) to perform this operation. In KOL, neither an abstract method nor a fictitious progenitor is needed for all "drawn" objects, and in general, to draw a graphic object on a canvas, it is not at all necessary to have this canvas encapsulated into an object - it is enough to have a DC descriptor, i.e. just a 32-bit value.

4.21.1.5 Canvas and modification of your own image through it

Canvas - the canvas of the image object itself, allows you to use all the capabilities of the canvas object to draw on the image in memory. Drawing in memory is usually faster than directly on the window context, and besides, if you draw on a window, the user will be able to observe the drawing process itself, or at least flicker will be observed when the window is redrawn. In-memory images are often used to prevent such flickering: the image is prepared on the TBitmap object in memory, then quickly copied into the window context, for example, using the Draw method of the TBitmap object;

© 2024 Carl Peeraer

RemoveCanvas - destroys the canvas object, freeing all resources and tools with it (the image itself is not affected, because the canvas is a temporary object attached to the image when it is necessary to draw on the image);
BkColor - background color. Generally, it syncs with the canvas brush color, if available, but can work to fill the background even when the canvas is not being used;
Pixels[X, Y] - slow access to image pixels, requires creating a canvas and allocating a handle for the image;
CopyRect(dstR, srcBmp, srcR) - copies the image or part of it from the specified TBitmap object using the canvas;
Invert - inverts the image;

4.21.1.6 Direct access to pixels and image modification without canvas

ScanLine[Y] - direct access to a row of pixels with a Y coordinate, allows you to quickly perform pixel-by-pixel processing of DIB images through pointers in memory (not applicable to DDB images, that is, device-dependent). When working with this property, it is necessary to take into account the size of pixels in bits (for example, for the pf1bit format, 8 pixels are located in one byte, and for the pf24bit format, three bytes are used to store one pixel). If you need to further increase the speed of accessing pixel lines, remember that you cannot simply add the resulting pointer for the top line ScanLine [0] with the size of the ScanLineSize pixel line to get the beginning of the next pixel line. The fact is that in Windows, DIB images in memory are stored upside down: first comes the bottom line, then the penultimate line, and at the very end - the top one.
ScanLineSize - returns the size of a pixel line in bytes (taking into account alignment to a double word, i.e. in Windows for an image of any size and any pixel format, the length of a pixel line must contain an integer of 4-byte words);
DibPixels[X, Y] - this property provides a slightly faster way to access image pixels than Pixels, and does not require using a canvas and creating a descriptor (but this method is still slower than through ScanLine, and especially slow for pf15bit and pf16bit formats, for which you need to convert the color of the pixel to TColor and vice versa);
DibDrawRect(DC, X, Y, R) - allows you to draw a DIB image (or part of it enclosed by rectangle R) on the target context without having to create a Handle for this operation or attach a canvas. Drawing directly is no slower than the Draw operation (and may even be faster if the device format matches the image format, so no pixel format conversion will be performed during the drawing process);

RotateRight - rotates the DIB image clockwise by 90 degrees, as a result, the width becomes the same height, and the height - the same height. This operation, like all other rotations, is performed via direct pixel access (ScanLine), and is very fast;
RotateLeft - rotates the image 90 degrees counterclockwise. There are some more methods for rotating an image, but for a specific pixel format (**RotateRightMono**, **RotateLeftMono**, **RotateRight4bit**, **RotateLeft4bit**, **RotateRight8bit**, **RotateLeft8bit**, **RotateRight16bit**, **RotateLeft16bit**, **RotateRightTrueColor**, **RotateLeftTrueColor**);
FlipVertical - quickly flips the DIB image vertically;
FlipHorizontal - quickly flips the DIB image horizontally;

© 2024 Carl Peeraer

4.21.1.7 DIB image parameters

DibPalEntryCount - returns the number of colors in the palette, depending on the bit depth of the pixel (only for formats that have a palette: pf1bit, pf4bit, pf8bit);
DibPalEntries[i] - access to the colors of the palette by index, allows you to change the palette for DIB images (formats 1, 4 or 8 bits per pixel);
DibPalNearestEntry(C) - finds in the palette the index of the color closest to the color specified by the C parameter;
DibBits - the pointer to the memory that stores the pixels of the images is intended for internal use (can be used by professionals for their own purposes);
DibSize - the size of the memory array for storing pixels;
DibHeader - access to the internal header of the DIB-image, like the two previous properties, is intended mainly for internal use;

The KOL library contains a number of additional functions for working with bitmaps. For example, the following set of functions allows you to load an image from resources, modifying it in such a way that its standard colors are automatically adjusted to the current settings of the desktop system colors:

LoadMappedBitmap(Inst, ResID, Map) - loads an image from a resource (by the numeric resource identifier), replacing colors along the way in accordance with those specified in the Map array. Returns a handle to the loaded bitmap of type hBitmap;
LoadMappedBitmapEx(MasterObj, Inst, ResName, Map)- similar to the previous one, but loads by resource name and understands any pixel formats in resources. In addition, if the MasterObj object is specified, then it becomes the owner of the loaded descriptor, and when this object is destroyed, the loaded descriptor will also be automatically destroyed;
CreateMappedBitmap(Inst, Bmp, Flags, ColorMap, i) - performs color replacement for an existing bitmap;
CreateMappedBitmapEx(Inst, ResName, Flags, ColorMap, i) - the same as the previous function, but the image is first loaded from the resource by name;
LoadBmp(Instance: Integer; Rsrc: PChar; MasterObj: PObj) - loads an image from a resource, adding it to the list of objects for deletion along with the MasterObj object;

4.21.2 Image in Memory - Syntax

TBitmap(unit KOL.pas) ← TObj 92 ← _TObj 92
TBitmap = object(TObj 92)
Bitmap incapsulation object.

type **PCanvas** = ^ TCanvas 161;

type **TPixelFormat** =(pfDevice, pf1bit, pf4bit, pf8bit, pf15bit, pf16bit, pf24bit, pf32bit, pfCustom);
Available pixel formats.

© 2024 Carl Peeraer

```
type TBitmapHandleType =( bmDIB, bmDDB );
```
Available bitmap handle types.

```
function DesktopPixelFormat: TPixelFormat;
```
Returns the pixel format correspondent to current desktop color resolution. Use this function to decide which format to use for converting bitmap, planned to draw transparently using TBitmap.DrawTransparent 166 or TBitmap.StretchDrawTransparent 166 methods.

```
function NewBitmap( W, H: Integer ): PBitmap;
```
Creates bitmap object of given size. If it is possible, do not change its size (Width and Heigth) later - this can economy code a bit. See TBitmap 161.

```
function NewDIBBitmap( W, H: Integer; PixelFormat: TPixelFormat ): PBitmap;
```
Creates DIB bitmap object of given size and pixel format. If it is possible, do not change its size (Width and Heigth) later - this can economy code a bit. See TBitmap 161.

TBitmap properties

```
property Width: Integer;
```
Width of bitmap. To make code smaller, avoid changing Width or Height 162 after bitmap is created (using NewBitmap 162) or after it is loaded from file, stream of resource.

```
property Height: Integer;
```
Height of bitmap. To make code smaller, avoid changing Width 162 or Height after bitmap is created (using NewBitmap 162) or after it is loaded from file, stream of resource.

```
property BoundsRect: TRect;
```
Returns rectangle (0,0,Width 162,Height 162).

```
property Empty: Boolean;
```
Returns True if Width 162 or Height 162 is 0.

```
property Handle: HBitmap;
```
Handle of bitmap. Created whenever property accessed. To check if handle is allocated (without allocating it), use HandleAllocated 162 property.

```
property HandleAllocated: Boolean;
```
Returns True, if Handle 162 already allocated.

```
property HandleType: TBitmapHandleType;
```
162

© 2024 Carl Peeraer

bmDIB, if DIB part of image data is filled and stored internally in TBitmap object. DIB image therefore can have Handle 162 allocated, which require resources. Use HandleAllocated 162 funtion to determine if handle is allocated and Dormant 166 method to remove it, if You want to economy GDI resources. (Actually Handle 162 needed for DIB bitmap only in case when Canvas 163 is used to draw on bitmap surface). Please note also, that before saving bitmap to file or stream, it is converted to DIB.

`property` **`PixelFormat:`** `TPixelFormat 161;`
Current pixel format. If format of bitmap is unknown, or bitmap is DDB, value is pfDevice. Setting PixelFormat to any other format converts bitmap to DIB, back to pfDevice converts bitmap to DDB again. Avoid such conversations for large bitmaps or for numerous bitmaps in your application to keep good performance
.

`property` **`Canvas:`** `PCanvas 161;`
Canvas can be used to draw onto bitmap. Whenever it is accessed, handle is allocated for bitmap, if it is not yet (to make it possible to select bitmap to display compatible device context).

`property` **`BkColor:`** `TColor;`
Used to fill background for Bitmap, when its width or height is increased. Although this value always synchronized with Canvas.Brush.Color, use it instead if You do not use Canvas 163 for drawing on bitmap surface.

`property` **`Pixels`**`[ X, Y: Integer ]: TColor;`
Allows to obtain or change certain pixels of a bitmap. This method is both for DIB and DDB bitmaps, and leads to allocate handle anyway. For DIB bitmaps, it is possible to use property DIBPixels 163[ ] instead, which is much faster and does not require in Handle 162.

`property` **`ScanLineSize:`** `Integer;`
Returns size of scan line in bytes. Use it to measure size of a single ScanLine 163. To calculate increment value from first byte of ScanLine 163 to first byte of next ScanLine 163, use difference
Integer(ScanLine 163[1]-ScanLine 163[0])
(this is because bitmap can be oriented from bottom to top, so step can be negative).

`property` **`ScanLine`**`[ Y: Integer ]: Pointer;`
Use ScanLine to access DIB bitmap pixels in memory to direct access it fast. Take in attention, that for different pixel formats, different bit counts are used to represent bitmap pixels. Also do not forget, that for formats pf4bit and pf8bit, pixels actually are indices to palette entries, and for formats pf16bit, pf24bit and pf32bit are actually RGB values (for pf16bit B:5-G:6-R:5, for pf15bit B:5-G:5-R:5 (high order bit not used), for pf24bit B:8-G:8-R:8, and for pf32bit high order byte of TRGBQuad structure is not used).

`property` **`DIBPixels`**`[ X, Y: Integer ]: TColor;`

© 2024 Carl Peeraer

Allows direct access to pixels of DIB bitmap, faster then Pixels 163 [ ] property. Access to read is slower for pf15bit, pf16bit formats (because some conversation needed to translate packed RGB color to TColor). And for write, operation performed most slower for pf4bit, pf8bit (searching nearest color required) and fastest for pf24bit, pf32bit and pf1bit.

`property` **DIBPalEntryCount**`: Integer;`
Returns palette entries count for DIB image. Always returns 2 for pf1bit, 16 for pf4bit, 256 for pf8bit and 0 for other pixel formats.

`property` **DIBPalEntries**`[ Idx: Integer ]: TColor;`
Provides direct access to DIB palette.

`property` **DIBBits**`: Pointer;`
This property is mainly for internal use.

`property` **DIBSize**`: Integer;`
Size of DIBBits 164 array.

`property` **DIBHeader**`: PBitmapInfo;`

TBitmap methods

`procedure` **Clear**`;`
Makes bitmap empty, setting its Width 162 and Height 162 to 0.

`procedure` **LoadFromFile**`( const Filename: KOLString );`
Loads bitmap from file (LoadFromStream 165 used).

`function` **LoadFromFileEx**`( const Filename: KOLString ): Boolean;`
Loads bitmap from a file. If necessary, bitmap is RLE-decoded. Code given by Vyacheslav A. Gavrik.

`procedure` **SaveToFile**`( const Filename: KOLString );`
Stores bitmap to file (SaveToStream 165 used).

`procedure` **CoreSaveToFile**`( const Filename: KOLString );`
Stores bitmap to file (CoreSaveToStream 165 used).

`procedure` **RLESaveToFile**`( const Filename: KOLString );`
Stores bitmap to file (CoreSaveToStream 165 used).

© 2024 Carl Peeraer

```
procedure LoadFromStream( Strm: PStream );
```
Loads bitmap from stream. Follow loading, bitmap has DIB format (without handle allocated). It is possible to draw DIB bitmap without creating handle for it, which can economy GDI resources.

```
function LoadFromStreamEx( Strm: PStream ): Boolean;
```
Loads bitmap from a stream. Difference is that RLE decoding supported. Code given by Vyacheslav A. Gavrik.

```
procedure SaveToStream( Strm: PStream );
```
Saves bitmap to stream. If bitmap is not DIB, it is converted to DIB before saving.

```
procedure CoreSaveToStream( Strm: PStream );
```
Saves bitmap to stream using CORE format with RGBTRIPLE palette and with BITMAPCOREHEADER as a header. If bitmap is not DIB, it is converted to DIB before saving.

```
procedure RLESaveToStream( Strm: PStream );
```
Saves bitmap to stream using CORE format with RGBTRIPLE palette and with BITMAPCOREHEADER as a header. If bitmap is not DIB, it is converted to DIB before saving.

```
procedure LoadFromResourceID( Inst: DWORD; ResID: Integer );
```
Loads bitmap from resource using integer ID of resource. To load by name, use LoadFromResurceName. To load resource of application itself, pass hInstance as first parameter. This method also can be used to load system predefined bitmaps, if 0 is passed as Inst parameter:

OBM_BTNCORNERS	OBM_REDUCE
OBM_BTSIZE	OBM_REDUCED
OBM_CHECK	OBM_RESTORE
OBM_CHECKBOXES	OBM_RESTORED
OBM_CLOSE	OBM_RGARROW
OBM_COMBO	OBM_RGARROWD
OBM_DNARROW	OBM_RGARROWI
OBM_DNARROWD	OBM_SIZE
OBM_DNARROWI	OBM_UPARROW
OBM_LFARROW	OBM_UPARROWD
OBM_LFARROWD	OBM_UPARROWI
OBM_LFARROWI	OBM_ZOOM
OBM_MNARROW	OBM_ZOOMD

© 2024 Carl Peeraer

`procedure` **`LoadFromResourceName`**`( Inst: DWORD; ResName: PKOLChar );`
Loads bitmap from resurce (using passed name of bitmap resource.

`function` **`Assign`**`( SrcBmp: PBitmap ): Boolean;`
Assigns bitmap from another. Returns False if not success. Note: remember, that Canvas 163 is not assigned - only bitmap image is copied. And for DIB, handle is not allocating due this process.

`function` **`ReleaseHandle`**`: HBitmap;`
Returns Handle 162 and releases it, so bitmap no more know about handle. This method does not destroy bitmap image, but converts it into DIB. Returned Handle 162 actually is a handle of copy of original bitmap. If You need not in keping it up, use Dormant 166 method instead.

`procedure` **`Dormant`**`;`
Releases handle from bitmap and destroys it. But image is not destroyed and its data are preserved in DIB format. Please note, that in KOL, DIB bitmaps can be drawn onto given device context without allocating of handle. So, it is very useful to call Dormant preparing it using Canvas 163 drawing operations - to economy GDI resources.

`function` **`BitsPerPixel`**`: Integer;`
Returns bits per pixel if possible.

`procedure` **`Draw`**`( DC: HDC; X, Y: Integer );`
Draws bitmap to given device context. If bitmap is DIB, it is always drawing using SetDIBitsToDevice API call, which does not require bitmap handle (so, it is very sensible to call Dormant 166 method to free correspondent GDI resources).

`procedure` **`StretchDraw`**`( DC: HDC; const Rect: TRect );`
Draws bitmap onto DC, stretching it to fit given rectangle Rect.

`procedure` **`DrawTransparent`**`( DC: HDC; X, Y: Integer; TranspColor: TColor );`
Draws bitmap onto DC transparently, using TranspColor as transparent. See function PixelFormat 163 also.

`procedure` **`StretchDrawTransparent`**`( DC: HDC; const Rect: TRect; TranspColor: TColor );`
Draws bitmap onto given rectangle of destination DC (with stretching it to fit Rect) - transparently, using TranspColor as transparent. See function DesktopPixelFormat 162 also.

`procedure` **`DrawMasked`**`( DC: HDC; X, Y: Integer; Mask: HBitmap );`
Draws bitmap to destination DC transparently by mask. It is possible to pass as a mask handle of another TBitmap, previously converted to monochrome mask using Convert2Mask 167 method.

© 2024 Carl Peeraer

`procedure` **`StretchDrawMasked`**`( DC: HDC; const Rect: TRect; Mask: HBitmap );`
Like DrawMasked 166, but with stretching image onto given rectangle.

`procedure` **`Convert2Mask`**`( TranspColor: TColor );`
Converts bitmap to monochrome (mask) bitmap with TranspColor replaced to clBlack and all other ones to clWhite. Such mask bitmap can be used to draw original bitmap transparently, with given TranspColor as transparent. (To preserve original bitmap, create new instance of TBitmap and assign original bitmap to it). See also DrawTransparent 166 and StretchDrawTransparent 166 methods.

`procedure` **`Invert;`**
Obvious.

`procedure` **`RemoveCanvas;`**
Call this method to destroy Canvas 163 and free GDI resources.

`function` **`DIBPalNearestEntry`**`( Color: TColor ): Integer;`
Returns index of entry in DIB palette with color nearest (or matching) to given one.

`procedure` **`DIBDrawRect`**`( DC: HDC; X, Y: Integer; const R: TRect );`
This procedure copies given rectangle to the target device context, but only for DIB bitmap (using SetDIBBitsToDevice API call).

`procedure` **`RotateRight;`**
Rotates bitmap right (90 degree). Bitmap must be DIB. If You definitevely know format of a bitmap, use instead one of methods RotateRightMono 167, RotateRight4bit 168, RotateRight8bit 168, RotateRight16bit 168 or RotateRightTrueColor 168 - this will economy code. But if for most of formats such methods are called, this can be more economy just to call always universal method RotateRight.

`procedure` **`RotateLeft;`**
Rotates bitmap left (90 degree). Bitmap must be DIB. If You definitevely know format of a bitmap, use instead one of methods RotateLeftMono 167, RotateLeft4bit 168, RotateLeft8bit 168, RotateLeft16bit 168 or RotateLeftTrueColor 168 - this will economy code. But if for most of formats such methods are called, this can be more economy just to call always universal method RotateLeft.

`procedure` **`RotateRightMono;`**
Rotates bitmap right, but only if bitmap is monochrome (pf1bit).

© 2024 Carl Peeraer

`procedure` **RotateLeftMono;**
Rotates bitmap left, but only if bitmap is monochrome (pf1bit).

`procedure` **RotateRight4bit;**
Rotates bitmap right, but only if PixelFormat 163 is pf4bit.

`procedure` **RotateLeft4bit;**
Rotates bitmap left, but only if PixelFormat 163 is pf4bit.

`procedure` **RotateRight8bit;**
Rotates bitmap right, but only if PixelFormat 163 is pf8bit.

`procedure` **RotateLeft8bit;**
Rotates bitmap left, but only if PixelFormat 163 is pf8bit.

`procedure` **RotateRight16bit;**
Rotates bitmap right, but only if PixelFormat 163 is pf16bit.

`procedure` **RotateLeft16bit;**
Rotates bitmap left, but only if PixelFormat 163 is pf16bit.

`procedure` **RotateRightTrueColor;**
Rotates bitmap right, but only if PixelFormat 163 is pf24bit or pf32bit.

`procedure` **RotateLeftTrueColor;**
Rotates bitmap left, but only if PixelFormat 163 is pf24bit or pf32bit.

`procedure` **FlipVertical;**
Flips bitmap vertically

`procedure` **FlipHorizontal;**
Flips bitmap horizontally

`procedure` **CopyRect**`( const DstRect: TRect; SrcBmp: PBitmap; const SrcRect: TRect );`
It is possible to use Canvas.CopyRect for such purpose, but if You do not want use TCanvas 161, it is possible to copy rectangle from one bitmap to another using this function.

`function` **CopyToClipboardAsDIB**`: Boolean;`
Copies bitmap to clipboard, converting it to DIB format first.

© 2024 Carl Peeraer

```
function CopyToClipboard: Boolean;
```
Copies bitmap to clipboard. When Handle 162 = 0, CLIPBOARD is emptied!!!

```
function PasteFromClipboard: Boolean;
```
Takes CF_DIB format bitmap from clipboard and assigns it to the TBitmap object.

```
function LoadMappedBitmap( hInst: THandle; BmpResID: Integer; const Map: array of
TColor ): HBitmap;
```
This function can be used to load bitmap and replace some it colors to desired ones. This function especially useful when loaded by the such way bitmap is used as toolbar bitmap - to replace some original colors to system default colors. To use this function properly, the bitmap should be prepared as 16-color bitmap, which uses only system colors. To do so, create a new 16-color bitmap with needed dimensions in Borland Image Editor and paste a bitmap image, copied in another graphic tool, and then save it. If this is not done, bitmap will not be loaded correctly!

```
function LoadMappedBitmapEx( MasterObj: PObj 92; hInst: THandle; BmpResName:
PKOLChar; const Map: array of TColor ): HBitmap;
```
by Alex Pravdin: like LoadMappedBitmap 169, but much powerful. It uses CreateMappedBitmapEx 169, so it understands any bitmap color format, including pf24bit. Also, LoadMappedBitmapEx provides auto-destroying loaded resource when MasterObj is destroyed.

```
function CreateMappedBitmap( Instance: THandle; Bitmap: Integer; Flags: UINT;
ColorMap: PColorMap; NumMaps: Integer ): HBitmap; stdcall;
```
Creates mapped bitmap replacing colors correspondently to the ColorMap (each pare of colors defines color replaced and a color used for replace it in the bitmap). See also CreateMappedBitmapEx 169.

```
function CreateMappedBitmapEx( Instance: THandle; BmpRsrcName: PKOLChar; Flags:
Cardinal; ColorMap: PColorMap; NumMaps: Integer ): HBitmap;
```


© 2024 Carl Peeraer

4.22 Pictogram

Pictogram - TIcon

Another no less important graphic object is a pictogram (in computer slang - "icon"). Icons in Windows applications are used as icons to identify windows, to distinguish between different buttons on toolbars, and for an immeasurable number of purposes. You can also work with icons through API functions, but in many cases it is convenient to use the TIcon object for this. Constructor:

NewIcon - creates an empty **TIcon** object and returns a pointer to it of the PIcon type;

Basic methods and properties of the icon object:

Handle - GDI handle to the icon. For a non-empty object, the icon is always not 0. To assign a descriptor of type hIcon or hCursor to an object, it must be assigned to this property;
ShareIcon - determines whether the descriptor is "shared": the shared resource of the icon is not considered to belong to the object, and is not destroyed when the object is destroyed;
Empty - checks that the icon is empty (that is, it is not loaded into the object, and the descriptor is 0);
Clear - clears the object (makes it empty), while if there was a descriptor and it was not shared (ShareIcon), then the descriptor is destroyed, freeing the corresponding GDI resource in the system;
Size - for square icons, shows their size (height and width).
Before loading an icon from external sources (file, resource), this value can be assigned a nonzero value so that when loading an icon that has several image options, the image of the specified size is loaded (by default, the 32x32 icon is always loaded first, and secondly - as close as possible to her in size);
Width - the width of the icon (in order for the width and height to differ, the **ICON_DIFF_WH** conditional compilation symbol should be included in the project option);
Height - the height of the pictogram (the remark about the symbol of conditional compilation is also true for the height);
HotSpot - the coordinates of the point, which is used to store the "touch point" for cursors (hCursor can also be stored and managed by the TIcon object, since in fact they are no different);
Draw(DC, X, Y) - Draws an icon image on the specified DC context. Drawing a transparent pictogram is always done in a "transparent" manner; Areas corresponding to transparent areas on the target canvas are not affected.
StretchDraw(DC, R) - draws an icon with scaling, fitting it into the specified rectangle;
LoadFromFile(s) - loads an icon from a file;
LoadFromStream(strm) - loads an icon from a data stream;
SaveToFile(s) - saves the icon in the specified file;
SaveToStream(strm) - saves the icon in the data stream;
LoadFromResourceID(inst, resID, sz) - loads an icon from a resource by a numeric resource identifier;

© 2024 Carl Peeraer

LoadFromResourceName(inst, resName, sz) - loads an icon from a resource by resource name;
LoadFromExecutable(s, i) - loads an icon from the specified executable file (.exe, .dll, etc.), according to the resource number of icons in this file (the global function GetFileIconCount (s) returns the number of resources of icons in the specified executable file);
ConvertToBitmap - creates an hBitmap image based on the icon and returns it;

In addition, there are a number of **global functions** that allow you to work (load and save) groups of icons as one icon with several image options:

SaveIcons2Stream(icons, strm) - saves the icons specified in the icons array to the specified stream as a single icon with several image options. The array of icons should contain several icons of different sizes;
SaveIcons2File(icons, s) - similar to the previous one, but the multiple icon resource is saved in the file;
And one more global function for loading an icon:
LoadImgIcon(resName, sz) - loads an icon from the resource of the application itself by the name of the resource, as close as possible to the specified size (if 0, then the 32x32 icon is loaded by default);

4.22.1 Pictogram - Syntax

TIcon (unit KOL.pas) ← TObj 92 ← _TObj 92
TIcon = object(TObj 92)
Object type to incapsulate icon or cursor image.

function **NewIcon**: PIcon;
Creates new icon object, setting its Size to 32 by default. Created icon is Empty.

TIcon properties

property **Size**: Integer;
Icon dimension (width and/or height, which are equal to each other always).

property **Handle**: HIcon;
Windows icon object handle.

property **Empty**: Boolean;
Returns True if icon is Empty.

property **ShareIcon**: Boolean;

© 2024 Carl Peeraer

True, if icon object is shared and can not be deleted when TIcon object is destroyed (set this flag is to True, if an icon is obtained from another TIcon object, for example).

`property` **`HotSpot`**`: TPoint;`
Hot spot point - for cursors.

TIcon methods

`procedure` **`SetHandleEx`**`( NewHandle: HIcon );`
Set Handle 171 without changing Size 171 (Width/Height).

`procedure` **`Clear`**`;`
Clears icon, freeing image and allocated GDI resource (Handle 171).

`procedure` **`Draw`**`( DC: HDC; X, Y: Integer );`
Draws icon onto given device context. Icon always is drawn transparently using its transparency mask (stored internally in icon object).

`procedure` **`StretchDraw`**`( DC: HDC; Dest: TRect );`
Draws icon onto given device context with stretching it to fit destination rectangle. See also Draw 172.

`procedure` **`LoadFromStream`**`( Strm: PStream );`
Loads icon from stream. If stream contains several icons (of different dimentions), icon with the most appropriate size is loading.

`procedure` **`LoadFromFile`**`( const FileName: KOLString );`
Load icon from file. If file contains several icons (of different dimensions), icon with the most appropriate size is loading.

`procedure` **`LoadFromResourceID`**`( Inst: Integer; ResID: Integer; DesiredSize: Integer );`
Loads icon from resource. To load system default icon, pass 0 as Inst and one of followin values as ResID:

IDI_APPLICATION	Default application icon.
IDI_ASTERISK	Asterisk (used in informative messages).
IDI_EXCLAMATION	Exclamation point (used in warning messages).
IDI_HAND	Hand-shaped icon (used in serious warning messages).

© 2024 Carl Peeraer

IDI_QUESTION	Question mark (used in prompting messages).
IDI_WINLOGO	Windows logo.

It is also possible to load icon from resources of another module, if pass instance handle of loaded module as Inst parameter.

```
procedure LoadFromResourceName( Inst: Integer; ResName: PKOLChar; DesiredSize: Integer );
```

Loads icon from resource. To load own application resource, pass hInstance as Inst parameter. It is possible to load resource from another module, if pass its instance handle as Inst.

```
procedure LoadFromExecutable( const FileName: KOLString; IconIdx: Integer );
```

Loads icon from executable (exe or dll file). Always default sized icon is loaded. It is possible also to get know how much icons are contained in executable using gloabl function GetFileIconCount[174]. To obtain icon of another size, try to load given executable and use LoadFromResourceID[172] method.

```
procedure SaveToStream( Strm: PStream );
```

Saves single icon to stream. To save icons with several different dimensions, use global procedure SaveIcons2Stream[173].

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

Saves single icon to file. To save icons with several different dimensions, use global procedure SaveIcons2File[174].

```
function Convert2Bitmap( TranColor: TColor ): HBitmap;
```

Converts icon to bitmap, returning Windows GDI bitmap resource as a result. It is possible later to assign returned bitmap handle to Handle[171] property of **TBitmap**[161] object to use features of **TBitmap**[161]. Pass TranColor to replace transparent area of icon with given color.

Global Functions

```
procedure SaveIcons2Stream( const Icons: array of PIcon; Strm: PStream );
```


© 2024 Carl Peeraer

Saves several icons (of different dimentions) to stream.

`function SaveIcons2StreamEx( const BmpHandles: array of HBitmap; Strm: PStream ): Boolean;`
Saves icons creating it from pairs of bitmaps and their masks. BmpHandles array must contain pairs of bitmap handles, each pair of color bitmap and mask bitmap of the same size.

`procedure SaveIcons2File( const Icons: array of PIcon; const FileName: KOLString );`
Saves several icons (of different dimentions) to file. (Single file with extension .ico can contain several different sized icon images to use later one with the most appropriate size).

`function GetFileIconCount( const FileName: KOLString ): Integer;`
Returns number of icon resources stored in given (executable) file.

`function LoadImgIcon( RsrcName: PKOLChar; Size: Integer ): HIcon;`
Loads icon of specified size from the resource.

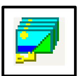
`function FileIconSystemIdx( const Path: KOLString ): Integer;`
Returns index of the index of the system icon correspondent to the file or directory in system icon image list.

`function FileIconSysIdxOffline( const Path: KOLString ): Integer;`
The same as FileIconSystemIdx[174], but an icon is calculated for the file as it were offline (it is possible to get an icon for file even if it is not existing, on base of its extension only).

`function DirIconSysIdxOffline( const Path: KOLString ): Integer;`
The same as FileIconSysIdxOffline[174], but for a folder rather then for a file.

4.23 List of Images

List of images (TImageList)



To store a set of icons of the same size, there is a special GDI object in Windows, which is called so, image list - a list of images. The **TImageList** object represents its object encapsulation. Constructor:

NewImageList - creates an empty list of images.

© 2024 Carl Peeraer

4.23.1 The methods and properties of the TImageList object

Let's consider a set of methods and properties of an image list:

- Descriptor and parameters 175
- Image manipulation: add, remove, load 175
- Accessing images 176
- Drawing 176

4.23.1.1 Descriptor and parameters

Handle - system descriptor of the GDI object of the image list;
ShareImages - a flag that controls the sharing of the descriptor between this object and other owners (if true, then the descriptor does not belong to this object, and will not be destroyed when the object is destroyed);
Colors - the color format of the images stored in the list. After adding images to the list, the format can no longer be changed (for this property to be changed, the list must be empty);
Masked - specifies whether the list of images uses a transparency mask for images (if not, then all images in the list are not transparent). Similar to Colors, this property can only be changed for an empty list;
ImgWidth - the width of each individual image stored in the list, in pixels;
ImgHeight - the height of each image in the list. Both the height and width of an individual image can also be set only before adding the first image to the list;
AllocBy - determines how many more images will be backed up in the list when the current reserve is exhausted. This property is passed to the system at the moment of creating a descriptor for the list of images, and it is no longer possible to change it after adding at least one image;

4.23.1.2 Image manipulation: add, remove, load

Add(bmp, msk) - adds a bitmap with the specified mask (both parameters are descriptors of bitmap types of hBitmap type);
AddMasked(bmp, C) - adds a bmp image (of the hBitmap type), building a mask for it by the image itself (assuming the color C in the image is transparent);
AddIcon(ico) - adds an icon to the list (the parameter is a descriptor of an icon of the hIcon type);
Delete(i) - removes the image with index i from the list (all other images in the list are shifted one position to the left, i.e. their indices are reduced);
LoadBitmap(resName, C) - the main method for loading images from the resources of the application itself, the C parameter is used as the color of the transparent area (in the image resource, one of the colors should be used to identify transparent areas);
LoadFromFile(s, C, imgType)- loads images from the specified file. Similar to the LoadBitmap method, the C color is used to build a transparency mask for masked lists. The last parameter sets the type of images (for icons and cursors, transparency is taken from the loaded images themselves, and the C parameter is ignored);
LoadSystemIcons(smallIcons) - Associates the list of images with the global system icon list, which stores, among other things, icons corresponding to the file types registered in the system.

© 2024 Carl Peeraer

In fact, although the name of this method begins with the word Load, there is no physical "loading" of the images. Simply referring to the images of the given list leads to referring to the icons of the system list. The system list cannot be modified (access is possible in the "read-only" mode).

To get information about which icons from the list correspond to which types of files, you need to use either API functions or KOL functions (**FileIconSystemIdx**, **FileIconSysIdxOffline**, **DirIconSysIdxOffline**);

4.23.1.3 Accessing images

Count- the number of images in the list;
Bitmap - returns the system descriptor of the bitmap (of the hBitmap type), in which the system stores all the images in the list (from left to right);
Mask - returns the hBitmap of the monochrome bitmap mask for all images in the list;
ImgRect(i) - returns a rectangle that stores the image indicated by the index i in the common bitmap;
Overlay[i] - manages overdraw images (or modifiers). Index i = 1..15 allows to set the overlay number for each overlay modifier, assigning a value to this property informs the image with which index in the list of images the overlay is located. These modifiers can be used when displaying element states in the list view and tree view visual objects;

4.23.1.4 Drawing

BkColor - background color, used for opaque drawing of transparent images from the list, in place of transparent areas;
BlendColor - the color that is blended with the color of the images, in the so-called "blended" drawing of the image;
DrawingStyle - style of drawing images from the list (built as a combination of possible flags "transparent", "by mask", "with 50% BlendColor blending", "25% BlendColor blending");
Draw(DC, X, Y, i) - draws an image with index i from the list to the specified DC context from the specified coordinate, while drawing the current drawing style DrawingStyle is used;
StretchDraw(DC, R, i) - similar to the previous one, but drawing is performed with scaling, and the image "fits" into the specified rectangle;

Running a little ahead, it should be said that the Mirror Classes Kit has a mirror component for representing a **TImageList** (the mirror is called **TKOLImageList**). You can put it on the form from the palette of components, and set its properties. As well as for the VCL TImageList component, double-clicking on this component calls its editor, where you can edit the list of images by loading the desired images from the image files.

An important feature of the mirrored **TKOLImageList** object is its ability to save application size. The general picture for the list of icons is saved in the application resources in the most compact format, in which all the colors present are preserved. That is, even if you used True Color for thumbnails, but the total number of colors does not exceed 256, 16 or 2, then the corresponding format 8, 4 or 2 bits per pixel will be used to store the final image (with a palette). And if all the

© 2024 Carl Peeraer

colors present contain 0 in the least significant bits (in 3 bits for the R and B channels, and in 2 bits for the G channel), then the 16 bits per point format will automatically be used.

4.23.2 List of Images - Syntax

```
const
  CLR_NONE                  = $FFFFFFFF;
  CLR_DEFAULT               = $FF000000;

const
  ILC_MASK                  = $0001;
  ILC_COLOR                 = $00FE;
  ILC_COLORDDB              = $00FE;
  ILC_COLOR4                = $0004;
  ILC_COLOR8                = $0008;
  ILC_COLOR16               = $0010;
  ILC_COLOR24               = $0018;
  ILC_COLOR32               = $0020;
  ILC_PALETTE               = $0800;

const
  ILD_NORMAL                = $0000;
  ILD_TRANSPARENT           = $0001;
  ILD_MASK                  = $0010;
  ILD_IMAGE                 = $0020;
  ILD_BLEND25               = $0002;
  ILD_BLEND50               = $0004;
  ILD_OVERLAYMASK           = $0F00;

const
  ILD_SELECTED              = ILD_BLEND50;
  ILD_FOCUS                 = ILD_BLEND25;
  ILD_BLEND                 = ILD_BLEND50;
  CLR_HILIGHT               = CLR_DEFAULT;
```

TImageList(unit KOL.pas) ← TObj 92 ← _TObj 92
TImageList = object(TObj 92)
Object type to incapsulate icon or cursor image.

type **TImageListColors** = (ilcColor, ilcColor4, ilcColor8, ilcColor16, ilcColor24, ilcColor32, ilcColorDDB, ilcDefault);
ImageList color schemes available.

type **TDrawingStyles** = (dsBlend25, dsBlend50, dsMask, dsTransparent);
ImageList drawing styles available.

type **TDrawingStyle** = Set of TDrawingStyles 177;
Style of drawing is a combination of all available drawing styles.

© 2024 Carl Peeraer

```
type TImgLOVrlayIdx = 1..15;
```

```
type TImageType =( itBitmap, itIcon, itCursor );
```

ImageList types available.

```
type PImageList = ^ TImageList;
```

```
type HImageList = THandle;
```

```
type PImageInfo = ^TImageInfo;
```

```
type TImageInfo = packed record
    hbmImage: HBitmap;
    hbmMask: HBitmap;
    Unused1: Integer;
    Unused2: Integer;
    rcImage: TRect;
end;
```

```
function NewImageList( AOwner: PControl ): PImageList 178;
```

Constructor of TImageList object. Unlike other non-visual objects, image list can be parented by TControl object (but this does not *must*), and in that case it is destroyed automatically when its parent control is destroyed. Every control can have several TImageList objects, linked to a simple list. But if any TImageList object is destroyed, all following ones are destroyed too (at least, now I implemented it so).

TImageList Properties

```
property Handle : THandle;
```

Handle of ImageList object.

```
property ShareImages : Boolean;
```

True if images are shared between processes (it is set to True, if its Handle is assigned to given value, which is a handle of already existing ImageList object).

```
property Colors : TImageListColors 177;
```

Colors used to represent images.

```
property Masked : Boolean;
```

True, if mask is used. It is set to True, if first added image is icon, e.g.

```
property ImgWidth : Integer;
```


© 2024 Carl Peeraer

Width of every image in list. If change, ImageList is cleared.

`property` **`ImgHeight`** `: Integer;`
Height of every image in list. If change, ImageList is cleared.

`property` **`Count`** `: Integer;`
Number of images in list.

`property` **`AllocBy`** `: Integer;`
Allocation factor. Default is 1. Set it to size of ImageList if this value is known - to optimize speed of allocation.

`property` **`BkColor`** `: TColor;`
Background color.

`property` **`BlendColor`** `: TColor;`
Blend color.

`property` **`Bitmap`** `: HBitmap;`
Bitmap, containing all ImageList images (tiled horizontally).

`property` **`Mask`** `: HBitmap;`
Monochrome bitmap, containing masks for all images in list (if not Masked, always returns nil).

`property` **`DrawingStyle`** `: TDrawingStyle`177`;`
Drawing style.

`property` **`Overlay`**`[ Idx: TImgLOVrlayIdx`178` ]: Integer;`
Overlay images for image list (images, used as overlay images to draw over other images from the image list). These overalay images can be used in **listview** and **treeview** as overlaying images (up to four masks at the same time).

`property` **`OverlayIdx`**`: Integer;`
Set this value to 1..15 to draw images overlayed (using Draw or DrawEx).

TImageList Methods

`function` **`ImgRect`**`( Idx : Integer ) : TRect;`

© 2024 Carl Peeraer

Rectangle occupied of given image in ImageList.

`function **Add**( Bmp, Msk : HBitmap ) : Integer;`
Adds bitmap and given mask to ImageList.

`function **AddMasked**( Bmp : HBitmap; Color : TColor ) : Integer;`
Adds bitmap to ImageList, using given color to create mask.

`function **AddIcon**( Ico : HIcon ) : Integer;`
Adds icon to ImageList (always masked).

`procedure **Delete**( Idx : Integer );`
Deletes given image from ImageList.

`procedure **Clear**;`
Makes ImageList empty.

`function **Replace**( Idx : Integer; Bmp, Msk : HBitmap ) : Boolean;`
Replaces given (by index) image with bitmap and its mask with mask bitmap.

`function **ReplaceIcon**( Idx : Integer; Ico : HIcon ) : Boolean;`
Replaces given (by index) image with an icon.

`function **Merge**( Idx : Integer; ImgList2 : PImageList 178; Idx2 : Integer; X, Y : Integer ) : PImageList 178;`
Merges two ImageList objects, returns resulting ImageList.

`function **ExtractIcon**( Idx : Integer ) : HIcon;`
Extracts icon by index.

`function **ExtractIconEx**( Idx : Integer ) : HIcon;`
Extracts icon (is created using current drawing style).

`procedure **Draw**( Idx : Integer; DC : HDC; X, Y : Integer );`
Draws given (by index) image from ImageList onto passed Device Context.

`procedure **StretchDraw**( Idx : Integer; DC : HDC; const Rect : TRect );`
Draws given image with stratching.

`function **LoadBitmap**( ResourceName : PKOLChar; TranspColor : TColor ) : Boolean;`
Loads ImageList from resource.

`function **LoadIcon**( ResourceName : PAnsiChar ) : Boolean;`

© 2024 Carl Peeraer

```
function LoadCursor( ResourceName : PAnsiChar ) : Boolean;
```

```
function LoadFromFile( FileName : PKOLChar; TranspColor : TColor; ImgType : TImageType ) : Boolean;
```

Loads ImageList from file.

```
function LoadSystemIcons( SmallIcons : Boolean ) : Boolean;
```

Assigns ImageList to system icons list (big or small).

```
function ImageList_Create(CX, CY: Integer; Flags: UINT; Initial, Grow: Integer): HImageList; stdcall;

function ImageList_Destroy(ImageList: HImageList): Bool; stdcall;

function ImageList_GetImageCount(ImageList: HImageList): Integer; stdcall;

function ImageList_SetImageCount(ImageList: HImageList; Count: Integer): Integer; stdcall;

function ImageList_Add(ImageList: HImageList; Image, Mask: HBitmap): Integer; stdcall;

function ImageList_ReplaceIcon(ImageList: HImageList; Index: Integer; Icon: HIcon): Integer; stdcall;

function ImageList_SetBkColor(ImageList: HImageList; ClrBk: TColorRef): TColorRef; stdcall;

function ImageList_GetBkColor(ImageList: HImageList): TColorRef; stdcall;

function ImageList_SetOverlayImage(ImageList: HImageList; Image: Integer; Overlay: Integer): Bool; stdcall;

function ImageList_AddIcon(ImageList: HImageList; Icon: HIcon): Integer;

function Index2OverlayMask(Index: Integer): Integer;

function ImageList_Draw(ImageList: HImageList; Index: Integer; Dest: HDC; X, Y: Integer; Style: UINT): Bool; stdcall;

function ImageList_Replace(ImageList: HImageList; Index: Integer; Image, Mask: HBitmap): Bool; stdcall;

function ImageList_AddMasked(ImageList: HImageList; Image: HBitmap; Mask: TColorRef): Integer; stdcall;

function ImageList_DrawEx(ImageList: HImageList; Index: Integer; Dest: HDC; X, Y, DX, DY: Integer; Bk, Fg: TColorRef; Style: Cardinal): Bool; stdcall;
```


© 2024 Carl Peeraer

function **ImageList_Remove**(ImageList: HImageList; Index: Integer): Bool; stdcall;

function **ImageList_GetIcon**(ImageList: HImageList; Index: Integer; Flags: Cardinal): HIcon; stdcall;

function **ImageList_LoadImage**(Instance: THandle; Bmp: PWideChar; CX, Grow: Integer; Mask: TColorRef; pType, Flags: Cardinal): HImageList; stdcall;

function **ImageList_LoadImage**(Instance: THandle; Bmp: PAnsiChar; CX, Grow: Integer; Mask: TColorRef; pType, Flags: Cardinal): HImageList; stdcall;

function **ImageList_BeginDrag**(ImageList: HImageList; Track: Integer; XHotSpot, YHotSpot: Integer): Bool; stdcall;

function **ImageList_EndDrag**: Bool; stdcall;

function **ImageList_DragEnter**(LockWnd: HWnd; X, Y: Integer): Bool; stdcall;

function **ImageList_DragLeave**(LockWnd: HWnd): Bool; stdcall;

function **ImageList_DragMove**(X, Y: Integer): Bool; stdcall;

function **ImageList_SetDragCursorImage**(ImageList: HImageList; Drag: Integer; XHotSpot, YHotSpot: Integer): Bool; stdcall;

function **ImageList_DragShowNolock**(Show: Bool): Bool; stdcall;

function **ImageList_GetDragImage**(Point, HotSpot: PPoint): HImageList; stdcall;

procedure **ImageList_RemoveAll**(ImageList: HImageList); stdcall;

function **ImageList_ExtractIcon**(Instance: THandle; ImageList: HImageList; Image: Integer): HIcon; stdcall;

function **ImageList_LoadBitmap**(Instance: THandle; Bmp: PKOLChar; CX, Grow: Integer; MasK: TColorRef): HImageList; stdcall;

function **ImageList_GetIconSize**(ImageList: HImageList; var CX, CY: Integer): Bool; stdcall;

function **ImageList_SetIconSize**(ImageList: HImageList; CX, CY: Integer): Bool; stdcall;

function **ImageList_GetImageInfo**(ImageList: HImageList; Index: Integer; var ImageInfo: TImageInfo): Bool; stdcall;

function **ImageList_Merge**(ImageList1: HImageList; Index1: Integer; ImageList2: HImageList; Index2: Integer; DX, DY: Integer): HImageList; stdcall;

© 2024 Carl Peeraer

```
function LoadBmp( Instance: Integer; Rsrc: PKOLChar; MasterObj: PObj ): HBitmap;

function LoadBmp32( Instance: Integer; Rsrc: PKOLChar; MasterObj: PObj ): HBitmap;
```

4.24 Before getting started with Visual Objects

At this, perhaps, it is worth pause and stop for a while the story about all sorts of "simple" objects produced from **TObj** 92. The fact is that very many of them, in the future, either interact with **visual objects**, or are useful only in the presence of visual objects. For example, dialogs, menus, even timers - they already need window objects to function properly. So now is the time to start describing the most important part of KOL - the TControl object type.

The **xHelpGen utility for KOL** provides a lot about KOL (and you can learn more by looking at the source code). But, unfortunately, much less information is available about MCK mirrors and about visual programming of MCK projects, so I will dwell on this point in more detail in the next article. The previous objects are objects of a significantly more temporary nature than what it would make sense to call "components" and customize at the time of form design. Therefore, they do not have MCK mirrors, except for graphical tools, the properties of which can be adjusted together with the properties of the visual objects that own them. The exception is **TImageList** 177, for which there is a mirrored **TKOLImageList** component.

As I mentioned at the beginning, the **TControl** object in **KOL** is not some basic type from which all visual objects corresponding to different types of windows would inherit. In the KOL library, all the main visual objects are encapsulated directly in TControl, similar to how it is implemented for **data streams** (**TStream** 108) or for **graphic canvas tools** (**TGraphicTool** 151). Visual window objects are created by various constructors, which all return a result of the PControl type, but the appearance and behavior of the resulting objects differs (it is in the constructor that it is determined what kind of functionality the created control element, or "control", performs - I will call them that and sometimes hereinafter , to be short).

When designing the TControl type in this way, I had to resort to some tricks. Not only function pointers are used, but also tables for handling the most common messages, and dynamic event handlers, and all kinds of flags. All this in order to save the size of the involved code, and for different "controls" to perform different required actions, if possible, with the same code.

In many cases it was possible to achieve "**functional polymorphism**", when the same method or property, without changing the name, performs different (but similar in meaning) work for different types of "controls". But sometimes it was not possible to do this, and for all such methods and properties that do not apply to all types of "controls" at once, the source code contains clarifications for which cases they can be applied. For some visual elements that have a large set of additional properties and methods, the names of these properties and methods contain a two-letter prefix that identifies the type of element to which they only refer (LV - list view, TB - tool bar, TC - tab control, TV - tree view, RE - rich edit, etc.).

© 2024 Carl Peeraer

4.25 Common Properties and Methods - TControl

Common Properties and Methods of Window Objects - TControl

Since the **TControl** object type provides the ability to work with window (sometimes pseudo-window) objects, properties and methods that determine the appearance, shape, size of the window, location (in relation to the parent, or to screen). Also, since windows "enter" into the relationship "parent" - "child window", all visual objects have a number of properties to implement these relationships (i.e., parent objects store a list of child window objects in relation to them, child objects have a link to parent window, and a number of properties for setting the order of work with them in the input focus).

All windows in the Windows environment are not just rectangles in which something is drawn, they are also code that works by processing messages sent to windows (from others or from the same application, as well as by the system). It is this code that provides the required functionality. Of course, some of the messages are processed automatically, depending on the type of the created window, and on the various styles and parameters specified during its creation. Moreover, some styles or modes of operation of windows are allowed to be changed in the process of work (and some remain with them while they are alive). That is, event handling is also an important common property of all window objects. And, finally, almost every window can have a number of optional attributes (that is, they can be present for one kind of windows, but for others they are simply ignored).

Let's immediately agree on some terms that will be used in describing the properties of window objects. In this context, I call the "parent" of an object not the "ancestor" of the object in the hierarchy of objects, but the "owner" of the object window. Those. the parent of the control object is the visual object in which this "child" object is nested. In Windows, some windows are nested inside others, while the nested windows become children of the parent, which is exactly what is meant when we talk about parent and child windows.

Note: in VCL there is a separate concept of "owner" of an object (Owner). The owner is the object that is responsible for destroying the given object when its own lifetime ends. KOL has an automatic destruction mechanism (Add2AutoFree) to provide similar ownership of objects, and when a visual object is first created, its parent window object is automatically owned.

Another important term (which I have already introduced above, getting a little ahead of myself) is the Applet. in a sense, it is analogous to the TApplication object in the VCL. The difference from VCL is that in KOL this object is encapsulated in the same TControl object type and is, generally speaking, a window object (representing an application button on the taskbar). There are also a number of differences. Perhaps it is worth noting here that the creation of this object is optional for a simple KOL application consisting of a single form (and in this case, the main, it is also the only, form of the application can successfully play the role of the applet, and then the Applet variable also refers to the main application form). Conversely, you can use additional instances of applets - on your forms, in order to

© 2024 Carl Peeraer

A form is defined in the same way as in the VCL - a top-level window object. These windows are called popups in Windows API terms. Their window parent is the "desktop". This does not in the least prevent us from assuming that the "parent" of a form in KOL is a special Applet object. Those. the coordinates of the borders of the forms are set relative to the entire screen, of course, but the Parent property will show on the Applet - this is done so that using the ChildCount and Children properties of the applet, you can view all the forms of the application.

KOL also has the ability to create "graphical" visual objects (and, the choice of such visual elements is much wider than in the VCL). In the following description, I will talk about visual objects, meaning any visual elements on the form, including non-windowed graphic controls. If the combination "window object" is used, then we are talking about the object to which the window is associated, with its own handle of the window registered in the system.

4.25.1 Properties and Methods of window objects

Now let's look at those properties, methods and events that are most common for all window objects encapsulated in the **TControl object type**. Since there are too many of these properties common to all types of "controls", I will try to divide them into groups and somehow classify them by purpose and scope:

- Window handle 185
- Parent and Child controls 186
- Availability and visibility 187
- Position and dimensions 188
- Painting 190
- Window text and font for the window 191
- Window color and window frame 191
- Messages (all window objects) 192
- Dispatching messages in KOL 193
- Keyboard and tabs between controls 196
- Mouse and mouse cursor 197
- Menu and Help 198
- Form and applet properties, methods, and events 198
  - Appearance (form, applet) 198
  - Messages (form, applet) 200
  - OnFormClick event (for form) 201
- Modal dialogs 202
- Reference system 202

4.25.1.1 Window handle

Handle - window handle (hWnd type, i.e. 32-bit number that uniquely identifies the window in the system). In KOL projects it is also possible to create pseudo-window objects, similar to

© 2024 Carl Peeraer

TGraphicControl in VCL, which do not have their own descriptor - for them Handle always contains 0;
HandleAllocated - checks if a window has been created for the object;
GetWindowHandle - returns a handle (Handle) of the window, creating it if it has not yet been created (this automatically creates windows for all parent window objects, if they have not been created yet);
CreateWindow - does the same as **GetWindowHandle**, i.e. creates a window for the window object if it has not been created yet;
CreateChildWindows - creates windows of all child TControl window objects (recursively) if they have not been created yet;
Close - closes the window, and destroys the object (if the window is the main form of the application or an Applet object, then the application ends);
ClsStyle - the style of the window class (the number used when creating the window as the **ClsStyle** parameter in the call to **CreateWindowEx**), usually after creating the object there is no need to change this property;
Style - window styles, changing this property allows you to fine-tune window properties, intended for professionals;
ExStyle - extended window styles, similar to the previous one;
SubClassName - the name of the window class, by default it returns the string 'obj_XXXX', where XXXX is the name of the window class (for example, for buttons: 'obj_BUTTON').

4.25.1.2 Parent and Child controls

Unlike the VCL, KOL does not have an Owner property for components. Ownership is implemented here by the **Add2AutoFree [Ex]** method. If you use MCK to create a project, then all objects of the form are attached to the object-holder of the form precisely by calling the **Add2AutoFree [Ex]** methods in order to ensure that they are automatically destroyed along with the form when its existence ends. For visual objects, there is a Parent property (and others accompanying it), which uniquely defines the relationship between a parent window and a child window in the Windows environment. Just like in the VCL, through the Parent property, you can change the parent of a visual object dynamically, at runtime. But not always: for example, the combo box does not allow such liberties, this is the behavior of the Windows API.

Parent - reference (**PControl**) to the parent window object (may be absent for top-level windows (for Applet and for forms), as well as in the case when the window was created as a child in relation to someone else's window (in the latter case, the **parent** is indicated by **ParentWindow**);
ParentWindow - returns a handle to the **parent** window;
ParentForm - looks through all the parents up the chain, and returns the object of the form on which the given window object lies;
ChildCount - returns the number of child window objects;
Children[i] - returns a pointer (**PControl**) for the i-th child element in the list (note: KOL still retains the **MembersCount** and **Members [i]** properties, but this is a "tail" that has been going on since the XCL days, in reality these properties are not needed, enough ChildCount and Children [i]);

© 2024 Carl Peeraer

ChildIndex(C) - returns the index of the specified child window object (or -1 if such an object is not found in the list of children);
MoveChild(C, i) - "moves" child "control" to the specified position in the list of child objects;
IsControl - checks that the object is exactly a "control" (that is, not a form or an applet).

4.25.1.3 Availability and visibility

Enabled - the "window is allowed" property, i.e. reacts to mouse and keyboard (usually it also changes its visual style, if "not allowed", to show the user that it is useless to click on it at the moment). An important point: although by setting the Enabled property to false, we thereby "disable" automatically and all its child window objects, nevertheless, the appearance for child objects will not change itself, you should use the EnableChildren method to ensure that all child window objects are unavailable. objects);
EnableChildren(e, recursive) - allows or denies an object together with its child objects (if recursive = true, then all child elements of all lower levels are recursively enumerated);

Visible - property "visible window". In fact, it sets the potential visibility of the window, i.e. acts exactly the same as in the VCL for TwinControl. When Visible is true, the window is truly visible only when all of its parent objects are visible. And if one of the parents is an object of the tab control type, then it is also required that the tab on which this object is located, together with all intermediate parent objects, is selected as the current one;
ToBeVisible - the property for reading "the window is really visible", in this property the drawback of the Visible property is eliminated, and it is taken into account that all parent objects are visible, and all tabs in the tab control are selected as the current one - the parents on which this object lies;
CreateVisible - this property sets whether the object will be made visible immediately at the time of creation; by default this property is set to false, which ensures the minimum number of redrawings at the time of creating the form and its first display on the screen;
Show - a method for displaying a window and activating it, i.e. transferring keyboard focus to the object window (this method is usually used, by analogy with VCL, to "show" a form, but it can also be successfully used for any visual object, especially for an object that can accept keyboard input, which is why I cite this method is here);
Hide - hides the window (equivalent to setting the Visible property to false);

OnShow - an event that fires every time the object window becomes visible. I emphasized, because sometimes it is necessary to perform some actions when the form is first shown, for example, but the programmer forgets to check that the form is actually displayed now for the first time (and not the second, third, and all subsequent times). An important detail: when a window becomes visible as a result of showing a parent window (for example, a form), this event is not triggered;
OnHide - an event that fires when the window becomes invisible. Just like the OnShow event, it fires only when the visibility (i.e. the Visible property) of the window itself changes, and does not react to the change in the visibility of the window parents;
BringToFront - a method for transferring a visual (window) object to the foreground (i.e. if it is partially or completely obscured by other objects, then it comes to the fore, and itself becomes overlapping them in whole or in part);

© 2024 Carl Peeraer

SendToBack - the inverse method of **BringToFront**. The order of overlapping windows is changed in such a way that the window of the given object goes to the back in comparison with the neighboring windows.

4.25.1.4 Position and dimensions

Align - "alignment" of the window object on the parent window. The values are the same as in the VCL for the Align property available in the VCL for some window object classes. The main difference from VCL is that in KOL it is possible to align almost all types of "controls" (perhaps with a few exceptions, namely: it is undesirable to try to align the combo box in height, as this can lead to unpleasant consequences in Windows 9x). If an alignment is specified that is not equal to caNone, then some of the characteristics of the bounding rectangle (below) cannot be changed and is controlled by the alignment (for example, if the window is aligned to the left - caLeft, then it is possible to change only the window width, but not the height and not the coordinate of the upper left corner);
SetAlign(align) is a "pass-through" method that sets the Align property and returns a pointer to the window object itself. "Through" methods are convenient to use when creating window objects, for example:

```
NewButton (Panel1, 'Button1') .SetAlign (caLeft) .OnClick: = Click1;
```

BoundsRect - a rectangle that sets (and returns) the coordinates of the window object relative to the parent window (or relative to the entire screen - for top-level windows, i.e. for the form and applet); There are also properties for separately accessing / changing each of the window coordinates in terms of "top-left corner position" / "width" / "height":

Left - left coordinate of the window;
Top - the top coordinate of the window;
Width - window width;
Height - window height;

ControlRect - the same as BoundsRect, but calculated by the stored coordinates, and not by calling the API;
Position - coordinates of the upper left corner of the window as a point (TPoint);
ControlAtPos(X, Y, IgnoreDisabled) - determines which (child) visual object is located at the given coordinates;
OnMove - an event triggered when the coordinates change (when the window moves on the screen);
Dragging - checks if the object window is currently being "dragged" with the mouse (true after calling **DragStartEx**);
DragStart - starts "dragging" the object window (control or form) with the mouse. Dragging ends when the user releases the left mouse button (if at the time of the method call the left mouse button is not pressed, then dragging does not start);
DragStartEx - similar to the previous one, but to stop dragging from the program it is required (and possibly) to call the **DragStopEx** procedure;
PlaceRight - "through" method that places the current object to the right of the previous one in the list (at a distance specified by the property **parent**);

© 2024 Carl Peeraer

PlaceDown - "through" method, placing the object below the previous one, the left coordinate is set to the leftmost accessible position (Border + Margin of the parent), and the top coordinate is provided below the bottom edge of all previous visual objects;
PlaceUnder - the "pass-through" method, which places the object directly under the previous one (ignoring all the others previously placed);
Border - short integer (from –128 to +127) specifying the width of the parent window border. This property is used both for alignment purposes and for the above PlaceXXXX placement functions - to ensure the distance between the visual objects child with respect to this control. That is, unlike VCL, when aligning child visual objects, the space specified by the parent's Border property is provided between them. (See also the properties of MarginXXXX);
MarginTop - a short integer that specifies the additional distance (added with Border) from the top edge of the client part of the parent window to the first child window, when it is automatically placed (for example, when aligning child visual objects using the Align property). This number can be negative;
MarginBottom - similar to **MarginTop**, but sets an additional distance from the bottom edge of the parent visual object;
MarginLeft - similar to **MargintTop**, but for the left edge of the parent window;
MarginRight - similar to **MarginTop**, but for the right edge of the parent object;
SetSize(W, H) - "through" method, allows you to set a new window size (if W or H does not exceed zero, the corresponding size does not change);
Size(W, H) - similar to the previous one, but automatically resizes the parent window accordingly (and all parent windows - recursively);
AutoSize(on) - enables or disables automatic resizing of the window to its contents (unlike VCL, such automatic resizing in KOL takes place not only for the "label" TLabel, as in VCL, but for a slightly larger number of visual objects - buttons, check boxes and radio boxes, for example);
IsAutoSize - checks if automatic resizing is enabled for the object;
CanResize - sets whether the window can be resized (not only by the user with the mouse or from the window's system menu, but any API call will not be able to resize the window for which the **CanResize** property is set to false: you should be careful and set this value to false only after how the window is already sized). The MCK mirror of the form (class TKOLForm) has a corresponding design-time property CanResize, setting which to false will add false to the form's CanResize property in the form initialization code - after the form's dimensions are determined;
MinHeight - minimum window height;
MinWidth - minimum window width;
MaxHeight - maximum window height;
MaxWidth - maximum window width;
OnResize - an event triggered when the object window resizes, for any reason;
AnchorRight - setting this property to true results in the position of the right edge of the visual object being snapped to the size of the parent "control";
AnchorBottom - similar to the previous one, but snaps the bottom edge of the object to the height of the parent;
AnchorTop - similar to the previous ones, but anchors the window to the top edge of the parent;
AnchorLeft - the same, but snapping occurs to the left edge of the parent window;
Anchors(L, T, R, B) - "pass-through" method that allows you to set anchor **AnchorLeft**, **AnchorTop**, **AnchorRight** or **AnchorBottom** in one call;

© 2024 Carl Peeraer

ClientRect - returns the coordinates of the client side of the window. Because the result is returned in the coordinates of the client-side itself, then for all window objects, except for "graphic" ones, the Left and Top fields of the returned rectangle are always equal to 0;
ClientWidth - the width of the client side of the window (including it can be changed through this properties: the width of the entire window will be changed accordingly);
ClientHeight - similar to the previous property, the height of the client area of the window;
SetClientSize(W, H) - similar to SetSize, but resizes in terms of the client side of the window;
Client2Screen(P) - translates the coordinates of a point from the client coordinate system to coordinates on the screen;
Screen2Client(P) - back to the **Client2Screen** method, for the specified coordinates on the screen, returns the client coordinates of the point.

4.25.1.5 Painting

Invalidate - forces the window to be redrawn as soon as possible (that is, marks the client part of the window as "invalid", as a result, the system, in the order of the queue, sends the window all the necessary messages to redraw its contents);
InvalidateEx - forces the window and all its child windows to be redrawn recursively;
InvalidateNC(recursive) - Marks as "invalid" the entire window along with the non-client part, and if the parameter recursive = true, then in this case the same is done recursively for all child windows;
Update - immediately refreshes the window (if required, the system sends it all the necessary messages);
BeginUpdate - increases the counter of "prohibitions" on updating the window, as a result the window will not redraw its contents until this counter is reset after the corresponding number of calls to the EndUpdate method;
EndUpdate - decreases the counter of "prohibitions" for redrawing. When this counter reaches 0, the window is redrawn if necessary;

DoubleBuffered- determines whether double buffering is used to draw the window (and all windows that are children of this object). Double buffering increases the speed at which the visual is redrawn. In KOL, almost any visual object (including a form) can use double buffering. Exception: rich edit;
DblBufTopParent - returns the parent visual with the top-level double buffering property;
Transparent - transparency of the background of the client part of the visual object. In KOL, a lot of visual elements (there are a few exceptions) can be "transparent". To implement transparency, double buffering is also used, i.e. all parents of transparent "controls" are automatically set to DoubleBuffered = true;
UpdateRgn - handle of the region to be redrawn. Set in the OnPaint event handler, can be used to restrict the redrawing area in order to optimize performance;
EraseBackground - a boolean flag for passing it to the GetUpdateRegion function, when the system asks for the window area to be redrawn when processing the WM_PAINT message (what did you think?). If you set this property to true, then the system will ensure that the background is erased in the entire marked area (which can lead to flickering with frequent redrawing, so this property is set to false by default);

© 2024 Carl Peeraer

OnPaint - for almost any windowed and non-windowed visual object in KOL, it is possible to override the drawing procedure by assigning a handler for this event (but in this case, your code must provide all the drawing of the client side of the window). Usually such a handler is assigned to a paint box object specially designed for this purpose, sometimes for panels (in fact, panels do not differ from paint box controls except for the presence of text: in KOL, the "paint box" is an ordinary window object, and can serve parent for other visual objects;
OnEraseBkgnd - this event will allow you to define your own procedure for erasing the background. If such a handler is assigned, the system no longer erases the background on its own by the default procedure. If at the same time the handler code does nothing, then the erasure is not performed at all, and if you ensure the correct erasure of the background simultaneously with painting in the OnPaint event handler, this allows you to completely eliminate the flickering of the image during redrawing;
Canvas - the canvas of the visual object. Almost all windows allow you to override or redefine the standard window drawing procedure, for example, see the events OnPaint, OnDrawItem, in this case it is convenient to program the drawing procedure through the methods and properties of the canvas object provided here.

4.25.1.6 Window text and font for the window

Caption - the same as Text - the main text of the visual object (for the form - the title, for the applet - the title name, for the button - the text of the button, etc.). Stored in a buffer, at least until the moment the window is created for the object (after which the contents of the buffer are ignored, and the property is read and written by reading or setting the window text through API functions). Although there are such types of windows for which this property is meaningless (list view, for example - this property cannot affect its appearance in any way), I still list it in the list of the most common properties, because visual objects that use it , lots of;
Text - the same as Caption - above;
Font - the font for displaying the text in the object window, if it is not changed, then the font assigned to the parent is used, and if there were not only changes to the font, but even references to it, then the system default font (FixedSys) is used. I draw your attention to the fact that when setting the properties of visual elements in MCK, there is an additional design-time property ParentFont, the value of which allows you to control whether the font setting will be performed when the form is initialized (false), or the parent font will be used (true) ;
TextAlign - text alignment horizontally (left, right, center), makes sense for almost all controls that have a displayed text (Caption) *;
VerticalAlign - vertical alignment of text in the window (text can be pressed to the top, to the bottom, centered). Same as TextAlign, it works for almost all mono-text visuals (sometimes there are restrictions on the combination of certain values of TextAlign and VerticalAlign).

4.25.1.7 Window color and window frame

Color - the main color of the visual object. The application depends on the purpose of the object: for panels and labels, this property sets the background color, for edit boxes and other editable "controls" - the background color for the text input field. When setting up a visual element in MCK, there is also an additional design-time property: **ParentColor**, which controls

© 2024 Carl Peeraer

whether the color will be set in the form initialization code that generates the MCK, or the color will not be set (and, accordingly, will be inherited parent's color);
Brush - a brush for filling the background, if set, then the Color property is not used for filling, but the brush settings are used;
Ctl3D - the style of pseudo-three-dimensionality of the borders of the visual object (by default, true, and this exactly corresponds to the standard external design of window objects in Windows), this property is also inherited from the parent object by default;
HasBorder - just like **Ctl3D**, it controls the presence of a 3D frame around the window, but in a slightly different way. In particular, it is allowed to change this property at runtime. For a form, this property also makes sense, and allows, including dynamically, to remove or add a window frame, by "hooking" the mouse cursor over the border of which the window can be resized. In addition, a borderless form cannot have a non-client part at all, including a title. Such borderless windows are often used as splash windows, or to create full-screen applications.

4.25.1.8 Messages (all window objects)

WndProc(Msg) - one of the few virtual methods in TControl, it can also be used to override the main message handling procedure when inheriting a new window object from TControl;
OnMessage - a custom handler for any window messages arriving at the object window. In KOL, it is possible to assign such a handler to any window object, including an applet, form, or control, and handle any window messages sent by the system, with your code or other applications as needed. (See also **AttachProc** and **AttachProcEx**).
Many programmers, starting to work in KOL, are puzzled by the fact that the usual Delphi way of handling arbitrary messages by declaring (for example, in the declaration of a form object) a dynamic handler using the message directive does not work. So, it shouldn't work (or even compile). This directive is not supported by simple Pascal objects, but only by classes derived from TObject.

However, nothing is easier than assigning an event handler to the OnMessage event and writing code to handle the requested message. Moreover, this method allows you to expand the functionality of any window object, and not only the form (and for this there is no need to produce a new heir). On the right is an example of a hypothetical handler that processes only a few window messages while allowing others to be processed the same way.

```
procedure TForm1.Form1Message (
Sender: PObj; var Msg: TMsg;
 var Rslt: Integer): Boolean;
begin
 Result: = false;
 case Msg.message of
 WM_USER + 100: // your code
 ...
 end;
end;
```

The **OnMessage** event handler must return (Result) a boolean flag that allows further processing of this message (thus, it has the ability to prevent the message from being passed to other handlers if further processing is unnecessary or harmful). To the sender of the message (for example, to the system, if this is a normal window message), the result is returned as an integer through the Rslt parameter, please do not confuse them.
The message handling provided by the **OnMessage** event is not only as good as, but also more convenient than the dynamic message mechanism in the VCL. For example, such a handler can

© 2024 Carl Peeraer

be easily added to any visual object on a form (MCK). And at the same time, there is no need to inherit your visual object class, and then come up with a way to replace the object on the form with your object (which is especially frustrating when it comes to the object used as a parent, or more precisely, the owner of the child visual objects lying on it).

Perform(msgcode, wParam, lParam) - sends a message to the object window for immediate execution;

Postmsg(msgcode, wParam, lParam) - puts a message in the queue of the object window, execution itself does not start until it is selected in the order of the queue

AttachProc(proc)- attaches the specified procedure to the list of dynamic message handlers for the window. In KOL, this is the main way to extend the functionality of existing window objects, and one of the most important tools for saving code size, since it allows you to attach a message handler required by a property only when there is a call to a certain property or when an event handler is assigned. In addition, in this way, you can attach several handlers that are called sequentially, from the last one attached to the first, until one of them returns the "do not process anymore" flag (Result = FALSE).

AttachProcEx(proc, flag) - similar to the previous one, it also allows you to specify that this handler continues to function after the moment when the application termination process has already begun (in most cases, it is required, on the contrary, to stop processing all messages from the moment when the application began to close, but there are also exceptions to of this rule);

IsProcAttached(proc) - verifies that the specified procedure is attached to the list of dynamic message handlers for the window;

DetachProc(proc) - removes the specified procedure from the list of dynamic handlers.

4.25.1.9 Dispatching messages in KOL

The main message loop is in the Run procedure, which starts immediately after the specified forms and applet are created. You can easily replace this procedure with your own (for example, to provide higher or lower priority for some types of messages). In the case of the MCK project, for this it is enough to place your Run procedure directly in the main file of the DPR project, before the INCLUDE directives, which include the generated application start code. In the case of programming without MCK, you write the call to the Run procedure yourself: you can not call it, but write your own message loop.

Dispatching is that for each message selected from the queue, **TranslateMessage** and **DispatchMessage** are called. These are API functions that handle routing messages to specified windows. After that, messages go to the global procedure WndFunc *. This is where the main message dispatcher resides in KOL. Its main task is to find an object corresponding to the window indicated in the message (hwnd field). Previously, the GetProp API function was used to map a visual object to a window in KOL, but more recently it was decided that the so-called "custom" window attribute field is usually not involved, and using it gives a slightly faster and shorter code. As a result, in KOL, a window can be bound to an object either in one way or another, depending on the value of the USE_PROP option.

When (and if) the WndFunc procedure has received the address of an object (it must be a **TControl** object), it can already call its methods. This is where the WndProc method is called,

© 2024 Carl Peeraer

which carries out further processing of window messages. But if the address of the object could not be obtained, that is, the message is intended for a window that does not have a **TControl** object attached to it, then the WndProc method of the Applet object is called, and if it is not assigned, then the message is passed to the system's default handler, the DefWindowProc procedure.

A window may not be associated with a specific TControl object in several cases. This can be a subordinate service window for some window object (for example, the drop-down list and the input field for the combo box itself are organized in the system as children of the combobox itself). It can be a window that you created without using TControl, your own code. Or the window is still being created and the object has not yet been bound.

All **TControl** objects have the same WndProc procedure, and only the most common messages are handled in it. The most important thing is that before starting to process the received message, this method first tries to "disown" it, if possible. Namely, the algorithm is as follows.

First, if the Applet variable exists and an **OnMessage** event handler is assigned to it, then that (your) handler is called first. If the **OnMessage** event handler returns FALSE, then the message is considered fully processed, and it is no longer processed in any way, including by the default message handling procedures.

Second, if an **OnMessage** event handler is assigned to the **TControl** object itself that received the message, then this (your) handler is called. Again, if the event handler returns FALSE, then the message is considered fully processed and all processing for that message ends.

© 2024 Carl Peeraer

Of course, returning TRUE to the WndProc procedure from the **OnMessage** event handler (and other handlers) does not mean that the system will necessarily agree with your instructions. If any requirements for processing the message are not met, then there is a possibility that the system will send this message again and again until it receives what it wants. For example, if in response to the WM_PAINT message the system does not return the value 0 (the Rslt parameter), the system will assume that the application for some reason does not want to fulfill the order immediately and wants to postpone it for some time. And he will send a message again, and again. There is a possibility that an incorrectly written message handler can seriously slow down the operation of the application; you must be careful when writing the code for the OnMessage event.

© 2024 Carl Peeraer

Third, and this is the most important point, the executor of the message handlers dynamically attached (by the **AttachProc**, **AttachProcEx** methods) to the object is called. These handlers are intended primarily for internal use within the KOL library itself, to add the required message handling functionality for specific kinds of window objects. But they can be used by programmers as well, just like the fixed custom OnMessage handler. If such dynamic handlers are attached (and this is true for almost all window objects), then the EnumDynHandlers function begins to work. Its task is to view all dynamic handlers (from the last attached to the first), calling them one by one, at least until the next handler in the chain "says"

In the picture attached above, I tried to depict the message dispatching process. And only if the specified message handling is not found or has allowed the message to be processed further, the TControl.WndProc method will handle the most common messages. Namely: WM_CLOSE, WM_SIZE, WM_SYSCOMMAND, WM_SETFOCUS, WM_SETCURSOR, WM_CTLCOLORxxxx, WM_COMMAND, WM_KEYxxxx. And for all other messages will call the default system handler.

4.25.1.10 Keyboard and tabs between controls

TabStop - determines whether the window object is contained in the general list of form objects that can receive focus when pressing the Tab and Shift + Tab keys (but by default, processing of these keys is not connected, this must be done additionally by your own code, see the **Tabulate** and **TabulateEx** methods for the form and applet);
TabOrder - determines the order of the windowed object in the list of objects that can sequentially receive keyboard input focus when pressing the Tab and Shift + Tab keys (see the note on the previous TabStop property);
LookTabKeys - a list of keys that can be used for tabulating between controls that have **TabStop** = true (usually the default list does not need to be changed, each type of window object can have its own list of such keys);
GotoControl(Key) - performs tabulation as if the specified key was pressed;
Focused - checks if the window object has captured the keyboard input focus, and allows this focus to be passed to the object window when this property is set to true;
DoSetFocus - this method tries to transfer the input focus to the window of this object (similar to assigning true to the Focused property), and returns a sign of the success of this operation;
OnEnter - an event that is triggered when the object window receives keyboard input focus;
OnLeave - the event inverse to OnEnter: fires when the object window loses keyboard focus and the focus moves to another application window (but this event does not fire when the entire application loses focus; to respond to such an event, you need to catch it, for example, in the OnMessage handler forms, message WM_ACTIVATE);
OnKeyDown - an event that is triggered for an object when a button is pressed on the keyboard, when the object window has the keyboard input focus. It is possible to perform some actions when pressing any keys, and reset the Key parameter, preventing further processing of this key;
OnKeyUp - similar to the previous one, but triggered when the pressed button is released;
OnChar - triggered when a typed character arrives from the keyboard. Because there are usually fewer buttons on the keyboard than can be entered printable characters, then pressing and releasing buttons are translated into printable characters in such a way that sometimes a key combination forms a character or several characters, depending on the installed equipment, on

© 2024 Carl Peeraer

regional settings, the currently selected input language and etc. Just as it was done for the **OnKeyDown** event, in the OnChar event handler, it is also possible to set the Key parameter to # 0 when some characters are received, preventing further processing of these characters;
IgnoreDefault - if this property for a control is set to true, then when it is in focus from the keyboard, the <Enter> key does not cause the default button to be "pressed" (DefaultBtn);

4.25.1.11 Mouse and mouse cursor

Cursor - the mouse cursor in the window (can be overlapped for all visual objects by setting the global variable **ScreenCursor**). Unlike VCL, this property in KOL, observing the principle of the minimum number of object references, stores a number that is a handle to a cursor, of the hCursor type. Note: to immediately change the cursor on the screen, it is not enough to change this property; you must also call the SetCursor API function, passing the same cursor as a parameter to it;
CursorLoad(inst, s) - loads the specified resource and sets it as the window cursor;
OnMouseDown - the event of pressing one of the mouse buttons (left, middle or right). Unlike keyboard events, which are sent only to the window in focus, all mouse events are triggered for all visual objects, starting with the most nested ones (which makes it possible to create a common handler for the required mouse events for the parent window);
OnMouseUp - mouse button release event;
OnMouseMove - event of moving the mouse pointer;
OnMouseDblClk - mouse double click event. May only occur for windows that have a corresponding **CS_DBLCLKS** flag in their class style;
OnMouseWheel - mouse wheel rotation event;
OnClick - the event of "pressing" on the control. I included it here, in the list of the most common properties, since it is typical for almost all visual objects (except for the form itself - only **OnMouseXXXX** events are valid for it!). Perhaps the placement of this event here is also somewhat out of place because this event can be triggered not only by clicking the left mouse button (for buttons, for example, this event also occurs as a result of pressing the <spacebar> key on the keyboard). But its name quite eloquently describes the purpose of this event (to react to a mouse click), so let it be here. Although on the form itself, this event just does not fire, just because this event is processed not in response to a mouse click, but to the arrival of a WM_COMMAND or WM_NOTIFY window message.
OnMouseEnter - an event that is triggered when the mouse enters the area visually occupied by the object for the first time;
OnMouseLeave - an event that is triggered when the mouse cursor leaves the window;
MouseInControl - checks if the mouse cursor is within the visual boundaries of the object. Works only if at least one of the handlers for the **OnMouseEnter**, **OnMouseLeave** events is assigned (otherwise it always returns false);
LikeSpeedButton - this method changes the behavior of the window object in such a way that when the mouse button is pressed on it, the OnClick event (if assigned) is triggered, but after that, the input focus is returned to the window to which it belonged before the mouse click. this is similar to how TSpeedButton works in VCL, but in KOL any window object can behave this way. Although, of course, this property is designed primarily for buttons;
OnDropFiles - this event is triggered when the user has selected one or more files in an application (most likely in Windows Explorer), then dragged them with the mouse onto the

© 2024 Carl Peeraer

window of our window object, and released the left mouse button (a common Windows operation is "dragging" objects with the mouse , drag and drop). In the event handler, you can get a list of files thrown in this way, and do something with them (the files themselves remain where they were - at least until the moment this event occurs, and our application receives only a list of lines containing paths to these files). This event can be assigned to the entire form, or to a child window object, and it affects, among other things, all nested controls.

4.25.1.12 Menu and Help

SetAutoPopupMenu(PM) - Assigns an auto-popup context menu to an object (and all child objects that do not have their own auto-popup menu). The automatic menu can be invoked by right-clicking (unless a right-click handler has been assigned to prevent this action), as well as by a specially designed key on the keyboard when the window is in focus (similarly, unless this key is prevented from being processed). See also the description of the TMenu object in the corresponding section;
HelpContext - the number used in the built-in help system to identify the visual element (to connect the help, the programmer must also perform additional steps, see HelpSupport);
AssignHelpContext(i)- "pass-through" (returns a pointer to Self) method for setting the value of the HelpContext property.

4.25.1.13 Form and applet properties, methods, and events

Since in KOL both the form and the controls are implemented in the same object type, all of the above properties are also valid for the form (and some for the applet). Where it was obvious that this property is applicable in general to all visual objects, I did not even stipulate this (for example, it is obvious that the size and position of the window are properties of windows in general, and therefore are applicable to forms, in particular). Where it's not entirely obvious, I've mentioned that this property applies to a form or applet as well. But below we will talk about the properties and methods that are characteristic of the latter. Those. this does not mean that syntactically these properties are closed for use. The KOL library does not bother the compiler with additional shields against inappropriate use of properties uncharacteristic for an object,

There will be fewer "form-only" properties, but it still makes sense to split them into some groups:

4.25.1.13.1 Appearance (form, applet)

HasCaption - setting this property to false (including dynamic) allows you to remove the title bar from the form window, including the window state control buttons located on it;
StayOnTop - a property for a form, allows you to place it on top of all windows for which the same method is not applied (and at least for some time after the call, on top of all windows in the system in general);
AlphaBlend - a number from 0 to 254 specifying the degree of translucency of the form (works only in Windows 2000 and higher). A value of 0 corresponds to full transparency, when the window is not visible at all, 255 means no transparency, i.e. other windows through this window in this case do not "shine through" at all;

© 2024 Carl Peeraer

Menu - descriptor of the form menu (not 0 if the form window has a main menu at the top of the window);
Icon - window icon (for form or Applet);
IconLoad(inst, s) - loads the specified resource of the RT_ICON type and sets it as a window icon (Icon);
IconLoadCursor(inst, s) - loads a resource of type RT_CURSOR and sets it as a window icon (Icon);
MinimizeNormalAnimated - this procedure includes a special additional handler for the application minimization message for the application, in which the animation of the minimization process becomes more similar to what is observed in other applications. If this procedure is not called, then minimization occurs visually in a somewhat strange way (but the programmer may wish that the windows in his application were minimized without animation at all). When setting up a form in MCK, the form object has an additional MinimizeNormalAnimated property that determines whether to generate a call to this procedure in the form initialization code;
OnMinimize - an event for the form and applet, allows you to perform some actions when minimizing the window. For example, the handler can hide all windows of its application, and instead show the application icon in the system area of the taskbar (called the tray, tray - "tray"). This action is called "minimizing to tray" in slang;
OnMaximize - an event for the form that is triggered when the window is maximized;
OnRestore - event of restoring a window from a minimized state (for a form or applet);
OnClose - an event that is triggered when trying to close the window. If the Accept parameter is set to false in the event handler, then the window will not close (additionally, you can hide the window - for the user this will not differ much in the visible result, but the form object will not be destroyed, and next time it will be enough make it visible, this is the usual mechanism for working with dialogs);
IsMainWindow - checks if the given object is the main form of the application. As with the default VCL, the first form created automatically becomes the main form. Closing this form (usually) terminates the application (but there may be exceptions), minimizing this form leads to minimizing the entire application, etc .;
IsApplet - checks that the given object (forms) is exactly an applet (this method can be called, generally speaking, for any TControl object);
IsForm - checks that the given visual object is a form, i.e. a top-level window located directly on the desktop. This method is also suitable in the context of any window object, including a "control", but for some voluntaristic reasons I put it here;
SimpleStatusText - simple text displayed in the status bar (assigning a value to this property creates a status bar with a single panel). In the KOL form, if any non-empty text is added to the status bar, the form is increased in height and the status bar appears. When you assign an empty string to this property, the form is reduced in height and the status bar disappears. In order for the form to have at least an empty status line initially, it is necessary to assign a string containing spaces to this property during its initialization (one space is enough);
StatusText[i] - text in the i-th panel of the status line (i = 0..254);
StatusCtl - returns a pointer to a special object associated with the status bar (the object itself is created automatically when the StatusText, SimpleStatusText properties change, and is destroyed when the status bar disappears);
StatusWindow - handle to the status bar window;
RemoveStatus - removes the status bar (reducing the shape in height);

© 2024 Carl Peeraer

StatusPanelCount - the number of panels in the status bar;
StatusPanelRightX[i] - the right border of the i-th panel in the status bar;
SizeGrip - the presence of a dashed element in the lower right corner of the status bar, which can be used to resize the shape with the mouse. This property should be set before creating the form window.

I would like to draw your attention to the Visible property, which is common for all visual objects. When it comes to a standalone applet object that does not match the main application form, this property refers to the visibility of the application button on the taskbar. Therefore, the question of how to remove this button from the taskbar is solved in KOL by setting the Visible property to the Applet variable. But, as I said, the applet must be a special object, and not the same as the main form.

4.25.1.13.2 Messages (form, applet)

ProcessMessage - processes one message in the message queue;
ProcessMessages - processes all accumulated messages in the message queue for the window;
ProcessMessagesEx - the same as the previous method, but works better for cases where the application is minimized or is not a background application. Usually the previous method is sufficient. This method pre-sends the CM_PROCESS message to the window in order to "stir up" the message queue;
ProcessPendingMessages - similar to **ProcessMessages**, but if there are no messages from the mouse and keyboard in the queue, then an immediate return occurs and execution continues;
ProcessPaintMessages - the same as **ProcessMessages**, but messages are executed only as long as there is at least one message for drawing WM_PAINT windows in the queue, and if there are no such messages in the queue, control returns immediately;
OnQueryEndSession - an event for an applet that allows you to "respond" to a system request about the possibility of terminating the current session (and rebooting or turning off the power). During execution, the handler for this event can ask the user to write unsaved data to disk, for example, or to perform some other action. The handler can further parse the reasons for the request in the **CloseQueryReason** property;

Please note: if such a handler is not installed, and you yourself do not process such a message in any way, then when the Windows session ends, your application will be unloaded without warning, and the **OnDestroy** events will not even be triggered. The assignment of such a handler, even if it does nothing, increases the size of the application by more than a hundred bytes, but ensures the "correct" termination of the application, and if (your) additional code is present, it allows you to perform other actions, including prevent session termination. Note, however, that even if your application is working with documents or databases, it is not very good to ask questions of the user unnecessarily at the moment when he has already initiated shutdown. Just imagine the situation when you have a couple of dozen applications running, and when you try to log out, they all start asking a bunch of questions like "Are you

© 2024 Carl Peeraer

sure? ...". This is annoying to say the least (and in an emergency it can lead to irreparable consequences!).
In addition, completion without calling **OnDestroy** and other final handlers is by no means incorrect in all cases. There is a fairly wide class of applications (for which the KOL library was originally positioned) that can be terminated at any time. Since they do not perform modifications of data important for the user, their immediate termination does not threaten any troubles either for the user, or for the system, or for this application itself (if it is launched in the future). Even if your application saves any files or modifies the database or registry, consider immediately saving the application state without asking the user for permission: when the session ends, he may really have no time to deal with your questions.

CloseQueryReason - contains one of the reasons for the request to end the session or to close the window (closing the window by the user, turning off the power, terminating the session by the user);
SupportMnemonics - this method provides processing by the form of pressing Alt + letter key combinations for mnemonics assigned to menu items, buttons, checkboxes (check box, radio box), and calling the OnClick handlers of the corresponding elements. If the method is called for an applet, then mnemonics are processed for all forms, and for each form this method is no longer required;
KeyPreview - for a form, provides preprocessing of keystrokes (for keys pressed on the keyboard when one of the form windows owns the input focus). For this property to work, you must also include the conditional compilation symbol KEY_PREVIEW in the project options;
ActiveControl - the active (i.e., keyboard focus) child visual object of the form.

4.25.1.13.3 OnFormClick event (for form)

The **OnClick** event does not fire for the form normally. This is because **OnClick** fires in response to a selection command, which is not the same thing as a mouse click. Those. for most window controls, a mouse click causes a command to be sent to that window item or its sub-item, but generally speaking, a selection command can also be sent as a result of pressing certain keys, such as the spacebar (on a button) or mnemonic code assigned to the item. A command is a WM_COMMAND or WM_NOTIFY window message with the corresponding NM_CLICK notification code (and the same goes for menu items), as a result of which the **OnClick** event is triggered according to the rules of Windows OS. But the form, as you might guess, is not a control, and clicking on it does not send such a message.

However, it is possible to handle the **OnClick** event for a form if you assign it by assigning the address of the event handler to the OnFormClick property (as opposed to OnClick). For a form, processing of the OnClick message is performed only as a result of pressing the mouse button on it (on the area free of child visual elements).

In the case of using MCK, only the **OnClick** event for the form continues to be displayed in the Object Property Inspector, but when the code is generated by the TKOLForm object, the value is

© 2024 Carl Peeraer

assigned to the OnFormClick event - in the case of a form. That is, technically in the case of MCK, both of these events are equivalent for the form.

There is a peculiarity of using the **OnClick** event for a form, which appears when you double-click on the form. Namely, **OnClick** is called twice - once for each click. And if you assign both events (**OnformClick** and **OnMouseDblClk**), then the events will be called for each double click in the following order: **OnClick**, **OnMouseDblClk**, **OnClick**.

4.25.1.14 Modal dialogs

ShowModal - a function for showing the form and displaying it in the modal dialog mode, for the duration of this dialog all other forms of the application are prohibited, and only reacts to the mouse click - to bring the application to the foreground. The modal dialog ends when the ModalResult property of the modally displayed form is set to a nonzero value (or if the user has closed the form, if possible), after which all forbidden forms are resolved again, and execution continues from the point of call (in this case, ShowModal returns the value set in ModalResult). A form called modally can nestedly call another modal dialog (and so on up to any nesting level). A special chapter of this book will be devoted to working with modal forms in more detail.
ShowModalParented(C) - shows the form in modal mode in relation to the specified form (i.e., only the specified form is prohibited, all other displayed forms continue to be allowed to switch to them);
ShowModalEx - the same as **ShowModal**, but not only all other KOL-forms are prohibited, but also all top-level windows for a given thread of execution of commands (thread). Useful for creating modal KOL forms in a VCL application (for example, when a KOL form is launched from a DLL, see the corresponding example);
ModalResult - an integer showing the result of the modal dialog execution (0 - the dialog is still ongoing, less than zero - usually corresponds to the answer "no", in particular, the value -1 is set when the user closes the modal form, all other values can be used by the developer at his discretion);
Modal - checks that the form is shown as modal.

4.25.1.15 Reference system

HelpPath - the path string to the help file in the **WinHelp format with the .hlp extension**, to use the help **files in the HtmlHelp (* .chm) format**, you must use the **AssignHtmlHelp** global procedure;
OnHelp - an event for a form, triggered when the F1 key is pressed or help is requested by clicking on a special icon, and then a visual element on the form is clicked with the mouse. The event handler receives the context of the help call (and can change it dynamically);
CallHelp(i, C)- a form or Applet method, can be called to display help on a visual element with a given context. By default, the help file in the WinHelp format is used; to use the help in the HtmlHelp format, you must first call the AssignHtmlHelp procedure.

```
var HelpFilePath: PKOLChar;
```


© 2024 Carl Peeraer

Path to application help file. If not assigned, application path with extension replaced to '.hlp' used. To use '.chm' file (HtmlHelp), call AssignHtmlHelp with a path to a html help file (or a name).

```
procedure HtmlHelpCommand( Wnd: HWnd; const HelpFilePath: KOLString; Cmd, Data: Integer );
```
Use this wrapper procedure to call HtmlHelp API function.

4.25.2 Common Properties and Methods - Syntax

TControl(unit KOL.pas) ← **TObj** ← **_TObj**
TControl = object(**TObj**)

TControl is the basic visual object of KOL. And now, all visual objects have the same type PControl, differing only in "constructor", which during creating of object adjusts it so it can play role of desired control. Idea of encapsulating of all visual objects having the most common set of properties, is belonging to Vladimir Kladov, (C) 2000.
Since all visual objects are represented in KOL by this single object type, not all methods, properties and events defined in **TControl**, are applicable to different visual objects. See also notes about certain control kinds, located together with its constructing functions definitions.

```
type PControl = ^ TControl 203;
```
Type of pointer to TControl 203 visual object. All constructing functions New[ControlName] are returning pointer of this type. Do not forget about some difference of using objects from using classes. Identifier Self for methods of object is not of pointer type, and to pass pointer to Self, it is necessary to pass @Self instead. At the same time, to use pointer to object in 'WITH' operator, it is necessary to apply suffix '^' to pointer to get know to compiler, what do You want.

```
type TWindowFunc = function( Sender: PControl 203; var Msg: TMsg; var Rslt: Integer ): Boolean;
```
Event type to define custom extended message handlers (as pointers to procedure entry points). Such handlers are usually defined like add-ons, extending behaviour of certain controls and attached using AttachProc method of TControl 203. If the handler detects, that it is necessary to stop further message processing, it should return True.

```
type TMouseButton =( mbNone, mbLeft, mbRight, mbMiddle );
```
Available mouse buttons. mbNone is useful to get know, that there were no mouse buttons pressed.

```
type TMouseEventData = packed Record
```
Record to pass it to mouse handling routines, assigned to OnMouseXXXX events.

`Button: TMouseButton 203;`	
`StopHandling: Boolean;`	Set it to True in OnMouseXXXX event handler to
`R1, R2: Byte;`	Not used
`Shift : DWORD;`	HiWord(Shift) = zDelta in WM_MOUSEWHEEL

© 2024 Carl Peeraer

```
    Button: TMouseButton 203;
    X, Y : SmallInt;
end;
```

```
type TOnMouse = procedure( Sender: PControl 203; var Mouse: TMouseEventData 203 ) of object;
```
Common mouse handling event type.

```
type TOnKey = procedure( Sender: PControl 203; var Key: Longint; Shift: DWORD ) of object;
```
Key events. Shift is a combination of flags MK_SHIFT, MK_CONTROL, MK_ALT. (See GetShiftState 211 funtion).

```
type TOnChar = procedure( Sender: PControl 203; var Key: KOLChar; Shift: DWORD ) of object;
```
Char event. Shift is a combination of flags MK_SHIFT, MK_CONTROL, MK_ALT.

```
type TTabKey =( tkTab, tkLeftRight, tkUpDown, tkPageUpPageDn );
```
Available tabulating key groups.

```
type TTabKeys = Set of TTabKey 204;
```
Set of tabulating key groups, allowed to be used in with a control (are installed by TControl.LookTabKey property).

```
type TOnMessage = function( var Msg: TMsg; var Rslt: Integer ): Boolean of object;
```
Event type for events, which allows to extend behaviour of windowed controls descendants using add-ons.

```
type TOnEventAccept = procedure( Sender: PObj; var Accept: Boolean ) of object;
```
Event type for OnClose event.

```
type TCloseQueryReason =( qClose, qShutdown, qLogoff );
```
Request reason type to call OnClose and OnQueryEndSession.

```
type TWindowState =( wsNormal, wsMinimized, wsMaximized );
```
Avalable states of TControl 203's window object.

```
type TOnSplit = function( Sender: PControl 203; NewSize1, NewSize2: Integer ): Boolean of object;
```
Event type for OnSplit event handler, designed specially for splitter control. Event handler must return True to accept new size of previous (to splitter) control and new size of the rest of client area of parent.

```
type TTreeViewOption =( tvoNoLines, tvoLinesRoot, tvoNoButtons, tvoEditLabels,
tvoHideSel, tvoDragDrop, tvoNoTooltips, tvoCheckBoxes, tvoTrackSelect,
```


© 2024 Carl Peeraer

tvoSingleExpand, tvoInfoTip, tvoFullRowSelect, tvoNoScroll, tvoNonEvenHeight);
Tree view options.

type **TTreeViewOptions** = set of TTreeViewOption 204;
Set of tree view options.

type **TOnTVBeginDrag** = procedure(Sender: PControl 203; Item: THandle) of object;
Event type for OnTVBeginDrag event (defined for tree view control).

type **TOnTVBeginEdit** = function(Sender: PControl 203; Item: THandle): Boolean of object;
Event type for OnTVBeginEdit event (for tree view control).

type **TOnTVEndEdit** = function(Sender: PControl 203; Item: THandle; const NewTxt: KOL_String): Boolean of object;
Event type for TOnTVEndEdit event.

type **TOnTVExpanding** = function(Sender: PControl 203; Item: THandle; Expand: Boolean): Boolean of object;
Event type for TOnTVExpanding event.

type **TOnTVExpanded** = procedure(Sender: PControl 203; Item: THandle; Expand: Boolean) of object;
Event type for OnTVExpanded event.

type **TOnTVDelete** = procedure(Sender: PControl 203; Item: THandle) of object;
Event type for OnTVDelete event.

type **TOnTVSelChanging** = function(Sender: PControl 203; oldItem, newItem: THandle): Boolean of object;
When the handler returns False, selection is not changed.

type **TOnDrag** = function(Sender: PControl 203; ScrX, ScrY: Integer; var CursorShape: Integer; var Stop: Boolean): Boolean of object;
Event, called during dragging operation (it is initiated with method Drag, where callback function of type TOnDrag is passed as a parameter). Callback function receives Stop parameter True, when operation is finishing. Otherwise, it can set it to True to force finishing the operation (in such case, returning False means cancelling drag operation, True - successful drag and in this last case callback is no more called). During the operation, when input Stop value is False, callback function can control Cursor shape, and return True, if the operation can be finished successfully at the given ScrX, ScrY position. ScrX, ScrY are screen coordinates of the mouse cursor.

type **TCreateParams** = packed record

© 2024 Carl Peeraer

Record to pass it through CreateSubClass method.

```
    Caption: PKOLChar;
    Style: cardinal;
    ExStyle: cardinal;
    X, Y: Integer;
    Width, Height: Integer;
    WndParent: HWnd;
    Param: Pointer;
    WindowClass: TWndClass;
    WinClassName:    array[0.array.63]    of
    KOLChar;
end;
```

type **TTextAlign** =(taLeft, taRight, taCenter);
Text alignments available.

type **TRichTextAlign** =(raLeft, raRight, raCenter, raJustify, raInterLetter, raScaled, raGlyphs, raSnapGrid);
Text alignment styles, available for RichEdit control.

type **TVerticalAlign** =(vaTop, vaCenter, vaBottom);
Vertical alignments available.

type **TControlAlign** =(caNone, caLeft, caTop, caRight, caBottom, caClient);
Control alignments available.

type **TBitBtnOption** =(bboImageList, bboNoBorder, bboNoCaption, bboFixed, bboFocusRect);
Options available for NewBitBtn.

type **TBitBtnOptions** = set of TBitBtnOption206;
Set of options, available for NewBitBtn.

type **TGlyphLayout** =(glyphLeft, glyphTop, glyphRight, glyphBottom, glyphOver);
Layout of glyph (for NewBitBtn). Layout glyphOver means that text is drawn over glyph.

type **TOnBitBtnDraw** = function(Sender: PControl203; BtnState: Integer): Boolean of object;
Event type for **TControl.OnBitBtnDraw** event (which is called just before drawing the BitBtn). If handler returns True, there are no drawing occure. BtnState, passed to a handler, determines current button state and can be following: 0 - not pressed, 1 - pressed, 2 - disabled, 3 - focused. Value 4 is reserved for highlight state (then mouse is over it), but highlighting is provided only if property Flat is set to True (or one of events OnMouseEnter / OnMouseLeave is assigned to

© 2024 Carl Peeraer

something).

```
type TListViewStyle =( lvsIcon, lvsSmallIcon, lvsList, lvsDetail,
lvsDetailNoHeader );
```
Styles of view for ListView control (see NewListVew).

```
type TListViewItemStates = ( lvisFocus, lvisSelect, lvisBlend, lvisHighlight );
```

```
type TListViewItemState = Set of TListViewItemStates 207;
```

```
type TOnEditLVItem = function( Sender: PControl 203; Idx, Col: Integer; NewText:
PKOL_Char ): Boolean of object;
```
Event type for OnEndEditLVItem. Return True in handler to accept new text value.

```
type TOnDeleteLVItem = procedure( Sender: PControl 203; Idx: Integer ) of object;
```
Event type for OnDeleteLVItem event.

```
type TOnLVData = procedure( Sender: PControl 203; Idx, SubItem: Integer; var Txt:
KOL_String; var ImgIdx: Integer; var State: DWORD; var Store: Boolean ) of object;
```
Event type for OnLVData event. Used to provide virtual list view control (i.e. having lvoOwnerData style) with actual data on request. Use parameter Store as a flag if control should store obtained data by itself or not.

```
type TOnCompareLVItems = function( Sender: PControl 203; Idx1, Idx2: Integer ):
Integer of object;
```
Event type to compare two items of the list view (while sorting it).

```
type TOnLVColumnClick = procedure( Sender: PControl 203; Idx: Integer ) of object;
```
Event type for OnColumnClick event.

```
type TOnLVStateChange = procedure( Sender: PControl 203; IdxFrom, IdxTo: Integer;
OldState, NewState: DWORD ) of object;
```
Event type for OnLVStateChange event, called in responce to select/unselect a single item or items range in list view control).

```
type TOnLVCustomDraw = function( Sender: PControl 203; DC: HDC; Stage: DWORD; ItemIdx,
SubItemIdx: Integer; const Rect: TRect; ItemState: TDrawState 210; var TextColor,
BackColor: TColor ): DWORD of object;
```
Event type for OnLVCustomDraw event.

© 2024 Carl Peeraer

type **TWherePosLVItem** = (lvwpOnIcon, lvwpOnLabel, lvwpOnStateIcon, lvwpOnColumn, lvwpOnItem);

type **TEdgeStyle** =(esRaised, esLowered, esNone, esTransparent, esSolid);
Edge styles (for panel - see NewPanel 347). esTransparent and esSolid - special styles equivalent to esNone except GRushControls are used via USE_GRUSH symbol (ToGRush.pas)

type **TGradientStyle** =(gsVertical, gsHorizontal, gsRectangle, gsElliptic, gsRombic, gsTopToBottom, gsBottomToTop);
Gradient fill styles. See also TGradientLayout 208.

type **TGradientLayout** =(glTopLeft, glTop, glTopRight, glLeft, glCenter, glRight, glBottomLeft, glBottom, glBottomRight);
Position of starting line / point for gradient filling. Depending on TGradientStyle 208, means either position of first line of first rectangle (ellipse) to be expanded in a loop to fit entire gradient panel area.

type **TProgressbarOption** =(pboVertical, pboSmooth);
Options for progress bar.

type **TProgressbarOptions** = set of TProgressbarOption 208;
Set of options available for progress bar.

type **TEditOption** =(eoNoHScroll, eoNoVScroll, eoLowercase, eoMultiline, eoNoHideSel, eoOemConvert, eoPassword, eoReadonly, eoUpperCase, eoWantReturn, eoWantTab, eoNumber);
Available edit options.
Please note, that **eoWantTab** option just removes TAB key from a list of keys available to tabulate from the edit control. To provide insertion of tabulating key, do so in TControl.OnChar 273 event handler. Sorry for inconvenience, but this is because such behavior is not must in all cases. See also TControl.EditTabChar 251 property.

type **TEditOptions** = Set of TEditOption 208;
Set of available edit options.

type **TRichFmtArea** =(raSelection, raWord, raAll);
Characters formatting area for RichEdit.

type **TRETextFormat** =(reRTF, reText, rePlainRTF, reRTFNoObjs, rePlainRTFNoObjs, reTextized, reUnicode, reTextUnicode);
Available formats for transfer RichEdit text using property TControl.RE_Text 244.

```
reRTF                    - normal rich text (no transformations)
```


© 2024 Carl Peeraer

```
reText                     - plain text only (without OLE objects)
reTextized                 - plain text with text representation of COM objects
rePlainRTF                 - reRTF without language-specific keywords
reRTFNoObjs                - reRTF without OLE objects
rePlainRTFNoObjs           - rePlainRTF without OLE objects
reUnicode                  - stream is 2-byte Unicode characters rather then 1-byte
                           Ansi
```

type **TRichUnderline** =(ruSingle, ruWord, ruDouble, ruDotted, ruDash, ruDashDot, ruDashDotDot, ruWave, ruThick, ruHairLine);
Rich text exteded underline styles (available only for RichEdit v2.0, and even for RichEdit v2.0 additional styles can not displayed - but ruDotted under Windows2000 is working).

type **TRichTextSizes** =(rtsNoUseCRLF, rtsNoPrecise, rtsClose, rtsBytes);
Options to calculate size of rich text. Available only for RichEdit2.0 or higher.

type **TRichTextSize** = set of TRichTextSizes209;
Set of all available optioins to calculate rich text size using property TControl.RE_TextSize238 [options].

type **TRichNumbering** =(rnNone, rnBullets, rnArabic, rnLLetter, rnULetter, rnLRoman, rnURoman);
Advanced numbering styles for paragraph (RichEdit).

```
rnNone                     - no numbering
rnBullets                  - bullets only
rnArabic                   - 1, 2, 3, 4, ...
rnLLetter                  - a, b, c, d, ...
rnULetter                  - A, B, C, D, ...
rnLRoman                   - i, ii, iii, iv, ...
rnURoman                   - I, II, III, IV, ...
rnNoNumber                 - do not show any numbers (but numbering is taking place).
```

type **TRichNumBrackets** =(rnbRight, rnbBoth, rnbPeriod, rnbPlain, rnbNoNumber);
Brackets around number:

```
rnbRight                   - 1) 2) 3)     - this is default !
rnbBoth                    - (1) (2) (3)
rnbPeriod                  - 1. 2. 3.
rnbPlain                   - 1 2 3
```

type **TBorderEdge** =(beLeft, beTop, beRight, beBottom);
Borders of rectangle.

type **TOnTestMouseOver** = function(Sender: PControl): Boolean of object;
Event type for TControl.OnTestMouseOver event. The handler should return True, if it detects if the mouse is over control.

© 2024 Carl Peeraer

type **TDrawStates** =(odsSelected, odsGrayed, odsDisabled, odsChecked, odsFocused, odsDefault, odsHotlist, odsInactive, odsNoAccel, odsNoFocusRect, ods400reserved, ods800reserved, odsComboboxEdit, odsMarked, odsIndeterminate);
Possible draw states.

odsSelected	- The menu item's status is selected.
odsGrayed	- The item is to be grayed. This bit is used only in a menu.
odsDisabled	- The item is to be drawn as disabled.
odsChecked	- The menu item is to be checked. This bit is used only in a menu.
odsFocused	- The item has the keyboard focus.
odsDefault	- The item is the default item.
odsHotList	- Windows 98, Windows 2000: The item is being hot-tracked, that is, the item will be highlighted when the mouse is on the item.
odsInactive	- Windows 98, Windows 2000: The item is inactive and the window associated with the menu is inactive.
odsNoAccel	- Windows 2000: The control is drawn without the keyboard accelerator cues.
odsNoFocusRect	- Windows 2000: The control is drawn without focus indicator cues.
odsComboboxEdit	- The drawing takes place in the selection field (edit control) of an owner-drawn combo box.
odsMarked	- for Common controls only. The item is marked. The meaning of this is up to the implementation.
odsIndeterminate	- for Common Controls only. The item is in an indeterminate state.

type **TDrawState** = Set of TDrawStates 210;
Set of possible draw states.

type **TOnDrawItem** = function(Sender: PObj 92; DC: HDC; const Rect: TRect; ItemIdx: Integer; DrawAction: TDrawAction; ItemState: TDrawState 210): Boolean of object;
Event type for OnDrawItem event (applied to list box, combo box, list view).

type **TOnMeasureItem** = function(Sender: PObj 92; Idx: Integer): Integer of object;
Event type for OnMeasureItem event. The event handler must return height of list box item as a result.

type **TListOption** =(loNoHideScroll, loNoExtendSel, loMultiColumn, loMultiSelect, loNoIntegralHeight, loNoSel, loSort, loTabstops, loNoStrings, loNoData, loOwnerDrawFixed, loOwnerDrawVariable, loHScroll);
Options for ListBox (see NewListbox 358). To use loHScroll, you also have to send LB_SETHORIZONTALEXTENT with a maximum width of a line in pixels (wParam)!

type **TListOptions** = Set of TListOption 210;
Set of available options for Listbox.

type **TComboOption** =(coReadOnly, coNoHScroll, coAlwaysVScroll, coLowerCase,

© 2024 Carl Peeraer

coNoIntegralHeight, coOemConvert, coSort, coUpperCase, coOwnerDrawFixed, coOwnerDrawVariable, coSimple);
Options for combobox.

type **TComboOptions** = Set of TComboOption 210;
Set of options available for combobox.

type **TToolbarOption** =(tboTextRight, tboTextBottom, tboFlat, tboTransparent, tboWrapable, tboNoDivider, tbo3DBorder, tboCustomErase);
Toolbar options. When tboFlat is set and toolbar is placed onto panel, set its property Transparent to TRUE to provide its correct view.

type **TToolbarOptions** = Set of TToolbarOption 211;
Set of toolbar options.

type **TOnToolbarButtonClick** = procedure(Sender: PControl 203; BtnID: Integer) of object;
Special event type to handle separate toolbar buttons click events.

type **TOnTBCustomDraw** = function(Sender: PControl 203; var NMCD: TNMTBCustomDraw): Integer of object;
Event type for OnTBCustomDraw event.

type **TTabControlOption** =(tcoButtons, tcoFixedWidth, tcoFocusTabs, tcoIconLeft, tcoLabelLeft, tcoMultiline, tcoMultiselect, tcoFitRows, tcoScrollOpposite, tcoBottom, tcoVertical, tcoFlat, tcoHotTrack, tcoBorder, tcoOwnerDrawFixed);
Options, available for TabControl.

type **TTabControlOptions** = set of TTabControlOption 211;
Set of options, available for TAbControl during its creation (by NewTabControl 366 function).

type **TDateTimePickerOption** =(dtpoTime, dtpoDateLong, dtpoUpDown, dtpoRightAlign, dtpoShowNone, dtpoParseInput);

type **TDateTimePickerOptions** = set of TDateTimePickerOption 211;

function **GetShiftState**: DWORD;
Returns shift state.

© 2024 Carl Peeraer

TControl properties

`property Parent: PControl;`
Parent of TParent object. Also must be of TParent type or derived from TParent.

`property Enabled: Boolean;`
Enabled usually used to decide if control can get keyboard focus or been clicked by mouse.

`property Visible: Boolean;`
Obvious.

`property ToBeVisible: Boolean;`
Returns True, if a control is supposed to be visible when its form is showing.

`property CreateVisible: Boolean;`
False by default. If You want your form to be created visible and flick due creation, set it to True. This does not affect size of executable anyway.

`property BoundsRect: TRect;`
Bounding rectangle of the visual. Coordinates are relative to top left corner of parent's ClientRect 248, or to top left corner of screen (for TForm).

`property Left: Integer;`
Left horizontal position.

`property Top: Integer;`
Top vertical position.

`property Width: Integer;`
Width of TVisual object.

`property Height: Integer;`
Height of TVisual object.

`property Position: TPoint;`
Represents top left position of the object. See also BoundsRect 212.

`property MinWidth: SmallInt;`
Minimal width constraint.

© 2024 Carl Peeraer

`property` **`MinHeight:`** `SmallInt;`
Minimal height constraint.

`property` **`MaxWidth:`** `SmallInt;`
Maximal width constraint.

`property` **`MaxHeight:`** `SmallInt;`
Maximal height constraint.

`property` **`ClientWidth:`** `Integer;`
Obvious. Accessing this property, program forces window latent creation.

`property` **`ClientHeight:`** `Integer;`
Obvious. Accessing this property, program forces window latent creation.

`property` **`Windowed:`** `Boolean;`
Constantly returns True, if object is windowed (i.e. owns correspondent window handle). Otherwise, returns False.
By now, all the controls are windowed (there are no controls in KOL, which are emulating window, acually belonging to Parent 212 - like TGraphicControl in VCL).
Writing of this property provided only for internal purposes, do not change it directly unless you understand well what you do.

`property` **`ChildCount:`** `Integer;`
Returns number of commonly accessed child objects.

`property` **`Children`**`[ Idx: Integer ]: PControl;`
Child items of TVisual object. Property is reintroduced here to separate access to always visible Children[] from restricted a bit Members[].

`property` **`WindowedParent:`** `PControl;`
Returns nearest windowed parent, the same as Parent 212.

`property` **`ActiveControl:`** `PControl;`

`property` **`Handle:`** `HWnd;`
Returns descriptor of system window object. If window is not yet created, 0 is returned. To allocate handle, call CreateWindow 250 method.

`property` **`ParentWindow:`** `HWnd;`
Returns handle of parent window (not TControl object, but system window object handle).

© 2024 Carl Peeraer

```
property ClsStyle: DWord;
```

Window class style. Available styles are:

CS_BYTEALIGNCLIENT	Aligns the window's client area on the byte boundary (in the x direction) to enhance performance during drawing operations.
CS_BYTEALIGNWINDOW	Aligns a window on a byte boundary (in the x direction).
CS_CLASSDC	Allocates one device context to be shared by all windows in the class.
CS_DBLCLKS	Sends double-click messages to the window procedure when the user double-clicks the mouse while the cursor is within a window belonging to the class.
CS_GLOBALCLASS	Allows an application to create a window of the class regardless of the value of the hInstance parameter. You can create a global class by creating the window class in a dynamic-link library (DLL) and listing the name of the DLL in the registry under specific keys.
CS_HREDRAW	Redraws the entire window if a movement or size adjustment changes the width of the client area.
CS_NOCLOSE	Disables the Close 250 command on the System menu.
CS_OWNDC	Allocates a unique device context for each window in the class.
CS_PARENTDC	Sets the clipping region of the child window to that of the parent window so that the child can draw on the parent.
CS_SAVEBITS	Saves, as a bitmap, the portion of the screen image obscured by a window. Windows uses the saved bitmap to re-create the screen image when the window is removed.
CS_VREDRAW	Redraws the entire window if a movement or size adjustment changes the height of the client area.

For more info, see Win32.hlp (keyword 'WndClass');

```
property Style: DWord;
```

Window styles. Available styles are:

WS_BORDER	Creates a window that has a thin-line border.
WS_CAPTION	Creates a window that has a title bar (includes the WS_BORDER style).
WS_CHILD	Creates a child window. This style cannot be used with the WS_POPUP style.
WS_CHILDWINDOW	Same as the WS_CHILD style.

© 2024 Carl Peeraer

WS_BORDER	Creates a window that has a thin-line border.
WS_CLIPCHILDREN	Excludes the area occupied by child windows when drawing occurs within the parent window. This style is used when creating the parent window.
WS_CLIPSIBLINGS	Clips child windows relative to each other; that is, when a particular child window receives a WM_PAINT message, the WS_CLIPSIBLINGS style clips all other overlapping child windows out of the region of the child window to be updated. If WS_CLIPSIBLINGS is not specified and child windows overlap, it is possible, when drawing within the client area of a child window, to draw within the client area of a neighboring child window.
WS_DISABLED	Creates a window that is initially disabled. A disabled window cannot receive input from the user.
WS_DLGFRAME	Creates a window that has a border of a style typically used with dialog boxes. A window with this style cannot have a title bar.
WS_GROUP	Specifies the first control of a group of controls. The group consists of this first control and all controls defined after it, up to the next control with the WS_GROUP style. The first control in each group usually has the WS_TABSTOP style so that the user can move from group to group. The user can subsequently change the keyboard focus from one control in the group to the next control in the group by using the direction keys.
WS_HSCROLL	Creates a window that has a horizontal scroll bar.
WS_ICONIC	Creates a window that is initially minimized. Same as the WS_MINIMIZE style.
WS_MAXIMIZE	Creates a window that is initially maximized.
WS_MAXIMIZEBOX	Creates a window that has a Maximize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified.
WS_MINIMIZE	Creates a window that is initially minimized. Same as the WS_ICONIC style.
WS_MINIMIZEBOX	Creates a window that has a Minimize button. Cannot be combined with the WS_EX_CONTEXTHELP style. The WS_SYSMENU style must also be specified.
WS_OVERLAPPED	Creates an overlapped window. An overlapped window has a title bar and a border. Same as the WS_TILED style.
WS_OVERLAPPEDWINDOW	Creates an overlapped window with the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME,

© 2024 Carl Peeraer

WS_BORDER	Creates a window that has a thin-line border.
	WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles. Same as the WS_TILEDWINDOW style.
WS_POPUP	Creates a pop-up window. This style cannot be used with the WS_CHILD style.
WS_POPUPWINDOW	Creates a pop-up window with WS_BORDER, WS_POPUP, and WS_SYSMENU styles. The WS_CAPTION and WS_POPUPWINDOW styles must be combined to make the window menu visible.
WS_SIZEBOX	Creates a window that has a sizing border. Same as the WS_THICKFRAME style.
WS_SYSMENU	Creates a window that has a window-menu on its title bar. The WS_CAPTION style must also be specified.
WS_TABSTOP	Specifies a control that can receive the keyboard focus when the user presses the TAB key. Pressing the TAB key changes the keyboard focus to the next control with the WS_TABSTOP style.
WS_THICKFRAME	Creates a window that has a sizing border. Same as the WS_SIZEBOX style.
WS_TILED	Creates an overlapped window. An overlapped window has a title bar and a border. Same as the WS_OVERLAPPED style.
WS_TILEDWINDOW	Creates an overlapped window with the WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, and WS_MAXIMIZEBOX styles. Same as the WS_OVERLAPPEDWINDOW style.
WS_VISIBLE	Creates a window that is initially visible.
WS_VSCROLL	Creates a window that has a vertical scroll bar.

See also Win32.hlp (topic CreateWindow 250).

```
property ExStyle: DWord;
```

Extra window styles. Available flags are following:

WS_EX_ACCEPTFILES	Specifies that a window created with this style accepts drag-drop files.
WS_EX_APPWINDOW	Forces a top-level window onto the taskbar when the window is minimized.
WS_EX_CLIENTEDGE	Specifies that a window has a border with a sunken edge.
WS_EX_CONTEXTHELP	Includes a question mark in the title bar of the window. When the user clicks the question mark, the cursor changes

© 2024 Carl Peeraer

WS_EX_ACCEPTFILES	Specifies that a window created with this style accepts drag-drop files.
	to a question mark with a pointer. If the user then clicks a child window, the child receives a WM_HELP message. The child window should pass the message to the parent window procedure, which should call the WinHelp function using the HELP_WM_HELP command. The Help application displays a pop-up window that typically contains help for the child window.WS_EX_CONTEXTHELP cannot be used with the WS_MAXIMIZEBOX or WS_MINIMIZEBOX styles.
WS_EX_CONTROLPARENT	Allows the user to navigate among the child windows of the window by using the TAB key.
WS_EX_DLGMODALFRAME	Creates a window that has a double border; the window can, optionally, be created with a title bar by specifying the WS_CAPTION style in the dwStyle parameter.
WS_EX_LEFT	Window has generic "left-aligned" properties. This is the default.
WS_EX_LEFTSCROLLBAR	If the shell language is Hebrew, Arabic, or another language that supports reading order alignment, the vertical scroll bar (if present) is to the left of the client area. For other languages, the style is ignored and not treated as an error.
WS_EX_LTRREADING	The window text is displayed using Left 212 to Right reading-order properties. This is the default.
WS_EX_MDICHILD	Creates an MDI child window.
WS_EX_NOPARENTNOTIFY	Specifies that a child window created with this style does not send the WM_PARENTNOTIFY message to its parent window when it is created or destroyed.
WS_EX_OVERLAPPEDWINDOW	Combines the WS_EX_CLIENTEDGE and WS_EX_WINDOWEDGE styles.
WS_EX_PALETTEWINDOW	Combines the WS_EX_WINDOWEDGE, WS_EX_TOOLWINDOW, and WS_EX_TOPMOST styles.
WS_EX_RIGHT	Window has generic "right-aligned" properties. This depends on the window class. This style has an effect only if the shell language is Hebrew, Arabic, or another language that supports reading order alignment; otherwise, the style is ignored and not treated as an error.
WS_EX_RIGHTSCROLLBAR	Vertical scroll bar (if present) is to the right of the client area. This is the default.
WS_EX_RTLREADING	If the shell language is Hebrew, Arabic, or another language that supports reading order alignment, the window text is displayed using Right to Left 212 reading-order properties.

© 2024 Carl Peeraer

WS_EX_ACCEPTFILES	Specifies that a window created with this style accepts drag-drop files.
	For other languages, the style is ignored and not treated as an error.
WS_EX_STATICEDGE	Creates a window with a three-dimensional border style intended to be used for items that do not accept user input.
WS_EX_TOOLWINDOW	Creates a tool window; that is, a window intended to be used as a floating toolbar. A tool window has a title bar that is shorter than a normal title bar, and the window title is drawn using a smaller font. A tool window does not appear in the taskbar or in the dialog that appears when the user presses ALT+TAB.
WS_EX_TOPMOST	Specifies that a window created with this style should be placed above all non-topmost windows and should stay above them, even when the window is deactivated. To add or remove this style, use the SetWindowPos function.
WS_EX_TRANSPARENT	Specifies that a window created with this style is to be transparent. That is, any windows that are beneath the window are not obscured by the window. A window created with this style receives WM_PAINT messages only after all sibling windows beneath it have been updated.
WS_EX_WINDOWEDGE	Specifies that a window has a border with a raised edge.

See also Win32.hlp (topic CreateWindowEx).

`property` **`Cursor`**`: HCursor;`
Current cursor. For most of controls, sets initially to IDC_ARROW. See also ScreenCursor.

`property` **`Icon`**`: HIcon;`
Icon. By default, icon of the Applet is used. To load icon from the resource, use IconLoad 250 or IconLoadCursor 250 method - this is more correct, because in such case a special flag is set to prevent attempts to destroy shared icon object in the destructor of the control.

`property` **`Menu`**`: HMenu;`
Menu (or ID of control - for standard GUI controls).

`property` **`HelpContext`**`: Integer;`
Help context.

`property` **`HelpPath`**`: KOLString;`
Property of a form or an Applet. Change it to provide custom path to WinHelp format help file. If HtmlHelp used, call global procedure AssignHtmlHelp instead.

© 2024 Carl Peeraer

```
property Caption: KOLString;
```
Caption of a window. For standard Windows buttons, labels and so on not a caption of a window, but text of the window.

```
property Text: KOLString;
```
The same as Caption 219. To make more convenient with Edit controls. For Rich Edit control, use property RE_Text 244.

```
property SelStart: Integer;
```
Start of selection (editbox - character position).

```
property SelLength: Integer;
```
Length of selection (editbox - number of characters selected, multiselect listbox or listview - number of items selected).
Note, that for combobox and single-select listbox it always returns 0 (though for single-select listview, returns 1, if there is an item selected).
It is possible to set SelLength only for memo and richedit controls.

```
property Selection: KOLString;
```
Selected text (editbox, richedit) as string. Can be useful to replace selection. For rich edit, use RE_Text 244[reText, TRUE], if you want to read correctly characters from another locale then ANSI only.

```
property CurIndex: Integer;
```
Index of current item (for listbox, combobox) or button index pressed or dropped down (for toolbar button, and only in appropriate event handler call).
You cannot use it to set or remove a selection in a multiple-selection list box, so you should set option loNoExtendSel to true.
In OnClick 271 event handler, CurIndex has not yet changed for listbox or combobox. Use OnSelChange 272 to respond to selection changes.

```
property Count: Integer;
```
Number of items (listbox, combobox, listview) or lines (multiline editbox, richedit control) or buttons (toolbar). It is possible to assign a value to this property only for listbox control with loNoData style and for list view control with lvoOwnerData style (virtual list box and list view).

```
property Items[ Idx: Integer ]: KOLString;
```
Obvious. Used with editboxes, listbox, combobox. With list view, use property LVItems 231 instead.

© 2024 Carl Peeraer

`property` **`ItemSelected`**`[ ItemIdx: Integer ]: Boolean;`
Returns True, if a line (in editbox) or an item (in listbox, combobox, listview) is selected. Can be set only for listboxes. For listboxes, which are not multiselect, and for combo lists, it is possible only to set to True, to change selection.

`property` **`ItemData`**`[ Idx: Integer ]: DWORD;`
Access to user-defined data, associated with the item of a list box and combo box.

`property` **`DroppedWidth`**`: Integer;`
Allows to change width of dropped down items list for combobox (only!) control.

`property` **`DroppedDown`**`: Boolean;`
Dropped down state for combo box. Set it to TRUE or FALSE to change dropped down state.

`property` **`BitBtnDrawMnemonic`**`: Boolean;`
Set this property to TRUE to provide correct drawing of bit btn control caption with '&' characters (to remove such characters, and underline follow ones).

`property` **`TextShiftX`**`: Integer;`
Horizontal shift for bitbtn text when the bitbtn is pressed.

`property` **`TextShiftY`**`: Integer;`
Vertical shift for bitbtn text when the bitbtn is pressed.

`property` **`BitBtnImgIdx`**`: Integer;`
BitBtn image index for the first image in list view, used as bitbtn image. It is used only in case when BitBtn is created with bboImageList option.

`property` **`BitBtnImgList`**`: THandle;`
BitBtn Image list. Assign image list handle to change it.

`property` **`DefaultBtn`**`: Boolean;`
Set this property to true to make control clicked when ENTER key is pressed. This property uses OnMessage 270 event of the parent form, storing it into fOldOnMessage field and calling in chain. So, assign default button after setting OnMessage 270 event for the form.

`property` **`CancelBtn`**`: Boolean;`
Set this property to true to make control clicked when escape key is pressed. This property uses OnMessage 270 event of the parent form, storing it into fOldOnMessage field and calling in chain. So, assign cancel button after setting OnMessage 270 event for the form.

© 2024 Carl Peeraer

```
property IgnoreDefault: Boolean;
```
Change this property to TRUE to ignore default button reaction on press ENTER key when a focus is grabbed of the control. Default value is different for different controls. By default, DefaultBtn 220 ignored in memo, richedit (even if read-only).

```
property Color: TColor;
```
Property Color is one of the most common for all visual elements (like form, control etc.) Please note, that standard GUI button can not change its color and the most characteristics of the Font 221. Also, standard button can not become Transparent 224. Use bitbtn for such purposes. Also, changing Color property for some kinds of control has no effect (rich edit, list view, tree view, etc.). To solve this, use native (for such controls) color property, or call Perform 265 method with appropriate message to set the background color.

```
property Font: PGraphicTool;
```
If the Font property is not accessed, correspondent TGraphicTool object is not created and its methods are not included into executable. Leaving properties Font and Brush 221 untouched can economy executable size a lot.

```
property Brush: PGraphicTool;
```
If not accessed, correspondent TGraphicTool object is not created and its methods are not referenced. See also note on Font 221 property.

```
property Ctl3D: Boolean;
```
Inheritable from parent controls to child ones.

```
property ModalResult: Integer;
```
Modal 221 result. Set it to value<>0 to stop modal dialog. By agreement, value 1 corresponds 'OK', 2 - 'Cancel'. But it is totally by decision of yours how to interpret this value.

```
property Modal: Boolean;
```
TRUE, if the form is shown modal.

```
property ModalForm: PControl;
```
Form currently shown modal from this form or from Applet.

```
property WindowState: TWindowState;
```
Window state.

```
property Canvas: PCanvas;
```
Placeholder for Canvas: PCanvas. But in KOL, it is possible to create applets without canvases at all. To do so, avoid using Canvas and use DC directly (which is passed in OnPaint 271 event).

© 2024 Carl Peeraer

```
property IsApplet: Boolean;
```
Returns true, if the control is created using NewApplet (or CreateApplet).

```
property IsForm: Boolean;
```
Returns True, if the object is form window.

```
property IsMDIChild: Boolean;
```
Returns TRUE, if the object is MDI child form. In such case, IsForm 222 also returns TRUE.

```
property IsControl: Boolean;
```
Returns True, is the control is control (not form or applet).

```
property IsButton: Boolean;
```
Returns True, if the control is button-like or containing buttons (button, bitbtn, checkbox, radiobox, toolbar).

```
property HasBorder: Boolean;
```
Obvious. Form-aware.

```
property HasCaption: Boolean;
```
Obvious. Form-aware.

```
property CanResize: Boolean;
```
Obvious. Form-aware.

```
property StayOnTop: Boolean;
```
Obvious. Form-aware, but can be applied to controls.

```
property Border: ShortInt;
```
Distance between edges and child controls and between child controls by default (if methods PlaceRight 263, PlaceDown 263, PlaceUnder 263, ResizeParent 264, ResizeParentRight 264, ResizeParentBottom 264 are called).
Originally was named Margin 222, now I recommend to use the name 'Border' to avoid confusion with MarginTop 223, MarginBottom 223, MarginLeft 223 and MarginRight 223 properties.
Initial value is always 2. Border property is used in realigning child controls (when its Align 245 property is not caNone), and value of this property determines size of borders between edges of children and its parent and between aligned controls too.
See also properties MarginLeft 223, MarginRight 223, MarginTop 223, MarginBottom 223.

```
property Margin: ShortInt;
```


© 2024 Carl Peeraer

Old name for property Border 222.

```
property MarginTop: ShortInt;
```
Additional distance between true window client top and logical top of client rectangle. This value is added to Top 212 of rectangle, returning by property ClientRect 248. Together with other margins and property Border 222, this property allows to change view of form for case, that Align 245 property is used to align controls on parent (it is possible to provide some distance from child controls to its parent, and between child controls.
Originally this property was introduced to compensate incorrect ClientRect 248 property, calculated for some types of controls.
See also properties Border 222, MarginBottom 223, MarginLeft 223, MarginRight 223.

```
property MarginBottom: ShortInt;
```
The same as MarginTop 223, but a distance between true window Bottom of client rectangle and logical bottom one. Take in attention, that this value should be POSITIVE to make logical bottom edge located above true edge.
See also properties Border 222, MarginTop 223, MarginLeft 223, MarginRight 223.

```
property MarginLeft: ShortInt;
```
The same as MarginTop 223, but a distance between true window Left 212 of client rectangle and logical left edge.
See also properties Border 222, MarginTop 223, MarginRight 223, MarginBottom 223.

```
property MarginRight: ShortInt;
```
The same as MarginLeft 223, but a distance between true window Right of client rectangle and logical bottom one. Take in attention, that this value should be POSITIVE to make logical right edge located left of true edge.
See also properties Border 222, MarginTop 223, MarginLeft 223, MarginBottom 223.

```
property Tabstop: Boolean;
```
True, if control can be focused using tabulating between controls. Set it to False to make control unavailable for keyboard, but only for mouse.

```
property TabOrder: SmallInt;
```
Order of tabulating of controls. Initially, TabOrder is equal to creation order of controls. If TabOrder changed, TabOrder of all controls with not less value of one is shifted up. To place control before another, assign TabOrder of one to another. For example:
```
Button1.TabOrder := EditBox1.TabOrder;
```
In code above, Button1 is placed just before EditBox1 in tabulating order (value of TabOrder of EditBox1 is incremented, as well as for all follow controls).

```
property Focused: Boolean;
```


© 2024 Carl Peeraer

True, if the control is current on form (but check also, what form itself is focused). For form it is True, if the form is active (i.e. it is foreground and capture keyboard). Set this value to True to make control current and focused (if applicable).

`property **TextAlign**: TTextAlign;`
Text 219 horizontal alignment. Applicable to labels, buttons, multi-line edit boxes, panels.

`property **VerticalAlign**: TVerticalAlign;`
Text 219 vertical alignment. Applicable to buttons, labels and panels.

`property **WordWrap**: Boolean;`
TRUE, if this is a label, created using NewWordWrapLabel.

`property **ShadowDeep**: Integer;`
Deep of a shadow (for label effect only, created calling NewLabelEffect 347).

`property **CannotDoubleBuf**: Boolean;`

`property **DoubleBuffered**: Boolean;`
Set it to true for some controls, which are flickering in repainting (like label effect). Slow, and requires additional code. This property is inherited by all child controls.
Note: RichEdit control can not become DoubleBuffered.

`property **Transparent**: Boolean;`
Set it to true to get special effects. Transparency also uses DoubleBuffered 224 and inherited by child controls.
Please note, that some controls can not be shown properly, when Transparent is set to True for it. If You want to make edit control transparent (e.g., over gradient filled panel), handle its OnChanged property and call there Invalidate 248 to provide repainting of edit control content. Note also, that for RichEdit control property Transparent has no effect (as well as DoubleBuffered 224). But special property RE_Transparent 245 is designed especially for RichEdit control (it works fine, but with great number of flicks while resizing of a control). Another note is about Edit control. To allow editing of transparent edit box, it is necessary to invalidate it for every pressed character. Or, use Ed_Transparent 224 property instead.

`property **Ed_Transparent**: Boolean;`
Use this property for editbox to make it really Transparent 224. Remember, that though Transparent 224 property is inherited by child controls from its parent, this is not so for Ed_Transparent. So, it is necessary to set Ed_Transparent to True for every edit control explicitly.

`property **AlphaBlend**: Byte;`

© 2024 Carl Peeraer

If assigned to 0..254, makes window (form or control) semi-transparent (Win2K only). Depending on value assigned, it is possible to adjust transparency level (0 - totally transparent, 255 - totally opaque).
Note: from XP, any control can be alpha blended!

`property` **LookTabKeys**`: TTabKeys;`
Set of keys which can be used as tabulation keys in a control.

`property` **SubClassName**`: KOLString;`
Name of window class - unique for every window class in every run session of a program.

`property` **CloseQueryReason**`: TCloseQueryReason;`
Reason why OnClose 270 or OnQueryEndSession 270 called.

`property` **UpdateRgn**`: HRgn;`
A handle of update region. Valid only in OnPaint 271 method. You can use it to improve painting (for speed), if necessary. When UpdateRgn is obtained in response to WM_PAINT message, value of the property EraseBackground 225 is used to pass it to the API function GetUpdateRgn. If UpdateRgn = 0, this means that entire window should be repainted. Otherwise, You (e.g.) can check if the rectangle is in clipping region using API function RectInRegion.

`property` **EraseBackground**`: Boolean;`
This value is used to pass it to the API function GetUpdateRgn, when UpadateRgn property is obtained first in responce to WM_PAINT message. If EraseBackground is set to True, system is responsible for erasing background of update region before painting. If not (default), the entire region invalidated should be painted by your event handler.

`property` **RightClick**`: Boolean;`
Use this property to determine which mouse button was clicked (applicable to toolbar in the OnClick 271 event handler).

`property` **MinSizePrev**`: Integer;`
Minimal allowed (while dragging splitter) size of previous control for splitter (see NewSplitter 348).

`property` **SplitMinSize1**`: Integer;`
The same as MinSizePrev 225

`property` **MinSizeNext**`: Integer;`
Minimal allowed (while dragging splitter) size of the rest of parent of splitter or of SecondControl 226 (see NewSplitter 348).

© 2024 Carl Peeraer

property **SplitMinSize2**: Integer;
The same as MinSizeNext 225.

property **SecondControl**: PControl;
Second control to check (while dragging splitter) if its size not less than SplitMinSize2 226 (see NewSplitter). By default, second control is not necessary, and needed only in rare case when SecondControl can not be determined automatically to restrict splitter right (bottom) position.

property **Dragging**: Boolean;
True, if splitter control is dragging now by user with left mouse button. Also, this property can be used to detect if the control is dragging with mouse (after calling DragStartEx 255 method).

property **ThreeButtonPress**: Boolean;
GDK (*nix) only. TRUE, if 3 button press detected. Check this flag in OnMouseDblClk 273 event handler. If 3rd button click is done for a short period of time after the double click, the control receives OnMouseDblClk 273 the second time and this flag is set. (Applicable to the GDK and other Linux systems).

property **MouseInControl**: Boolean;
This property can return True only if OnMouseEnter 273 / OnMouseLeave 273 event handlers are set for a control (or, for BitBtn, property Flat 226 is set to True. Otherwise, False is returned always.

property **Flat**: Boolean;
Set it to True for BitBtn, to provide either flat border for a button or availability of "highlighting" (correspondent to glyph index 4).
Note: this can work incorrectly a bit under win95 without comctl32.dll updated. Therefore, application will launch. To enforce correct working even under Win95, use your own timer, which event handler checks for mouse over bitbtn control, e.g.:

```
procedure TForm1.Timer1Timer(Sender: PObj);
var P: TPoint;
begin
  if not BitBtn1.MouseInControl 226 then Exit;
  GetCursorPos( P );
  P := BitBtn1.Screen2Client( P );
  if not PtInRect( BitBtn1.ClientRect, P ) then
  begin
    BitBtn1.Flat := FALSE;
    BitBtn1.Flat := TRUE;
  end;
end;
```

property **RepeatInterval**: Integer;
If this property is set to non-zero, it is interpreted (for BitBtn only) as an interval in milliseconds between repeat button down events, which are generated after first mouse or button click and

 © 2024 Carl Peeraer

until button is released. Though, if the button is pressed with keyboard (with space key), RepeatInterval value is ignored and frequency of repeatitive clicking is determined by user keyboard settings only.

`property` **`Progress`**`: Integer index(( PBM_SETPOS or $8000 ) shl 16 ) or PBM_GETPOS;`
Only for ProgressBar.

`property` **`MaxProgress`**`: Integer index(( PBM_SETRANGE32 or $8000 ) shl 16 ) or PBM_GETRANGE;`
Only for ProgressBar. 100 is the default value.

`property` **`ProgressColor`**`: TColor;`
Only for ProgressBar.

`property` **`ProgressBkColor`**`: TColor;`
Obsolete. Now the same as Color221.

`property` **`StatusText`**`[ Idx: Integer ]: KOLString;`
Only for forms to set/retrieve status text to/from given status panel. Panels are enumerated from 0 to 254, 255 is to indicate simple status bar. Size264 grip in right bottom corner of status window is displayed only if form still CanResize222.
When a status text is set first time, status bar window is created (always aligned to bottom), and form is resizing to preset client height. While status bar is showing, client height value is returned without height of status bar. To remove status bar, call RemoveStatus256 method for a form.
By default, text is left-aligned within the specified part of a status window. You can embed tab characters (#9) in the text to center or right-align it. Text219 to the right of a single tab character is centered, and text to the right of a second tab character is right-aligned.
If You use separate status bar onto several panels, these automatically align its widths to the same value (width divided to number of panels). To adjust status panel widths for every panel, use property StatusPanelRightX228.

`property` **`SimpleStatusText`**`: KOLString;`
Only for forms to set/retrive status text to/from simple status bar. Size264 grip in right bottom corner of status window is displayed only if form CanResize222.
When status text set first time, (simple) status bar window is created (always aligned to bottom), and form is resizing to preset client height. While status bar is showing, client height value is returned without height of status bar. To remove status bar, call RemoveStatus256 method for a form.
By default, text is left-aligned within the specified part of a status window. You can embed tab characters (#9) in the text to center or right-align it. Text219 to the right of a single tab character is centered, and text to the right of a second tab character is right-aligned.

© 2024 Carl Peeraer

```
property StatusCtl: PControl;
```
Pointer to Status bar control. To "create" child controls on the status bar, first create it as a child of form, for instance, and then change its property Parent 212, e.g.:

```
var Progress1: PControl;
...
Progress1 := NewProgressBar( Form1 );
Progress1.Parent := Form1.StatusCtl;
```
(If you use MCK, code should be another a bit, and in this case it is possible to create and adjust the control at design-time, and at run-time change its parent control. E.g. (Progress1 is created at run-time here too):
```
Progress1 := NewProgressBar( Form );
Progress1.Parent := Form.StatusCtl;
```
).

Do not forget to provide StatusCtl to be existing first (e.g. assign one-space string to SimpleStatusText 227 property of the form, for MCK do so using Object Inspector). Please note that not only a form can have status bar but any other control too!

```
property SizeGrip: Boolean;
```
Size 264 grip for status bar. Has effect only before creating window.

```
property StatusPanelRightX[ Idx: Integer ]: Integer;
```
Use this property to adjust status panel right edges (if the status bar is divided onto several subpanels). If the right edge for the last panel is set to -1 (by default) it is expanded to the right edge of a form window. Otherwise, status bar can be shorter then form width.

```
property StatusWindow: HWND;
```
Provided for case if You want to use API direct message sending to status bar.

```
property Color1: TColor;
```
Top 212 line color for GradientPanel 347.

```
property Color2: TColor;
```
Bottom line color for GradientPanel 347, or shadow color for LabelEffect 347. (If clNone, shadow color for LabelEffect is calculated as a mix between TextColor and clBlack).

```
property GradientStyle: TGradientStyle;
```
(TGradientStyle 208)

Styles other then gsVertical and gsHorizontal has effect only for gradient panel, created by NewGradientPanelEx.

```
property GradientLayout: TGradientLayout;
```
(TGradientLayout 208)

Has only effect for gradient panel, created by NewGradientPanelEx. Ignored for styles gsVertical and gsHorizontal.

© 2024 Carl Peeraer

property **ImageListSmall**: PImageList 178;
Image list with small icons used with List View control. If not set, last added (i.e. created with a control as an owner) image list with small icons is used.

property **ImageListNormal**: PImageList;
Image list with normal size icons used with List View control (or with icons for BitBtn, TreeView or TabControl). If not set, last added (i.e. created with a control as an owner) image list is used.

property ImageListState: PImageList 178;
Image list used as a state images list for ListView or TreeView control.

property **Pages**[Idx: Integer]: PControl;
Returns controls, which can be used as parent for controls, placed on different pages of a tab control. Use it like in follows example: Label1 := NewLabel(TabControl1.Pages[0], 'Label1'); To find number of pages available, check out Count 219 property of the tab control. Pages are enumerated from 0 to Count 219 - 1, as usual.

property **TC_Pages**[Idx: Integer]: PControl;
The same as above.

property **TC_Items**[Idx: Integer]: KOLString;
Text 219, displayed on tab control tabs.

property **TC_Images**[Idx: Integer]: Integer;
Image index for a tab in tab control.

property **TC_ItemRect**[Idx: Integer]: TRect;
Item rectangle for a tab in tab control.

property **LVStyle**: TListViewStyle 207;
ListView style of view. Can be changed at run time.

property **LVOptions**: TListViewOptions;
ListView options. Can be changed at run time.

property **LVTextColor**: TColor;
ListView text color. Use it instead of Font.Color.

property **LVTextBkColor**: TColor;

© 2024 Carl Peeraer

ListView background color for text.

`property LVBkColor: TColor;`
ListView background color. Use it instead of Color 221.

`property LVColCount: Integer;`
ListView (additional) column count. Value 0 means that there are no columns (single item text / icon is used). If You want to provide several columns, first call LVColAdd 257 to "insert" column 0, i.e. to provide header text for first column (with index 0). If there are no column, nothing will be shown in lvsDetail / lvsDetailNoHeader view style.

`property LVColWidth[ Item: Integer ]: Integer;`
Retrieves or changes column width. For lvsList view style, the same width is returned for all columns (ColIdx is ignored). It is possible to use special values to assign to a property:
LVSCW_AUTOSIZE - Automatically sizes the column
LVSCW_AUTOSIZE_USEHEADER - Automatically sizes the column to fit the header text
To set coumn width in lvsList view mode, column index must be -1 (and Width 212 to set must be in range 0..32767 always).

`property LVColText[ Idx: Integer ]: KOLString;`
Allows to get/change column header text at run time.

`property LVColAlign[ Idx: Integer ]: TTextAlign 206;`
Column text aligning.

`property LVColImage[ Idx: Integer ]: Integer;`
Only starting from comctrl32.dll of version 4.70 (IE4+). Allows to set an image for list view column itself from the ImageListSmall 229.

`property LVColOrder[ Idx: Integer ]: Integer;`
Only starting from comctrl32.dll of version 4.70 (IE4+). Allows to set visual order of the list view column from the ImageListSmall 229. This value does not affect the index, by which the column is still accessible in the column array.

`property LVCount: Integer;`
Returns item count for ListView control. It is possible to use Count 219 property instead when obtaining of item count is needed only. But this this property allows also to set actual count of list view items when a list view is virtual.

`property LVCurItem: Integer;`
Returns first selected item index in a list view. See also LVNextSelected 257, LVNextItem 257 and

© 2024 Carl Peeraer

LVFocusItem 231 functions.

```
property LVFocusItem: Integer;
```
Returns focused item index in a list view. See also LVCurItem 230.

```
property LVItemState[ Idx: Integer ]: TListViewItemState;
```
Access to list view item states set [lvisBlend, lvisHighlight, lvisFocus, lvisSelect]. When assign new value to the property, it is possible to use special index value -1 to change state for all items for a list view (but only when lvoMultiselect style is applied to the list view, otherwise index -1 is referring to the last item of the list view).

```
property LVItemIndent[ Idx: Integer ]: Integer;
```
Item indentation. Indentation is calculated as this value multiplied to image list ImgWidth value (Image list must be applied to list view). Note: indentation supported only if IE3.0 or higher installed.

```
property LVItemStateImgIdx[ Idx: Integer ]: Integer;
```
Access to state image of the item. Use index -1 to assign the same state image index to all items of the list view at once (fast). Option lvoCheckBoxes just means, that control itself creates special inner image list for two state images. Later it is possible to examine checked state for items or set checked state programmatically by changing LVItemStateImgIdx[ ] property. Value 1 corresponds to unchecked state, 2 to checked. Value 0 allows to remove checkbox at all. So, to check all added items by default (e.g.), do following:

```
ListView1.LVItemStateImgIdx[ -1 ] := 2;
```

Use 1-based index of the image in image list ImageListState 229. Value 0 reserved to use as "no state image". Values 1..15 can be used only - this is the Windows restriction on state images.

```
property LVItemOverlayImgIdx[ Idx: Integer ]: Integer;
```
Access to overlay image of the item. Use index -1 to assign the same overlay image to all items of the list view at once (fast).

```
property LVItemData[ Idx: Integer ]: DWORD;
```
Access to user defined data, assiciated with the item of the list view.

```
property LVSelCount: Integer;
```
Returns number of items selected in listview.

```
property LVItemImageIndex[ Idx: Integer ]: Integer;
```
Image index of items in listview. When an item is created (using LVItemAdd 258 or LVItemInsert 258), image index 0 is set by default (not -1 like in VCL!).

© 2024 Carl Peeraer

`property **LVItems**[ Idx, Col: Integer ]: KOLString;`
Access to List View item text.

`property **LVItemPos**[ Idx: Integer ]: TPoint;`
Position212 of List View item (can be changed in icon or small icon view).

`property **LVTopItem**: Integer;`
Returns index of topmost visible item of ListView in lvsList view style.

`property **LVPerPage**: Integer;`
Returns the number of fully-visible items if successful. If the current view is icon or small icon view, the return value is the total number of items in the list view control.

`property **LVItemHeight**: Integer;`

`property **TVSelected**: THandle;`
Returns or sets currently selected item handle in tree view.

`property **TVDropHilighted**: THandle;`
Returns or sets item, which is currently highlighted as a drop target.

`property **TVDropHilited**: THandle;`
The same as TVDropHilighted232.

`property **TVFirstVisible**: THandle;`
Returns or sets given item to top of tree view.

`property **TVIndent**: Integer;`
The amount, in pixels, that child items are indented relative to their parent items.

`property **TVVisibleCount**: Integer;`
Returns number of fully (not partially) visible items in tree view.

`property **TVRoot**: THandle;`
Returns handle of root item in tree view (or 0, if tree is empty).

`property **TVItemChild**[ Item: THandle ]: THandle;`
Returns first child item for given one.

`property **TVItemHasChildren**[ Item: THandle ]: Boolean;`

© 2024 Carl Peeraer

TRUE, if an Item has children. Set this value to true if you want to force [+] sign appearing left from the node, even if there are no subnodes added to the node yet.

`property` **TVItemChildCount**`[ Item: THandle ]: Integer;`
Returns number of node child items in tree view.

`property` **TVItemNext**`[ Item: THandle ]: THandle;`
Returns next sibling item handle for given one (or 0, if passed item is the last child for its parent node).

`property` **TVItemPrevious**`[ Item: THandle ]: THandle;`
Returns previous sibling item (or 0, if the is no such item).

`property` **TVItemNextVisible**`[ Item: THandle ]: THandle;`
Returns next visible item (passed item must be visible too, to determine, if it is really visible, use property TVItemRect 233 or TVItemVisible 233.

`property` **TVItemPreviousVisible**`[ Item: THandle ]: THandle;`
Returns previous visible item.

`property` **TVItemParent**`[ Item: THandle ]: THandle;`
Returns parent item for given one (or 0 for root item).

`property` **TVItemText**`[ Item: THandle ]: KOLString;`
Text 219 of tree view item.

`property` **TVItemRect**`[ Item: THandle; TextOnly: Boolean ]: TRect;`
Returns rectangle, occupied by an item in tree view.

`property` **TVItemVisible**`[ Item: THandle ]: Boolean;`
Returs True, if item is visible in tree view. It is also possible to assign True to this property to ensure that a tree view item is visible (if False is assigned, this does nothing).

`property` **TVRightClickSelect**`: Boolean;`
Set this property to True to allow change selection to an item, clicked with right mouse button.

`property` **TVEditing**`: Boolean;`
Returns True, if tree view control is editing its item label.

`property` **TVItemBold**`[ Item: THandle ]: Boolean;`

© 2024 Carl Peeraer

True, if item is bold.

`property TVItemCut[ Item: THandle ]: Boolean;`
True, if item is selected as part of "cut and paste" operation.

`property TVItemDropHighlighted[ Item: THandle ]: Boolean;`
True, if item is selected as drop target.

`property TVItemDropHilited[ Item: THandle ]: Boolean;`
The same as TVItemDropHighlighted 234.

`property TVItemExpanded[ Item: THandle ]: Boolean;`
True, if item's list of child items is currently expanded. To change expanded state, use method TVExpand 260.

`property TVItemExpandedOnce[ Item: THandle ]: Boolean;`
True, if item's list of child items has been expanded at least once.

`property TVItemSelected[ Item: THandle ]: Boolean;`
True, if item is selected.

`property TVItemImage[ Item: THandle ]: Integer;`
Image index for an item of tree view. To tell that there are no image set, use index -2 (value -1 is reserved for callback image).

`property TVItemSelImg[ Item: THandle ]: Integer;`
Image index for an item of tree view in selected state. Use value -2 to provide no image, -1 used for callback image.

`property TVItemOverlay[ Item: THandle ]: Integer;`
Overlay image index for an item in tree view. Values 1..15 can be used only - this is the Windows restriction on overlay images.

`property TVItemStateImg[ Item: THandle ]: Integer;`
State image index for an item in tree view. Use 1-based index of the image in image list ImageListState 229. Value 0 reserved to use as "no state image".

`property TVItemData[ Item: THandle ]: Pointer;`
Stores any program-defined pointer with the item.

© 2024 Carl Peeraer

property **TBCurItem**: Integer;
Same as CurItem.

property **TBButtonCount**: Integer;
Returns count of buttons on toolbar. The same as Count 219.

property **TBBtnImgWidth**: Integer;
Custom toolbar buttons width. Set it before assigning buttons bitmap. Changing this property after assigning the bitmap has no effect.

property **TBButtonEnabled**[BtnID: Integer]: Boolean;
Obvious.

property **TBButtonVisible**[BtnID: Integer]: Boolean;
Allows to hide/show some of toolbar buttons.

property **TBButtonChecked**[BtnID: Integer]: Boolean;
Allows to determine 'checked' state of a button (e.g., radio-button), and to check it programmatically.

property **TBButtonMarked**[BtnID: Integer]: Boolean;
Returns True if toolbar button is marked (highlighted). Allows to highlight buttons assigning True to this value.

property **TBButtonPressed**[BtnID: Integer]: Boolean;
Allows to detrmine if toolbar button (given by its command ID) pressed, and press/unpress it programmatically.

property **TBButtonText**[BtnID: Integer]: KOLString;
Obtains toolbar button text and allows to change it. Be sure that text is not empty for all buttons, if You want for it to be shown (if at least one button has empty text, no text labels will be shown at all). At least set it to ' ' for buttons, which You do not want to show labels, if You want from other ones to have it.

property **TBButtonImage**[BtnID: Integer]: Integer;
Allows to access/change button image. Do not read this property for separator buttons, returning value is not proper. If you do not know, is the button a separator, using function below.

property **TBButtonRect**[BtnID: Integer]: TRect;

© 2024 Carl Peeraer

Obtains rectangle occupied by toolbar button in toolbar window. (It is not possible to obtain rectangle for buttons, currently not visible). See also function ToolbarButtonRect.

property **TBButtonWidth**[BtnID: Integer]: Integer;
Allows to obtain / change toolbar button width.

property **TBButtonLParam**[const Idx: Integer]: DWORD;
Allows to access/change LParam. Dufa

property **TBButtonsMinWidth**: Integer;
Allows to set minimal width for all toolbar buttons.

property **TBButtonsMaxWidth**: Integer;
Allows to set maximal width for all toolbar buttons.

property **TBRows**: Integer;
Returns number of rows for toolbar and allows to try to set desired number of rows (but system can set another number of rows in some cases). This property has no effect if tboWrapable style not present in Options when toolbar is created.

property **DateTime**: TDateTime;
DateTime for DateTimePicker control only.

property **Date**: TDateTime;
Date only for DateTimePicker control only.

property **Time**: TDateTime;
Time only for DateTimePicker control only.

property **SystemTime**: TSystemTime;
Date236 and Time236 as TSystemTime. When assing, use year 0 to set "no value".

property **DateTimeRange**: TDateTimeRange;
DateTimePicker range. If first date in the agrument assigned is NAN, minimum system allowed value is used as the left bound, and if the second is NAN, maximum system allowed is used as the right one.

property **SBMin**: Longint;
Minimum scrolling area position.

property **SBMax**: Longint;
Maximum scrolling area position (size of the text or image to be scrolling). For case when SCROLL_OLD defined, this value should be set as scrolling object size without SBPageSize237.

© 2024 Carl Peeraer

`property` **`SBMinMax:`** `TPoint;`
The property to adjust SBMin 236 and SBMax 236 for a single call (set X to a minimum and Y to a maximum value).

`property` **`SBPosition:`** `Integer;`
Current scroll position. When set, should be between SBMin 236 and SBMax 236 - max(0, SBPageSize 237-1)

`property` **`SBPageSize:`** `Integer;`

`property` **`Checked:`** `Boolean;`
For checkbox and radiobox - if it is checked. Do not assign value for radiobox - use SetRadioChecked 265 instead.

`property` **`Check3:`** `TTriStateCheck;`
State of checkbox with BS_AUTO3STATE style.

`property` **`CustomData:`** `Pointer;`
Can be used to exend the object when new type of control added. Memory, pointed by this pointer, released automatically in the destructor.

`property` **`CustomObj:`** `PObj 92;`
Can be used to exend the object when new type of control added. Object, pointed by this pointer, released automatically in the destructor.

`property` **`MDIClient:`** `PControl;`
For MDI forms only: returns MDI client window control, containng all MDI children. Use this window to send specific messages to rule MDI children.

`property` **`LBTopIndex:`** `Integer;`
Index of the first visible item in a list box

`property` **`MaxTextSize:`** `DWORD;`
This property valid also for simple edit control, not only for RichEdit. But for usual edit control, maximum text size available is 32K. For RichEdit, limit is 4Gb. By default, RichEdit is limited to 32767 bytes (to set maximum size available to 2Gb, assign MaxInt value to a property). Also, to get current text size of RichEdit, use property TextSize 237 or RE_TextSize 238[ ].

`property` **`TextSize:`** `Integer;`
Common for edit and rich edit controls property, which returns size of text in edit control. Also, for any other control (or form, or applet window) returns size (in characters) of Caption 219 or

© 2024 Carl Peeraer

Text 219 (what is, the same property actually).

```
property RE_TextSize[ Units: TRichTextSize ]: Integer;
```
For RichEdit control, it returns text size, measured in desired units (rtsChars - characters, including OLE objects, counted as a single character; rtsBytes - presize length of text image (if it would be stored in file or stream). Please note, that for RichEdit1.0, only size in characters can be obtained.

```
property RE_CharFmtArea: TRichFmtArea;
```
By default, this property is raSelection. Changing it, You determine in for which area characters format is applyed, when changing character formatting properties below (not paragraph formatting).

```
property RE_CharFormat: TCharFormat;
```
In differ to follow properties, which allow to control certain formatting attributes, this property provides low level access for formatting current character area (see RE_CharFmtArea 238). It returns TCharFormat structure, filled in with formatting attributes, and by assigning another value to this property You can change desired attributes as You wish. Even if RichEdit1.0 is used, TCharFormat2 is returned (but extended fields are ignored for RichEdit1.0).

```
property RE_Font: PGraphicTool;
```
Font 221 of the first character in current selection (when retrieve). When set (or subproperties of RE_Font are set), all font attributes are applied to entire area 238. To apply only needed attributes, use another properties: RE_FmtBold 238, RE_FmtItalic 238, RE_FmtStrikeout 238, RE_FmtUnderline 239, RE_FmtName, etc.
Note, that font size is measured in twips, which is about 1/10 of pixel.

```
property RE_FmtBold: Boolean;
```
Formatting flag. When retrieve, returns True, if fsBold style RE_Font.FontStyle is valid for a first character in the selection. When set, changes fsBold style (True - set, False - reset) for all characters in area 238.

```
property RE_FmtBoldValid: Boolean;
```

```
property RE_FmtItalic: Boolean;
```
Formatting flag. Like RE_FmtBold 238, when retrieving, shows, is fsItalic style valid for the first character of the selection, and when set, changes only fsItalic style for an area 238.

```
property RE_FmtItalicValid: Boolean;
```

```
property RE_FmtStrikeout: Boolean;
```


© 2024 Carl Peeraer

Formatting flag. Like RE_FmtBold 238, when retrieving, shows, is fsStrikeout style valid for the first selected character, and when set, changes only fsStrikeout style for an area 238 .

`property` **RE_FmtStrikeoutValid:** `Boolean;`

`property` **RE_FmtUnderline:** `Boolean;`
Formatting flag. Like RE_FmtBold 238, when retrieving, shows, is fsUnderline style valid for the first selected character, and when set, changes fsUnderline style for an area 238 .

`property` **RE_FmtUnderlineValid:** `Boolean;`

`property` **RE_FmtUnderlineStyle:** `TRichUnderline;`
Extended underline style. To check, if this property is valid for entire selection, examine RE_FmtUnderlineValid 239 value.

`property` **RE_FmtProtected:** `Boolean;`
Formatting flag. When retrieving, shows, is the first character of the selection is protected from changing it by user (True) or not (False). To get know, if retrived value is valid for entire selection, check the property RE_FmtProtectedValid 239. When set, makes all characters in area 238 protected (True) or not (False).

`property` **RE_FmtProtectedValid:** `Boolean;`
True, if property RE_FmtProtected 239 is valid for entire selection, when retrieving it.

`property` **RE_FmtHidden:** `Boolean;`
For RichEdit3.0, makes text hidden (not displayed).

`property` **RE_FmtHiddenValid:** `Boolean;`
Returns True, if RE_FmtHidden 239 style is valid for entire selection.

`property` **RE_FmtLink:** `Boolean;`
Returns True, if the first selected character is a part of link (URL).

`property` **RE_FmtLinkValid:** `Boolean;`

`property` **RE_FmtFontSize:** `Integer index( 12 shl 16 ) or CFM_SIZE;`
Formatting value: font size, in twips (1/1440 of an inch, or 1/20 of a printer's point, or about 1/10 of pixel). When retrieving, returns RE_Font.FontHeight. When set, changes font size for entire area 238 (but does not change other font attributes).

© 2024 Carl Peeraer

`property` **`RE_FmtFontSizeValid:`** `Boolean;`
Returns True, if property RE_FmtFontSize 239 is valid for entire selection, when retrieving it.

`property` **`RE_FmtAutoBackColor:`** `Boolean;`
True, when automatic back color is used.

`property` **`RE_FmtAutoBackColorValid:`** `Boolean;`

`property` **`RE_FmtFontColor:`** `Integer index( 20 shl 16 ) or CFM_COLOR;`
Formatting value (font color). When retrieving, returns RE_Font.Color. When set, changes font color for entire area 238 (but does not change other font attributes).

`property` **`RE_FmtFontColorValid:`** `Boolean;`
Returns True, if property RE_FmtFontColor 240 valid for entire selection, when retrieving it.

`property` **`RE_FmtAutoColor:`** `Boolean;`
True, when automatic text color is used (in such case, RE_FmtFontColor 240 assignment is ignored for current area).

`property` **`RE_FmtAutoColorValid:`** `Boolean;`

`property` **`RE_FmtBackColor:`** `Integer index(( 64 + 32 ) shl 16 ) or CFM_BACKCOLOR;`
Formatting value (back color). Only available for Rich Edit 2.0 and higher. When set, changes background color for entire area 238 (but does not change other font attributes).

`property` **`RE_FmtBackColorValid:`** `Boolean;`

`property` **`RE_FmtFontOffset:`** `Integer index( 16 shl 16 ) or CFM_OFFSET;`
Formatting value (font vertical offset from baseline, positive values correspond to subscript). When retrieving, returns offset for first character in the selection. When set, changes font offset for entire area 238 . To get know, is retrieved value valid for entire selction, check RE_FmtFontOffsetValid 240 property.

`property` **`RE_FmtFontOffsetValid:`** `Boolean;`
Returns True, if property RE_FmtFontOffset 240 is valid for entire selection, when retrieving it.

`property` **`RE_FmtFontCharset:`** `Integer index( 25 shl 16 ) or CFM_CHARSET;`
Returns charset for first character in current selection, when retrieved (and to get know, if this value is valid for entire selection, check property RE_FmtFontCharsetValid 241). When set, changes charset for all characters in area 238 , but does not alter other formatting attributes.

© 2024 Carl Peeraer

`property` **`RE_FmtFontCharsetValid:`** `Boolean;`
Returns True, only if rerieved property RE_FmtFontCharset 240 is valid for entire selection.

`property` **`RE_FmtFontName:`** `KOLString;`
Returns font face name for first character in the selection, when retrieved, and sets font name for entire area 238, wnen assigned to (without changing of other formatting attributes). To get know, if retrived font name valid for entire selection, examine property RE_FmtFontNameValid 241.

`property` **`RE_FmtFontNameValid:`** `Boolean;`
Returns True, only if the font name is the same for entire selection, thus is, if rerieved property value RE_FmtFontName 241 is valid for entire selection.

`property` **`RE_ParaFmt:`** `TParaFormat;`
Allows to retrieve or set paragraph formatting attributes for currently selected paragraph(s) in RichEdit control. See also following properties, which allow to do the same for certain paragraph format attributes separately.

`property` **`RE_TextAlign:`** `TRichTextAlign` 206`;`
Returns text alignment for current selection and allows to change it (without changing other formatting attributes).

`property` **`RE_TextAlignValid:`** `Boolean;`
Returns True, if property RE_TextAlign 241 is valid for entire selection. If False, it is concerning only start of selection.

`property` **`RE_Numbering:`** `Boolean;`
Returns True, if selected text is numbered (or has style of list with bullets). To get / change numbering style, see properties RE_NumStyle 241 and RE_NumBrackets 241.

`property` **`RE_NumStyle:`** `TRichNumbering` 209`;`
Advanced numbering style, such as rnArabic etc. If You use it, do not change RE_Numbering 241 property simultaneously - this can cause changing style to rnBullets only.

`property` **`RE_NumStart:`** `Integer;`
Starting number for advanced numbering style. If this property is not set, numbering is starting by default from 0. For rnLRoman and rnURoman this cause, that first item has no number to be shown (ancient Roman people did not invent '0').

`property` **`RE_NumBrackets:`** `TRichNumBrackets` 209`;`

 © 2024 Carl Peeraer

Brackets style for advanced numbering. rnbPlain is default brackets style, and every time, when RE_NumStyle 241 is changed, RE_NumBrackets is reset to rnbPlain.

`property` **`RE_NumTab`**`: Integer;`
Tab between start of number and start of paragraph text. If too small too view number, number is not displayed. (Default value seems to be sufficient though).

`property` **`RE_NumberingValid`**`: Boolean;`
Returns True, if RE_Numbering 241, RE_NumStyle 241, RE_NumBrackets 241, RE_NumTab 242, RE_NumStart 241 properties are valid for entire selection.

`property` **`RE_Level`**`: Integer;`
Outline level (for numbering paragraphs?). Read only.

`property` **`RE_SpaceBefore`**`: Integer;`
Spacing before paragraph.

`property` **`RE_SpaceBeforeValid`**`: Boolean;`
True, if RE_SpaceBefore 242 value is valid for all selected paragraph (if False, this value is valid only for first paragraph.

`property` **`RE_SpaceAfter`**`: Integer;`
Spacing after paragraph.

`property` **`RE_SpaceAfterValid`**`: Boolean;`
True, only if RE_SpaceAfter 242 value is valid for all selected paragraphs.

`property` **`RE_LineSpacing`**`: Integer;`
Linespacing in paragraph (this value is based on RE_SpacingRule 242 property).

`property` **`RE_SpacingRule`**`: Integer;`
Linespacing rule. Do not know what is it.

`property` **`RE_LineSpacingValid`**`: Boolean;`
True, only if RE_LineSpacing 242 and RE_SpacingRule 242 values are valid for entire selection.

`property` **`RE_Indent`**`: Integer index( 20 shl 16 ) or PFM_OFFSET;`
Returns left indentation for paragraph in current selection and allows to change it (without changing other formatting attributes).

© 2024 Carl Peeraer

`property` **RE_IndentValid:** `Boolean;`
Returns True, if RE_Indent 242 property is valid for entire selection.

`property` **RE_StartIndent:** `Integer index( 12 shl 16 ) or PFM_STARTINDENT;`
Returns left indentation for first line in paragraph for current selection, and allows to change it (without changing other formatting attributes).

`property` **RE_StartIndentValid:** `Boolean;`
Returns True, if property RE_StartIndent 243 is valid for entire selection.

`property` **RE_RightIndent:** `Integer index( 16 shl 16 ) or PFM_RIGHTINDENT;`
Returns right indent for paragraph in current selection, and allow to change it (without changing other formatting attributes).

`property` **RE_RightIndentValid:** `Boolean;`
Returns True, if property RE_RightIndent 243 is valid for entire selection only.

`property` **RE_TabCount:** `Integer;`
Number of tab stops in current selection. This value can not be set greater then MAX_TAB_COUNT (32).

`property` **RE_Tabs**`[ Idx: Integer ]: Integer;`
Tab stops for RichEdit control.

`property` **RE_TabsValid:** `Boolean;`
Returns True, if properties RE_Tabs 243[ ] and RE_TabCount 243 are valid for entire selection.

`property` **RE_AutoKeyboard:** `Boolean;`
True if autokeyboard on (lovely "feature" of automatic switching keyboard language when caret is over another language text). For older RichEdit, is 'on' always, for newest - 'off' by default.

`property` **RE_AutoFont:** `Boolean;`
True if autofont on (automatic switching font when keyboard layout is changes). By default, is 'on' always. It is suggested to turn this option off for Unicode control.

`property` **RE_AutoFontSizeAdjust:** `Boolean;`
See IMF_AUTOFONTSIZEADJUST option in SDK: Font 221-bound font sizes are scaled from insertion point size according to script. For example, Asian fonts are slightly larger than Western ones. This option is turned on by default.

© 2024 Carl Peeraer

```
property RE_DualFont: Boolean;
```
See IMF_DUALFONT option in SDK: Sets the control to dual-font mode. Used for Asian language support. The control uses an English font for ASCII text and a Asian font for Asian text.

```
property RE_UIFonts: Boolean;
```
See IMF_UIFONTS option in SDK: Use user-interface default fonts. This option is turned off by default.

```
property RE_IMECancelComplete: Boolean;
```
See IMF_IMECANCELCOMPLETE option in SDK: This flag determines how the control uses the composition string of an IME if the user cancels it. If this flag is set, the control discards the composition string. If this flag is not set, the control uses the composition string as the result string.

```
property RE_IMEAlwaysSendNotify: Boolean;
```
See IMF_IMEALWAYSSENDNOTIFY option in SDK: Controls how Rich Edit notifies the client during IME composition:
0: No EN_CHANGED or EN_SELCHANGE notifications during undetermined state. Send notification when final string comes in. (default)
1: Send EN_CHANGED and EN_SELCHANGE events during undetermined state.

```
property RE_OverwriteMode: Boolean;
```
This property allows to control insert/overwrite mode. First, to examine, if insert or overwrite mode is current (but it is necessary either to access this property, at least once, immediately after creating RichEdit control, or to assign event OnRE_InsOvrMode_Change 277 to your handler). Second, to set desired mode programmatically - by assigning value to this property (You also have to initialize monitoring procedure by either reading RE_OverwriteMode property or assigning handler to event OnRE_InsOvrMode_Change 277 immediately following RichEdit control creation).

```
property RE_DisableOverwriteChange: Boolean;
```
It is possible to disable switching between "insert" and "overwrite" mode by user (therefore, event OnRE_InsOvrMode_Change 277 continue works, but it just called when key INSERT is pressed, though RE_OverwriteMode 244 property is not actually changed if switching is disabled).

```
property RE_Text[ Format: TRETextFormat; SelectionOnly: Boolean ]: KOLString;
```
This property allows to get / replace content of RichEdit control (entire text or selection only). Using different formats, it is possible to exclude or replace undesired formatting information (see TRETextFormat specification). To get or replace entire text in reText mode (plain text only), it is possible to use habitual for edit controls Text 219 property.
Note: it is possible to append text to the end of RichEdit control using method Add 255, but only if property RE_Text is accessed at least once:

© 2024 Carl Peeraer

```
RichEdit1.RE_Text[ reText, True ];
```

`property` **RE_Error**`: Integer;`
Contains error code, if access to RE_Text 244 failed.

`property` **RE_AutoURLDetect**`: Boolean;`
If set to True, automatically detects URLs (and highlights it with blue color, applying fsItalic and fsUnderline font styles (while typing and loading). Default value is False. Note: if event OnRE_URLClick 277 or event OnRE_OverURL 277 are set, property RE_AutoURLDetect is set to True automatically.

`property` **RE_URL**`: PKOLChar;`
Detected URL (valid in OnRE_OverURL 277 and OnRE_URLClick 277 event handlers).

`property` **RE_Transparent**`: Boolean;`
Use this property to make richedit control transparent, instead of Ed_Transparent 224 or Transparent 224. But do not place such transparent richedit control directly on form - it can be draw incorrectly when form is activated and rich editr control is not current active control. Use at least panel as a parent instead.

`property` **RE_Zoom**`: TSmallPoint;`
To set zooming for rich edit control (3.0 and above), pass X as numerator and Y as denominator. Resulting X/Y must be between 1/64 and 64.

`property` **PropInt**`[ PropName: PKOLChar ]: Integer;`
For any windowed control: use it to store desired property in window properties.

`property` **Align**`: TControlAlign;`
Align style of a control. If this property is not used in your application, there are no additional code added. Aligning of controls is made in KOL like in VCL. To align controls when initially create ones, use "transparent" function SetAlign 265 ("transparent" means that it returns @Self as a result).
Note, that it is better not to align combobox caClient, caLeft or caRight (better way is to place a panel with Border 222 = 0 and EdgeStyle = esNone, align it as desired and to place a combobox on it aligning caTop or caBottom). Otherwise, big problems could be under Win9x/Me, and some delay could occur under any other systems.
Do not attempt to align some kinds of controls (like combobox) caLeft or caRight, this can cause infinite recursion.

TControl methods

© 2024 Carl Peeraer

`procedure` **`Run`**`( var AppletCtl: PControl` 203 `);`
Call this procedure to process messages loop of your program. Pass here pointer to applet button object (if You have created it - see NewApplet 369) or your main form object of type PControl 203 (created using NewForm 367).

`function` **`FormGetIntParam`**`: Integer;`
Extracts the next integer parameter up to ',' or up to ';'

`function` **`FormGetColorParam`**`: Integer;`
Extracts the next integer parameter up to ',' or up to ';'

`procedure` **`FormGetStrParam`**`;`
Extracts the next string parameter up to ',' or up to ';' -> FormString 278

`procedure` **`FormCreateParameters`**`( alphabet: PFormInitFuncArray; params: PAnsiChar );`
Sets the initial alphabet and parameters with commands

`procedure` **`FormExecuteCommands`**`( AForm: PControl; ControlPtrOffsets: PSmallIntArray );`
Executes commands (with parameters) to the end or to ';'

`procedure` **`InitParented`**`( AParent: PControl ); virtual;`
Initialization of visual object.

`procedure` **`InitOrthaned`**`( AParentWnd: HWnd ); virtual;`
Initialization of visual object.

`PROCEDURE` **`InitParented`**`( AParent: PControl; widget: PGtkWidget; need_eventbox: Boolean ); VIRTUAL;`
Initialization of visual object.

`procedure` **`DestroyChildren`**`;`
Destroys children. Is called in destructor, and can be called in descending classes as earlier as needed to prevent problems of too late destroying of visuals.
Note: since v 2.40, used only for case when a symbol NOT_USE_AUTOFREE4CONTROLS is defined, otherwise all children are destroyed using common mechanism of Add2AutoFree.

`function` **`GetParentWnd`**`( NeedHandle: Boolean ): HWnd;`
Returns handle of parent window.

`function` **`GetParentWindow`**`: HWnd;`

`procedure` **`SetEnabled`**`( Value: Boolean );`
Changes Enabled 212 property value. Overriden here to change enabling status of a window.

`function` **`GetEnabled`**`: Boolean;`
Returns True, if Enabled 212. Overriden here to obtain real window state.

© 2024 Carl Peeraer

```
procedure SetVisible( Value: Boolean );
```
Sets Visible 212 property value. Overriden here to change visibility of correspondent window.

```
procedure Set_Visible( Value: Boolean );
```

```
function GetVisible: Boolean;
```
Returns True, if correspondent window is Visible 212. Overriden to get visibility of real window, not just value stored in object.

```
function Get_Visible: Boolean;
```
Returns True, if correspondent window is Visible 212, for forms and applet, or if fVisible flag is set, for controls.

```
procedure SetCtlColor( Value: TColor );
```
Sets TControl's Color 221 property value.

```
procedure SetBoundsRect( const Value: TRect );
```
Sets BoudsRect property value.

```
function GetBoundsRect: TRect;
```
Returns bounding rectangle.

```
function GetIcon: HIcon;
```
Returns Icon 218 property. By default, if it is not set, returns Icon 218 property of an Applet.

```
procedure CreateSubclass( var Params: TCreateParams; ControlClassName: PKOLChar );
```

Can be used in descending classes to subclass window with given standard Windows ControlClassName - must be called after creating Params but before CreateWindow 250. Usually it is called in overriden method CreateParams after calling of the inherited one.

```
procedure SetOnChar( const Value: TOnChar );
```

```
procedure SetOnDeadChar( const Value: TOnChar );
```

```
procedure SetOnKeyDown( const Value: TOnKey );
```

```
procedure SetOnKeyUp( const Value: TOnKey );
```

```
procedure SetHelpContext( Value: Integer );
```

```
procedure SetOnTVDelete( const Value: TOnTVDelete );
```

```
function DefaultBtnProc( var Msg: TMsg; var Rslt: Integer ): Boolean;
```

```
constructor CreateParented( AParent: PControl );
```
Creates new instance of TControl object, calling InitParented 246

© 2024 Carl Peeraer

`constructor` **`CreateOrthaned`**`( AParentWnd: HWnd );`
Creates new instance of TControl object, calling InitOrthaned 246

`CONSTRUCTOR` **`CreateParented`**`( AParent: PControl; widget: PGtkWidget; need_eventbox: Boolean );`
Creates new instance of TControl object, calling InitParented 246

`destructor` **`Destroy`**`; virtual;`
Destroyes object. First of all, destructors for all children are called.

`function` **`GetWindowHandle`**`: HWnd;`
Returns window handle. If window is not yet created, method CreateWindow 250 is called.

`procedure` **`CreateChildWindows`**`;`
Enumerates all children recursively and calls CreateWindow 250 for all of these.

`function` **`ChildIndex`**`( Child: PControl ): Integer;`
Returns index of given child.

`procedure` **`MoveChild`**`( Child: PControl; NewIdx: Integer );`
Moves given Child into new position.

`procedure` **`EnableChildren`**`( Enable, Recursive: Boolean );`
Enables (Enable = TRUE) or disables (Enable = FALSE) all the children of the control. If Recursive = TRUE then all the children of all the children are enabled or disabled recursively.

`function` **`ClientRect`**`: TRect;`
Client rectangle of TControl. Contrary to VCL, for some classes (e.g. for graphic controls) can be relative not to itself, but to top left corner of the parent's ClientRect rectangle.

`function` **`ControlRect`**`: TRect;`
Absolute bounding rectangle relatively to nearest Windowed 213 parent client rectangle (at least to a form, but usually to a Parent 212). Useful while drawing on device context, provided by such Windowed 213 parent. For form itself is the same as BoundsRect 212.

`function` **`ControlAtPos`**`( X, Y: Integer; IgnoreDisabled: Boolean ): PControl;`
Searches control at the given position (relatively to top left corner of the ClientRect 248).

`procedure` **`Invalidate`**`;`
Invalidates rectangle, occupied by the visual (but only if Showing = True).

© 2024 Carl Peeraer

`procedure **InvalidateEx**;`
Invalidates the window and all its children.

`procedure **InvalidateNC**( Recursive: Boolean );`
Invalidates the window and all its children including non-client area.

`procedure **Update**;`
Updates control's window and calls Update for all child controls.

`procedure **BeginUpdate**;`
Call this method to stop visual updates of the control until correspondent EndUpdate 249 called (pairs BeginUpdate - EndUpdate 249 can be nested).

`procedure **EndUpdate**;`
See BeginUpdate 249.

`function **HandleAllocated**: Boolean;`
Returns True, if window handle is allocated. Has no sense for non-Windowed 213 objects (but now, the KOL has no non-Windowed 213 controls).

`procedure **PaintBackground**( DC: HDC; Rect: PRect );`
Is called to paint background in given rectangle. This method is filling clipped area of the Rect rectangle with Color 221, but only if global event Global_OnPaintBkgnd is not assigned. If assigned, this one is called instead here.
This method made public, so it can be called directly to fill some device context's rectangle. But remember, that independantly of Rect, top left corner of background piece will be located so, if drawing is occure into ControlRect 248 rectangle.

`function **ParentForm**: PControl;`
Returns parent form for a control (of @Self for form itself.

`function **FormParentForm**: PControl;`
Returns parent form for a control (of @Self for form itself. For a frame, returns frame panel instead.

`function **MarkPanelAsForm**: PControl;`
Special function for MCK to mark panel as frame parent control.

`function **Client2Screen**( const P: TPoint ): TPoint;`
Converts the client coordinates of a specified point to screen coordinates.

`function **Screen2Client**( const P: TPoint ): TPoint;`

© 2024 Carl Peeraer

Converts screen coordinates of a specified point to client coordinates.

```
function CreateWindow: Boolean; virtual;
```
Creates correspondent window object. Returns True if success (if window is already created, False is returned). If applied to a form, all child controls also allocates handles that time.
Call this method to ensure, that a hanle is allocated for a form, an application button or a control. (It is not necessary to do so in the most cases, even if You plan to work with control's handle directly. But immediately after creating the object, if You want to pass its handle to API function, this can be helpful).

```
procedure Close;
```
Closes window. If a window is the main form, this closes application, terminating it. Also it is possible to call Close method for Applet window to stop application.

```
procedure CursorLoad( Inst: Integer; ResName: PKOLChar );
```
Loads Cursor 218 from the resource. See also comments for Icon 218 property.

```
procedure IconLoad( Inst: Integer; ResName: PKOLChar );
```
See Icon 218 property.

```
procedure IconLoadCursor( Inst: Integer; ResName: PKOLChar );
```
Loads Icon 218 from the cursor resource. See also Icon 218 property.

```
function AssignHelpContext( Context: Integer ): PControl;
```
Assigns HelpContext 218 and returns @ Self (can be used in initialization of a control in a chain of "transparent" calls).

```
procedure CallHelp( Context: Integer; CtxCtl: PControl );
```
Method of a form or Applet. Call it to show help with the given context ID. If the Context = 0, help contents is displayed. By default, WinHelp is used. To allow using HtmlHelp, call AssignHtmlHelp global function. When WinHelp used, HelpPath 218 variable can be assigned directly. If HelpPath 218 variable is not assigned, application name (and path) is used, with extension replaced to '.hlp'.

```
procedure SelectAll;
```
Makes all the text in editbox or RichEdit, or all items in listbox selected.

```
procedure ReplaceSelection( const Value: KOLString; aCanUndo: Boolean );
```
Replaces selection (in edit, RichEdit). Unlike assigning new value to Selection 219 property, it is possible to specify, if operation can be undone.

© 2024 Carl Peeraer

Use this method or assigning value to a Selection 219 property to format text initially in the rich edit. E.g.:

```
    RichEdit1.RE_FmtBold := TRUE;
    RichEdit1.Selection := 'bolded text'#13#10;
    RichEdit1.RE_FmtBold := FALSE;
    RichEdit1.RE_FmtItalic := TRUE;
    RichEdit1.Selection := 'italized text';
```

...

procedure **DeleteLines**(FromLine, ToLine: Integer);
Deletes lines from FromLine to ToLine (inclusively, i.e. 0 to 0 deletes one line with index 0). Current selection is restored as possible.

function **Item2Pos**(ItemIdx: Integer): DWORD;
Only for edit controls: converts line index to character position.

function **Pos2Item**(Pos: Integer): DWORD;
Only for edit controls: converts character position to line index.

function **SavePosition**: TEditPositions;
Only for edit controls: saves current editor selection and scroll positions. To restore position, use RestorePosition 251 with a structure, containing saved position as a parameter.

procedure **RestorePosition**(const p: TEditPositions);
Call RestorePosition with a structure, containing saved position as a parameter (this structure filled in in SavePosition 251 method). If you set RestoreScroll to FALSE, only selection is restored, without scroll position.

procedure **UpdatePosition**(var p: TEditPositions; FromPos, CountInsertDelChars, CountInsertDelLines: Integer);
If you called SavePosition 251 and then make some changes in the edit control, calling RestorePosition 251 will fail if chages are affecting selection size. The problem can be solved updating saved position info using this method. Pass a count of inserted characters and lines as a positive number and a count of deleted characters as a negative number here.
CountInsertDelLines is optional paramters: if you do not specify it, only selection is fixed.

function **EditTabChar**: PControl;
Call this method (once) to provide insertion of tab character (code #9) when tab key is pressed on keyboard.

function **IndexOf**(const S: KOLString): Integer;
Works for the most of control types, though some of those have its own methods to search given item. If a control is not list box or combobox, item is finding by enumerating all the Items 219 one by one. See also SearchFor 252 method.

`function SearchFor( const S: KOLString; StartAfter: Integer; Partial: Boolean ): Integer;`

Works for the most of control types, though some of those have its own methods to search given item. If a control is not list box or combobox, item is finding by enumerating all the Items[219] one by one. See also IndexOf[251] method.

`procedure AddDirList( const Filemask: KOLString; Attrs: DWORD );`

Can be used only with listbox and combobox - to add directory list items, filtered by given Filemask (can contain wildcards) and Attrs. Following flags can be combined in Attrs: If the listbox is sorted, directory items will be sorted (alpabetically).

`function SetButtonIcon( aIcon: HIcon ): PControl;`

Sets up button icon image and changes its styles. Returns button itself.

`function SetButtonBitmap( aBmp: HBitmap ): PControl;`

Sets up button icon image and changes its styles. Returns button itself.

`function AllBtnReturnClick: PControl;`

Call this method for a form or control to provide clicking a focused button when ENTER pressed. By default, a button can be clicked only by SPACE key from the keyboard, or by mouse.

`procedure Show;`

Makes control visible and activates it.

`function ShowModal: Integer;`

Can be used only with a forms to show it modal. See also global function ShowMsgModal.
To use a form as a modal, it is possible to make it either auto-created or dynamically created.
For a first case, You (may be prefer to hide a form after showing it as a modal:

```
procedure TForm1.Button1Click( Sender: PObj );
begin
  Form2.Form.ShowModal;
  Form2.Form.Hide;
end;
```

Another way is to create modal form just before showing it (this economies system resources):

```
procedure TForm1.Button1Click( Sender: PObj );
begin
  NewForm2( Form2, Applet );
  Form2.Form.ShowModal;
  Form2.Form.Free;
  // Never call Form2.Free or Form2.Form.Close
  // but always Form2.Form.Free; (!)
end;
```


© 2024 Carl Peeraer

In samples above, You certainly can place any wished code before and after calling ShowModal method.
Do not forget that if You have more than a single form in your project, separate Applet object should be used.
See also ShowModalEx253.

`function` **`ShowModalParented`**`( const AParent: PControl ): Integer;`
by Alexander Pravdin. The same as ShowModal252, but with a certain form as a parent.

`function` **`ShowModalEx`**`: Integer;`
The same as ShowModal252, but all the windows of current thread are disabled while showing form modal. This is useful if KOL form from a DLL is used modally in non-KOL application.

`procedure` **`Hide`**`;`
Makes control hidden.

`function` **`CallDefWndProc`**`( var Msg: TMsg ): Integer;`
Function to be called in WndProc254 method to redirect message handling to default window procedure.

`function` **`DoSetFocus`**`: Boolean;`
Sets focus for Enabled212 window. Returns True, if success.

`procedure` **`MinimizeNormalAnimated`**`;`
Apply this method to a main form (not to another form or Applet, even when separate Applet control is not used and main form matches it!). This provides normal animated visual minimization for the application. It therefore has no effect, if animation during minimize/resore is turned off by user.
Applying this method also provides for the main form (only for it) correct restoring the form maximized if it was maximized while minimizing the application. See also RestoreNormalMaximized253 method.

`procedure` **`RestoreNormalMaximized`**`;`
Apply to any form for which it is important to restore it maximized when the application was minimizing while such form was maximized. If the method MinimizeNormalAnimated253 was called for the main form, then the correct behaviour is already provided for the main form, so in such case it is no more necessary to call also this method, but calling it therefore is not an error.

`function` **`IsMainWindow`**`: Boolean;`
Returns True, if a window is the main in application (created first after the Applet, or matches the Applet).

© 2024 Carl Peeraer

`function` **ProcessMessage**`: Boolean;`
Processes one message. See also ProcessMessages 254.

`procedure` **ProcessMessages**`;`
Processes pending messages during long cycle of calculation, allowing to window to be repainted if needed and to respond to other messages. But if there are no such messages, your application can be stopped until such one appear in messages queue. To prevent such situation, use method ProcessPendingMessages 254 instead.

`procedure` **ProcessMessagesEx**`;`
Version of ProcessMessages 254, which works always correctly, even if the application is minimized or background.

`procedure` **ProcessPendingMessages**`;`
Similar to ProcessMessages 254, but without waiting of message in messages queue. I.e., if there are no pending messages, this method immediately returns control to your code. This method is better to call during long cycle of calculation (then ProcessMessages 254).

`procedure` **ProcessPaintMessages**`;`

`function` **WndProc**`( var Msg: TMsg ): Integer; virtual;`
Responds to all Windows messages, posted (sended) to the window, before all other proceeding. You can override it in derived controls, but in KOL there are several other ways to control message flow of existing controls without deriving another costom controls for only such purposes. See OnMessage 270, AttachProc 265.

`function` **SetBorder**`( Value: Integer ): PControl;`
Assigns new Border 222 value, and returns @ Self.

`function` **BringToFront**`: PControl;`
Changes z-order of the control, bringing it to the topmost level.

`function` **SendToBack**`: PControl;`
Changes z-order of the control, sending it to the back of siblings.

`function` **DblBufTopParent**`: PControl;`
Returns the topmost DoubleBuffered 224 Parent 212 control.

`function` **MouseTransparent**`: PControl;`

© 2024 Carl Peeraer

Call this method to set up mouse transparent control (which always returns HTTRANSPARENT in responce to WM_NCHITTEST). This function returns a pointer to a control itself.

```
procedure GotoControl( Key: DWORD );
```
Emulates tabulation key press w/o sending message to current control. Can be applied to a form or to any its control. If VK_TAB is used, state of shift kay is checked in: if it is pressed, tabulate is in backward direction.

```
procedure DragStart;
```
Call this method for a form or control to drag it with left mouse button, when mouse left button is already down. Dragging 226 is stopped when left mouse button is released. See also DragStartEx 255, DragStopEx 255.

```
procedure DragStartEx;
```
Call this method to start dragging the form by mouse. To stop dragging, call DragStopEx 255 method. (Tip: to detect mouse up event, use OnMouseUp 273 event of the dragging control). This method can be used to move any control with the mouse, not only entire form. State of mouse button is not significant. Determine dragging state of the control checking its Dragging 226 property.

```
procedure DragStopEx;
```
Call this method to stop dragging the form (started by DragStopEx).

```
procedure DragItem( OnDrag: TOnDrag );
```
Starts dragging something with mouse. During the process, callback function OnDrag is called, which allows to control drop target, change cursor shape, etc.

```
function LikeSpeedButton: PControl;
```
Transparent 224 method (returns control itself). Makes button not focusable.

```
function Add( const S: KOLString ): Integer;
```
Only for listbox and combobox.

```
function Insert( Idx: Integer; const S: KOLString ): Integer;
```
Only for listbox and combobox.

```
procedure Delete( Idx: Integer );
```
Deletes given (by index) pointer item from the list, shifting all follow item indexes up by one.

```
procedure Clear;
```
Clears object content. Has different sense for different controls. E.g., for label, editbox, button and other simple controls it assigns empty string to Caption 219 property. For listbox, combobox,

© 2024 Carl Peeraer

listview it deletes all items. For toolbar, it deletes all buttons. Et so on.

```
procedure RemoveStatus;
```
Call it to remove status bar from a form (created in result of assigning value(s) to StatusText 227 [], SimpleStatusText 227 properties). When status bar is removed, form is resized to preset client height.

```
function StatusPanelCount: Integer;
```
Returns number of status panels defined in status bar.

```
function SetUnicode( Unicode: Boolean ): PControl;
```
Sets control as Unicode or not. The control itself is returned as for other "transparent" functions. A conditional define UNICODE_CTRLS must be added to a project to provide handling unicode messages.

```
function TC_Insert( Idx: Integer; const TabText: KOLString; TabImgIdx: Integer ):
PControl;
```
Inserts new tab before given, returns correspondent page control (which can be used as a parent for controls to place on the page).

```
procedure TC_Delete( Idx: Integer );
```
Removes tab from tab control, destroying all its child controls.

```
procedure TC_InsertControl( Idx: Integer; const TabText: KOLString; TabImgIdx:
Integer; Page: PControl );
```
Inserts new tab before given, but not construt this Page (this control must be created before inserting, and may be not a Panel).

```
function TC_Remove( Idx: Integer ): PControl;
```
Only removes tab from tab control, and return this Page as Result.

```
procedure TC_SetPadding( cx, cy: Integer );
```
Sets space padding around tab text in a tab of tab control.

```
function TC_TabAtPos( x, y: Integer ): Integer;
```
Returns index of tab, found at the given position (relative to a client rectangle of tab control). If no tabs found at the position, -1 is returned.

```
function TC_DisplayRect: TRect;
```
Returns rectangle, occupied by a page rather then tab.

© 2024 Carl Peeraer

```
function TC_IndexOf( const S: KOLString ): Integer;
```
By Mr Brdo. Index of page by its Caption 219.

```
function TC_SearchFor( const S: KOLString; StartAfter: Integer; Partial: Boolean ):
Integer;
```
By Mr Brdo. Index of page by its Caption 219.

```
procedure LVColAdd( const aText: KOLString; aalign: TTextAlign 206; aWidth: Integer );
```
Adds new column. Pass 'width' <= 0 to provide default column width. 'text' is a column header text.

```
procedure LVColInsert( ColIdx: Integer; const aText: KOLString; aAlign: TTextAlign 206
; aWidth: Integer );
```
Inserts new column at the Idx position (1-based column index).

```
procedure LVColDelete( ColIdx: Integer );
```
Deletes column from List View

```
function LVNextItem( IdxPrev: Integer; Attrs: DWORD ): Integer;
```
Returns an index of the next after IdxPrev item with given attributes in the list view. Attributes can be: LVNI_ALL - Searches for a subsequent item by index, the default value.

Searchs by physical relationship to the index of the item where the search is to begin. LVNI_ABOVE - Searches for an item that is above the specified item. LVNI_BELOW - Searches for an item that is below the specified item. LVNI_TOLEFT - Searches for an item to the left of the specified item. LVNI_TORIGHT - Searches for an item to the right of the specified item.

The state of the item to find can be specified with one or a combination of the following values: LVNI_CUT - The item has the LVIS_CUT state flag set. LVNI_DROPHILITED - The item has the LVIS_DROPHILITED state flag set LVNI_FOCUSED - The item has the LVIS_FOCUSED state flag set. LVNI_SELECTED - The item has the LVIS_SELECTED state flag set.

```
function LVNextSelected( IdxPrev: Integer ): Integer;
```
Returns an index of next (after IdxPrev) selected item in a list view.

```
function LVAdd( const aText: KOLString; ImgIdx: Integer; State: TListViewItemState;
StateImgIdx, OverlayImgIdx: Integer; Data: DWORD ): Integer;
```
Adds new line to the end of ListView control. Only content of item itself is set (aText, ImgIdx). To change other column text and attributes of item added, use appropriate properties / methods (). Returns an index of added item.
There is no Unicode version defined, use LVItemAddW instead.

© 2024 Carl Peeraer

```
function LVItemAdd( const aText: KOLString ): Integer;
```
Adds an item to the end of list view. Returns an index of the item added.

```
function LVInsert( Idx: Integer; const aText: KOLString; ImgIdx: Integer; State:
TListViewItemState; StateImgIdx, OverlayImgIdx: Integer; Data: DWORD ): Integer;
```
Inserts new line before line with index Idx in ListView control. Only content of item itself is set (aText, ImgIdx). To change other column text and attributes of item added, use appropriate properties / methods (). if ImgIdx = I_IMAGECALLBACK, event handler OnGetLVItemImgIdx is responsible for returning image index for an item (/// not implemented yet ///) Pass StateImgIdx and OverlayImgIdx = 0 (ignored in that case) or 1..15 to use correspondent icon from ImageListState 229 image list.
Returns an index of item inserted.
There is no unicode version of this method, use LVItemInsertW.

```
function LVItemInsert( Idx: Integer; const aText: KOLString ): Integer;
```
Inserts an item to Idx position.

```
procedure LVDelete( Idx: Integer );
```
Deletes item of ListView with subitems (full row - in lvsDetail view style.

```
procedure LVSetItem( Idx, Col: Integer; const aText: KOLString; ImgIdx: Integer;
State: TListViewItemState; StateImgIdx, OverlayImgIdx: Integer; Data: DWORD );
```
Use this method to set item data and item columns data for ListView control. It is possible to pass I_SKIP as ImgIdx, StateImgIdx, OverlayImgIdx values to skip setting this fields. But all other are set always. Like in LVInsert 258 / LVAdd 257, ImgIdx can be I_IMAGECALLBACK to determine that image will be retrieved in OnGetItemImgIdx event handler when needed.
If this method is called to set data for column > 0, parameters ImgIdx and Data are ignored anyway.
There is no unicode version of this method, use other methods to set up listed properties separately using correspondent W-functions.

```
procedure LVSelectAll;
```
Call this method to select all the items of the list view control.

```
function LVItemRect( Idx: Integer; Part: TGetLVItemPart ): TRect;
```
Returns rectangle occupied by given item part(s) in ListView window. Empty rectangle is returned, if the item is not viewing currently.

```
function LVSubItemRect( Idx, ColIdx: Integer ): TRect;
```
Returns rectangle occupied by given item's subitem in ListView window, in lvsDetail or lvsDetailNoHeader style. Empty rectangle (0,0,0,0) is returned if the item is not viewing currently. Left 212 or/and right bounds of the rectangle returned can be outbound item rectangle if only a

© 2024 Carl Peeraer

part of the subitem is visible or the subitem is not visible in the item, which is visible itself.

`function LVItemAtPos( X, Y: Integer ): Integer;`
Return index of item at the given position.

`function LVItemAtPosEx( X, Y: Integer; var Where: TWherePosLVItem 207 ): Integer;`
Retrieves index of item and sets in Where, what part of item is under given coordinates. If there are no items at the specified position, -1 is returned.

`procedure LVMakeVisible( Item: Integer; PartiallyOK: Boolean );`
Makes listview item visible. Ignored when Item passed < 0.

`procedure LVEditItemLabel( Idx: Integer );`
Begins in-place editing of item label (first column text).

`procedure LVSort;`
Initiates sorting of list view items. This sorting procedure is available only for Win2K, WinNT4 with IE5, Win98 or Win95 with IE5. See also LVSortData 259.

`procedure LVSortData;`
Initiates sorting of list view items. This sorting procedure is always available in Windows95/98, NT/2000. But OnCompareLVItems 274 procedure receives not indexes of items compared but its Data field associated instead.

`procedure LVSortColumn( Idx: Integer );`
This is a method to simplify sort by column. Just call it in your OnColumnClick 274 event passing column index and enjoy with your list view sorted automatically when column header is clicked. Requieres Windows2000 or Winows98, not supported under WinNT 4.0 and below and under Windows95.
Either lvoSortAscending or lvoSortDescending option must be set in LVOptions 229, otherwise no sorting is performed.

`function LVIndexOf( const S: KOLString ): Integer;`
Returns first list view item index with caption matching S. The same as LVSearchFor 259(S, -1, FALSE).

`function LVSearchFor( const S: KOLString; StartAfter: Integer; Partial: Boolean ): Integer;`
Searches an item with Caption 219 equal to S (or starting from S, if Partial = TRUE). Searching is started after an item specified by StartAfter parameter.

```
function TVInsert( nParent, nAfter: THandle; const Txt: KOLString ): THandle;
```

Inserts item to a tree view. If nParent is 0 or TVI_ROOT, the item is inserted at the root of tree view. It is possible to pass following special values as nAfter parameter:

TVI_FIRST	Inserts the item at the beginning of the list.
TVI_LAST	Inserts the item at the end of the list.
TVI_SORT	Inserts the item into the list in alphabetical order.

```
procedure TVDelete( Item: THandle );
```

Removes an item from the tree view. If value TVI_ROOT is passed, all items are removed.

```
function TVItemPath( Item: THandle; Delimiter: KOLChar ): KOLString;
```

Returns full path from the root item to given item. Path is calculated as a concatenation of all parent nodes text strings, separated by given delimiter character.
Please note, that returned path has no trailing delimiter, this character is only separating different parts of the path.
If Item is not specified (=0), path is returned for Selected item.

```
function TVItemAtPos( x, y: Integer; var Where: DWORD ): THandle;
```

Returns handle of item found at specified position (relative to upper left corener of client area of the tree view). If no item found, 0 is returned. Variable Where receives additional flags combination, describing more detailed, on which part of item or tree view given point is located, such as:

TVHT_ABOVE	Above the client area
TVHT_BELOW	Below the client area
TVHT_NOWHERE	In the client area, but below the last item
TVHT_ONITEM	On the bitmap or label associated with an item
TVHT_ONITEMBUTTON	On the button associated with an item
TVHT_ONITEMICON	On the bitmap associated with an item
TVHT_ONITEMINDENT	In the indentation associated with an item
TVHT_ONITEMLABEL	On the label (string) associated with an item
TVHT_ONITEMRIGHT	In the area to the right of an item
TVHT_ONITEMSTATEICON	On the state icon for a tree-view item that is in a user-defined state
TVHT_TOLEFT	To the right of the client area
TVHT_TORIGHT	To the left of the client area

```
procedure TVExpand( Item: THandle; Flags: DWORD );
```

Call it to expand/collapse item's child nodes. Possible values for Flags parameter are:
TVE_COLLAPSE Collapses the list. TVE_COLLAPSERESET Collapses the list and removes the child items. Note that TVE_COLLAPSE must also be specified. TVE_EXPAND Expands the list.
TVE_TOGGLE Collapses the list if it is currently expanded or expands it if it is currently collapsed.

© 2024 Carl Peeraer

```
procedure TVSort( N: THandle );
```
By Alex Mokrov. Sorts treeview. If N = 0, entire treeview is sorted. Otherwise, children of the given node only.

```
procedure TVEditItem( Item: THandle );
```
Begins editing given item label in tree view.

```
procedure TVStopEdit( Cancel: Boolean );
```
Ends editing item label, started by user or explicitly by TVEditItem 261 method.

```
procedure TBAddBitmap( Bitmap: HBitmap );
```
Adds bitmaps to a toolbar. You can pass special values as Bitmap to add one of predefined system button images bitmaps:

THandle(-1) to add standard small icons,
THandle(-2) to add standard large icons,
THandle(-5) to add standard small view icons,
THandle(-6) to add standard large view icons,
THandle(-9) to add standard small history icons,
THandle(-10) to add standard large history icons, (in that case use following values as indexes to the standard and view bitmaps:

STD_COPY, STD_CUT, STD_DELETE, STD_FILENEW, STD_FILEOPEN, STD_FILESAVE, STD_FIND, STD_HELP, STD_PASTE, STD_PRINT, STD_PRINTPRE, STD_PROPERTIES, STD_REDO, STD_REPLACE, STD_UNDO,
VIEW_LARGEICONS, VIEW_SMALLICONS, VIEW_LIST, VIEW_DETAILS, VIEW_SORTNAME, VIEW_SORTSIZE, VIEW_SORTDATE, VIEW_SORTTYPE (use it as parameters BtnImgIdxArray in TBAddButtons 261 or TBInsertButtons 262 methods, and in assigning value to TBButtonImage 235[ ] property). Added bitmaps have indeces starting from previous count of images (as these are appended to existing - if any).
Note, that if You add your own (custom) bitmap, it is not transparent. Do not assume that clSilver is always equal to clBtnFace. Use API function CreateMappedBitmap to load bitmap from resource and map desired colors as you wish (e.g., convert clTeal to clBtnFace). Or, call defined in KOL function LoadMappedBitmap to do the same more easy. Unfortunately, resource identifier for bitmap to pass it to LoadMappedBitmap or to CreateMappedBitmap seems must be integer, so it is necessary to create rc-file manually and compile using Borland Resource Compiler to figure it out.

```
function TBAddButtons( const Buttons: array of PKOLChar; const BtnImgIdxArray: array
of Integer ): Integer;
```
Adds buttons to toolbar. Last string in Buttons array *must* be empty ('' or nil), so to add buttons without text, pass ' ' string (one space char). It is not necessary to provide image indexes for all buttons (it is sufficient to assign index for first button only). But in place, correspondent to separator button (defined by string '-'), any integer must be passed to assign follow image

indexes correctly. See example. To add check buttons, use prefix '+' or '-' in button definition string. If next character is '!', such buttons are grouped to a radio-group. Also, it is possible to use '^' prefix (must be first) to define button with small drop-down section (use also OnTBDropDown 276 event to respond to clicking drop down section of such buttons).
This function returns command id for first added button (other id's can be calculated incrementing the result by one for each button, except separators, which have no command id).
Note: for static toolbar (single in application and created once) ids are started from value 100.

`function **TBInsertButtons**( BeforeIdx: Integer; Buttons: array of PKOLChar; const BtnImgIdxArray: array of Integer ): Integer;`
Inserts buttons before button with given index on toolbar. Returns command identifier for first button inserted (other can be calculated incrementing returned value needed times. See also TBAddButtons 261.

`procedure **TBDeleteButton**( BtnID: Integer );`
Deletes single button given by its command id. To delete separator, use TBDeleteBtnByIdx 262 instead.

`procedure **TBDeleteBtnByIdx**( Idx: Integer );`
Deletes single button given by its index in toolbar (not by command ID).

`procedure **TBClear**;`
Deletes all buttons. Dufa

`procedure **TBAssignEvents**( BtnID: Integer; Events: array of TOnToolbarButtonClick );`
Allows to assign separate OnClick 271 events for every toolbar button. BtnID should be toolbar button ID or index of the first button to assign event. If it is an ID, events are assigned to buttons in creation order. Otherwise, events are assigned in placement order. Anyway, separator buttons are not skipped, so pass at least nil for such button as an event.
Please note, that though not all buttons should exist before assigning events to it, therefore at least the first button (specified by BtnID) must be already added before calling TBAssignEvents.

`procedure **TBResetImgIdx**( BtnID, BtnCount: Integer );`
Resets image index for BtnCount buttons starting from BtnID.

`function **TBItem2Index**( BtnID: Integer ): Integer;`
Converts button command id to button index for tool bar.

`function **TBIndex2Item**( Idx: Integer ): Integer;`
Converts toolbar button index to its command ID..

© 2024 Carl Peeraer

`procedure` **`TBConvertIdxArray2ID`**`( const IdxVars: array of PDWORD );`
Converts toolbar button indexes to its command IDs for an array of indexes (each item in the array passed is a pointer to Integer, containing button index when the procedure is callled, then all these indexes are relaced with a correspondent button ID).

`function` **`TBButtonSeparator`**`( BtnID: Integer ): Boolean;`
Returns TRUE, if a toolbar button is separator.

`function` **`TBButtonAtPos`**`( X, Y: Integer ): Integer;`
Returns command ID of button at the given position on toolbar, or -1, if there are no button at the position. Value 0 is returned for separators.

`function` **`TBBtnIdxAtPos`**`( X, Y: Integer ): Integer;`
Returns index of button at the given position on toolbar. This also can be index of separator button. -1 is returned if there are no buttons found at the position.

`function` **`TBBtnEvent`**`( Idx: Integer ): TOnToolbarButtonClick;`
Returns toolbar event handler assigned to a toolbar button (by its index).

`function` **`TBMoveBtn`**`( FromIdx, ToIdx: Integer ): Boolean;`
By TR"]F. Moves button from one position to another.

`procedure` **`TBSetTooltips`**`( BtnID1st: Integer; const Tooltips: array of PKOLChar );`
Allows to assign tooltips to several buttons. Until this procedure is not called, tooltips list is not created and no code is added to executable. This method of tooltips maintainance for toolbar buttons is useful both for static and dynamic toolbars (meaning "dynamic" - toolbars with buttons, deleted and inserted at run-time).

`function` **`TBBtnTooltip`**`( BtnID: Integer ): KOLString;`
Returns tooltip assigned to a toolbar button.

`function` **`PlaceRight`**`: PControl;`
Places control right (to previously created on the same parent).

`function` **`PlaceDown`**`: PControl;`
Places control below (to previously created on the same parent). Left 212 position is not changed (thus is, kept equal to Parent.Margin).

`function` **`PlaceUnder`**`: PControl;`
Places control below (to previously created one, aligning its Left 212 position to Left 212 position of previous control).

```
function SetSize( W, H: Integer ): PControl;
```
Changes size of a control. If W or H less or equal to 0, correspondent size is not changed.

```
function Size( W, H: Integer ): PControl;
```
Like SetSize 264, but provides automatic resizing of parent control (recursively). Especially useful for aligned controls.

```
function SetClientSize( W, H: Integer ): PControl;
```
Like SetSize 264, but works setting W = ClientWidth 213, H = ClientHeight 213. Use this method for forms, which can not be resized (dialogs).

```
function MakeWordWrap: PControl;
```
Determines if to autosize control (like label, button, etc.)

```
function IsAutoSize: Boolean;
```
TRUE, if a control is autosizing.

```
function AlignLeft( P: PControl ): PControl;
```
assigns Left 212 := P.Left

```
function AlignTop( P: PControl ): PControl;
```
assigns Top 212 := P.Top

```
function ResizeParent: PControl;
```
Resizes parent, calling ResizeParentRight 264 and ResizeParentBottom 264.

```
function ResizeParentRight: PControl;
```
Resizes parent right edge (Margin 222 of parent is added to right coordinate of a control). If called second time (for the same parent), resizes only for increasing of right edge of parent.

```
function ResizeParentBottom: PControl;
```
Resizes parent bottom edge (Margin 222 of parent is added to bottom coordinate of a control).

```
function CenterOnParent: PControl;
```
Centers control on parent, or if applied to a form, centers form on screen.

```
function CenterOnForm( Form1: PControl ): PControl;
```
Centers form on another form. If Form1 not present, centers on screen.

```
function CenterOnCurrentScreen: PControl;
```
Centers on a display where a mouse is located now. For forms only.

```
function Shift( dX, dY: Integer ): PControl;
```


© 2024 Carl Peeraer

Moves control respectively to current position (Left 212 := Left 212 + dX, Top 212 := Top 212 + dY).

```
function SetPosition( X, Y: Integer ): PControl;
```
Moves control directly to the specified position.

```
function Tabulate: PControl;
```
Call it once for form/applet to provide tabulation between controls on form/on all forms using TAB / SHIFT+TAB and arrow keys.

```
function TabulateEx: PControl;
```
Call it once for form/applet to provide tabulation between controls on form/on all forms using TAB / SHIFT+TAB and arrow keys. Arrow keys are used more smart, allowing go to nearest control in certain direction.

```
function SetAlign( AAlign: TControlAlign ): PControl;
```
Assigns passed value to property Align 245, aligning control on parent, and returns @Self (so it is "transparent" function, which can be used to adjust control at the creation, e.g.:
```
MyLabel := NewLabel( MyForm, 'Label1' ).SetAlign( caBottom );
```

```
function SetChecked( const Value: Boolean ): PControl;
```
Use it to check/uncheck check box control or push button. Do not apply it to check radio buttons - use SetRadioChecked 265 method below.

```
function SetRadioChecked: PControl;
```
Use it to check radio button item correctly (unchecking all alternative ones). Actually, method Click 265 is called, and control itself is returned.

```
procedure Click;
```
Emulates click on control programmatically, sending WM_COMMAND message with BN_CLICKED code. This method is sensible only for buttons, checkboxes and radioboxes.

```
function Perform( msgcode: DWORD; wParam, lParam: Integer ): Integer; stdcall;
```
Sends message to control's window (created if needed).

```
function Postmsg( msgcode: DWORD; wParam, lParam: Integer ): Boolean; stdcall;
```
Sends message to control's window (created if needed).

```
procedure AttachProc( Proc: TWindowFunc );
```
It is possible to attach dynamically any message handler to window procedure using this method. Last attached procedure is called first. If procedure returns True, further processing of a message is stopped. Attached procedure can be detached using DetachProc 266 (but do not attach/detach procedures during handling of attached procedure - this can hang application).

© 2024 Carl Peeraer

```
procedure AttachProcEx( Proc: TWindowFunc; ExecuteAfterAppletTerminated: Boolean );
```
The same as AttachProc 265, but a handler is executed even after terminating the main message loop processing (i.e. after assigning true to AppletTerminated global variable.

```
function IsProcAttached( Proc: TWindowFunc ): Boolean;
```
Returns True, if given procedure is already in chain of attached ones for given control window proc.

```
procedure DetachProc( Proc: TWindowFunc );
```
Detaches procedure attached earlier using AttachProc 265.

```
procedure SetAutoPopupMenu( PopupMenu: PObj );
```
To assign a popup menu to the control, call SetAutoPopupMenu method of the control with popup menu object as a parameter.

```
function SupportMnemonics: PControl;
```
This method provides supporting mnemonic keys in menus, buttons, checkboxes, toolbar buttons.

```
function LBItemAtPos( X, Y: Integer ): Integer;
```
Return index of item at the given position.

```
function RE_TextSizePrecise: Integer;
```
By Savva. Returns length of rich edit text.

```
function RE_FmtStandard: PControl;
```
"Transparent" method (returns @Self as a result), which (when called) provides "standard" keyboard interface for formatting Rich text (just call this method, for example:
```
RichEd1 := NewRichEdit( Panel1, [ ] ).SetAlign( caClient ).RE_FmtStandard;
```

Following keys will be maintained additionally:

• CTRL+I	switch "Italic"
• CTRL+B	switch "Bold"
• CTRL+U	switch "Underline"
• CTRL+SHIFT+U	switch underline type and turn underline on (note, that some of underline styles can not be shown properly in RichEdit v2.0 and lower, though RichEdit2.0 stores data successfully).
• CTRL+O	switch "StrikeOut"
• CTRL+'gray+'	increase font size
• CTRL+'gray-'	decrease font size

© 2024 Carl Peeraer

- CTRL+SHIFT+'gray+' superscript
- CTRL+SHIFT+'gray-' subscript
- CTRL+SHIFT+Z ReDo

And, though following standard formatting keys are provided by RichEdit control itself in Windows2000, some of these are not functioning automatically in earlier Windows versions, even for RichEdit2.0. So, functionality of some of these (marked with (*)) are added here too:

- CTRL+L align paragraph left
- CTRL+R align paragraph right
- CTRL+E align paragraph center
- CTRL+A select all, double-click on word - select word
- CTRL+Right to next word
- CTRL+Left 212 to previous word
- CTRL+Home to the beginning of text
- CTRL+End to the end of text
- CTRL+Z UnDo

If You originally assign some (plain) text to Text 219 property, switching "underline" can also change other font attributes, e.g., "bold" - if fsBold style is in default Font 221. To prevent such behavior, select entire text first (see SelectAll 250) and make assignment to RE_Font 238 property, e.g.:

```
RichEd1.SelectAll;
RichEd1.RE_Font := RichEd1.RE_Font;
RichEd1.SelLength := 0;
```

And, some other notices about formatting. Please remember, that only True Type fonts can be succefully scaled and transformed to get desired effects (e.g., bold). By default, RichEdit uses System font face name, which can even have problems with fsBold style. Please remember also, that assigning RE_Font 238 to RE_Font 238 just initializying formatting attributes, making all those valid in entire text, but does not change font attributes. To use True Type font, directly assign face name You wish, e.g.:

```
RichEd1.SelectAll;
RichEd1.RE_Font := RichEd1.RE_Font;
RichEd1.RE_Font.FontName := 'Arial';
RichEd1.SelLength := 0;
```

procedure **RE_CancelFmtStandard**;
Cancels RE_FmtStandard 266 (detaching window procedure handler).

function **RE_LoadFromStream**(Stream: PStream; Length: Integer; Format: TRETextFormat; SelectionOnly: Boolean): Boolean;
Use this method rather then assignment to RE_Text 244 property, if source is stored in file or stream (to minimize resources during loading of RichEdit content). Data is loading starting from current position in stream and no more then Length bytes are loaded (use -1 value to load to the end of stream). Loaded data replaces entire content of RichEdit control, or selection only,

depending on SelectionOnly flag.
If You want to provide progress (e.g. in form of progress bar), assign OnProgress 277 event to your handler - and to examine current position of loading, read TSream.Position property of soiurce stream).

```
function RE_SaveToStream( Stream: PStream; Format: TRETextFormat; SelectionOnly: Boolean ): Boolean;
```

Use this method rather then RE_TextProperty to store data to file or stream (to minimize resources during saving of RichEdit content). Data is saving starting from current position in a stream (until end of RichEdit data). If SelectionOnly flag is True, only selected part of RichEdit text is saved.
Like for RE_LoadFromStream 267, it is possible to assign your method to OnProgress 277 event (but to calculate progress of save-to-stream operation, compare current stream position with RE_Size[rsBytes] property value).

```
function RE_LoadFromFile( const Filename: KOLString; Format: TRETextFormat; SelectionOnly: Boolean ): Boolean;
```

Use this method rather then other assignments to RE_Text 244 property, if a source for RichEdit is the file. See also RE_LoadFromStream 267.

```
function RE_SaveToFile( const Filename: KOLString; Format: TRETextFormat; SelectionOnly: Boolean ): Boolean;
```

Use this method rather then other similar, if You want to store entire content of RichEdit or selection only of RichEdit to a file.

```
procedure RE_Append( const S: KOLString; ACanUndo: Boolean );
```

```
procedure RE_InsertRTF( const S: KOLString );
```

```
procedure RE_HideSelection( aHide: Boolean );
```

Allows to hide / show selection in RichEdit.

```
function RE_SearchText( const Value: KOLString; MatchCase, WholeWord, ScanForward: Boolean; SearchFrom, SearchTo: Integer ): Integer;
```

Searches given string starting from SearchFrom position up to SearchTo position (to the end of text, if SearchTo is -1). Returns zero-based character position of the next match, or -1 if there are no more matches. To search in bacward direction, set ScanForward to False, and pass SearchFrom > SearchTo (or even SearchFrom = -1 and SearchTo = 0).

```
function RE_WSearchText( const Value: KOLWideString; MatchCase, WholeWord, ScanForward: Boolean; SearchFrom, SearchTo: Integer ): Integer;
```

Searches given string starting from SearchFrom position up to SearchTo position (to the end of text, if SearchTo is -1). Returns zero-based character position of the next match, or -1 if there

© 2024 Carl Peeraer

are no more matches. To search in bacward direction, set ScanForward to False, and pass SearchFrom > SearchTo (or even SearchFrom = -1 and SearchTo = 0).

`function` **`RE_NoOLEDragDrop:`** `PControl;`
Just prevents drop OLE objects to the rich edit control. Seems not working for some cases.

`function` **`CanUndo:`** `Boolean;`
Returns True, if the edit (or RichEdit) control can correctly process the EM_UNDO message.

`procedure` **`EmptyUndoBuffer;`**
Reset the undo flag of an edit control, preventing undoing all previous changes.

`function` **`Undo:`** `Boolean;`
For a single-line edit control, the return value is always TRUE. For a multiline edit control and RichEdit control, the return value is TRUE if the undo operation is successful, or FALSE if the undo operation fails.

`procedure` **`FreeCharFormatRec;`**
Only for RichEdit control: Returns True if successful.

TControl events

`property` **`OnHelp:`** `TOnHelp;`
An event of a form, it is called when F1 pressed or help topic requested by any other way. To prevent showing help, nullify Sender. Set Popup to TRUE to provide showing help in a pop-up window. It is also possible to change Context dynamically.

`property` **`OnDropDown:`** `TOnEvent;`
Is called when combobox is dropped down (or drop-down button of toolbar is pressed - see also OnTBDropDown276).

`property` **`OnCloseUp:`** `TOnEvent;`
Is called when combobox is closed up. When drop down list is closed because user pressed "Escape" key, previous selection is restored. To test if it is so, call GetKeyState(VK_ESCAPE) and check, if negative value is returned (i.e. Escape key is pressed when event handler is calling).

`property` **`OnBitBtnDraw:`** `TOnBitBtnDraw;`
Special event for BitBtn. Using it, it is possible to provide additional effects, such as highlighting button text (by changing its Font221 and other properties). If the handler returns True, it is supposed that it made all drawing and there are no further drawing occur.

© 2024 Carl Peeraer

`property` **OnMeasureItem:** `TOnMeasureItem;`
This event is called for owner-drawn controls, such as list box, combo box, list view with appropriate owner-drawn style. For fixed item height controls (list box with loOwnerDrawFixed style, combobox with coOwnerDrawFixed and list view with lvoOwnerDrawFixed option) this event is called once. For list box with loOwnerDrawVariable style and for combobox with coOwnerDrawVariable style this event is called for every item.

`property` **OnShow:** `TOnEvent;`
Is called when a control or form is to be shown. This event is not fired for a form, if its WindowState 221 initially is set to wsMaximized or wsMinimized. This behaviour is by design (the window does not receive WM_SHOW message in such case).

`property` **OnHide:** `TOnEvent;`
Is called when a control or form becomes hidden.

`property` **OnMessage:** `TOnMessage;`
Is called for every message processed by TControl object. And for Applet window, this event is called also for all messages, handled by all its child windows (forms).

`property` **OnClose:** `TOnEventAccept;`
Called before closing the window. It is possible to set Accept parameter to False to prevent closing the window. This event events is not called when windows session is finishing (to handle this event, handle WM_QUERYENDSESSION message, or assign OnQueryEndSession 270 event to another or the same event handler).

`property` **OnQueryEndSession:** `TOnEventAccept;`
Called when WM_QUERYENDSESSION message come in. It is possible to set Accept parameter to False to prevent closing the window (in such case session ending is halted). It is possible to check CloseQueryReason 225 property to find out, why event occur.
To provide normal application close while handling OnQueryEndSession, call in your code PostQuitMessage(0) or call method Close 250 for the main form, this is enough to provide all OnClose 270 and OnDestroy handlers to be called.

`property` **OnMinimize:** `TOnEvent;`
Called when window is minimized.

`property` **OnMaximize:** `TOnEvent;`
Called when window is maximized.

`property` **OnRestore:** `TOnEvent;`

© 2024 Carl Peeraer

Called when window is restored from minimized or maximized state.

property **OnPaint:** TOnPaint;
Event to set to override standard control painting. Can be applied to any control (though originally was designed only for paintbox control). When an event handler is called, it is possible to use UpdateRgn 225 to examine what parts of window require painting to improve performance of the painting operation.

property **OnPrePaint:** TOnPaint;
Only for graphic controls. If you assign it, call Invalidate 248 also.

property **OnPostPaint:** TOnPaint;
Only for graphic controls. If you assign it, call Invalidate 248 also.

property **OnEraseBkgnd:** TOnPaint;
This event allows to override erasing window background in response to WM_ERASEBKGND message. This allows to add some decorations to standard controls without overriding its painting in total. Note: When erase background, remember, that property ClientRect 248 can return not true client rectangle of the window - use GetClientRect API function instead. For example:

```
var BkBmp: HBitmap;
procedure TForm1.KOLForm1FormCreate(Sender: PObj);
begin
  Toolbar1.OnEraseBkgnd := DecorateToolbar;
  BkBmp := LoadBitmap( hInstance, 'BK1' );
end;

procedure TForm1.DecorateToolbar(Sender: PControl; DC: HDC);
var CR: TRect;
begin
  GetClientRect( Sender.Handle, CR );
  Sender.Canvas.Brush.BrushBitmap := BkBmp;
  Sender.Canvas.FillRect( CR );
end;
```

property **OnClick:** TOnEvent;
Called on click at control. For buttons, checkboxes and radioboxes is called regadless if control clicked by mouse or keyboard. For toolbar, the same event is used for all toolbar buttons and toolbar itself. To determine which toolbar button is clicked, check CurIndex 219 property. And note, that all the buttons including separator buttons are enumerated starting from 0. Though images are stored (and prepared) only for non-separator buttons. And to determine, if toolbar button was clicked with right mouse button, check RightClick 225 property.
This event does not work on a Form, still it is fired in responce to WM_COMMAND window message mainly rather direct to mouse down. But, if you want to have OnClick event to be fired on a Form, use (following) property OnFormClick 271 to assign it.

© 2024 Carl Peeraer

```
property OnFormClick: TOnEvent;
```
Assign you OnClick 271 event handler using this property, if you want it to be fired in result of mouse click on a form surface. Use to assign the event only for forms (to avoid doublicated firing the handler).
Note: for a form, in case of WM_xDOUBLECLK event, this event is fired for both clicks. So if you install both OnFormClick and OnMouseDblClk 273, handlers will be called in the following sequence for each double click: OnFormClick; OnMouseDblClk 273; OnFormClick.

```
property OnEnter: TOnEvent;
```
Called when control receives focus.

```
property OnLeave: TOnEvent;
```
Called when control looses focus.

```
property OnChange: TOnEvent;
```
Called when edit control is changed, or selection in listbox or current index in combobox is changed (but if OnSelChanged assigned, the last is called for change selection). To respond to check/uncheck checkbox or radiobox events, use OnClick 271 instead.

```
property OnSelChange: TOnEvent;
```
Called for rich edit control, listbox, combobox or treeview when current selection (range, or current item) is changed. If not assigned, but OnChange 272 is assigned, OnChange 272 is called instead.

```
property OnResize: TOnEvent;
```
Called whenever control receives message WM_SIZE (thus is, if control is resized.

```
property OnMove: TOnEvent;
```
Called whenever control receives message WM_MOVE (i.e. when control is moved over its parent).

```
property OnMoving: TOnEventMoving;
```
Called whenever control receives message WM_MOVE (i.e. when control is moved over its parent).

```
property OnSplit: TOnSplit;
```
Called when splitter control is dragging - to allow for your event handler to decide if to accept new size of left (top) control, and new size of the rest area of parent.

```
property OnKeyDown: TOnKey;
```
Obvious.

© 2024 Carl Peeraer

`property` **`OnKeyUp`**`: TOnKey;`
Obvious.

`property` **`OnChar`**`: TOnChar;`
Deprecated event, use OnKeyChar 273.

`property` **`OnKeyChar`**`: TOnChar;`
Obviuos.

`property` **`OnKeyDeadChar`**`: TOnChar;`
Obviuos.

`property` **`OnMouseUp`**`: TOnMouse;`
Obvious.

`property` **`OnMouseDown`**`: TOnMouse;`
Obvious.

`property` **`OnMouseMove`**`: TOnMouse;`
Obvious.
.

`property` **`OnMouseDblClk`**`: TOnMouse;`
Obvious.

`property` **`OnMouseWheel`**`: TOnMouse;`
Mouse wheel (up or down) event. In Windows, only focused controls and controls having scrollbars (or a scrollbar iteself) receive such message. To get direction and amount of wheel, use typecast: SmallInt(HiWord(Mouse.Shift)). Value 120 corresponds to one wheel step (-120 - for step back).

`property` **`OnMouseEnter`**`: TOnEvent;`
Is called when mouse is entered into control. See also OnMouseLeave 273.

`property` **`OnMouseLeave`**`: TOnEvent;`
Is called when mouse is leaved control. If this event is assigned, then mouse is captured on mouse enter event to handle all other mouse events until mouse cursor leaves the control.

property **OnTestMouseOver**: TOnTestMouseOver 209;
Special event, which allows to extend OnMouseEnter 273 / OnMouseLeave 273 (and also Flat 226 property for **BitBtn** control). If a handler is assigned to this event, actual testing whether mouse

© 2024 Carl Peeraer

is in control or not, is occuring in the handler. So, it is possible to simulate more careful hot tracking for controls with non-rectangular shape (such as glyphed BitBtn control).

`property` **`OnEndEditLVItem:`** `TOnEditLVItem;`
Called when edit of an item label in ListView control finished. Return True to accept new label text, or false - to not accept it (item label will not be changed). If handler not set to an event, all changes are accepted.

`property` **`OnLVDelete:`** `TOnDeleteLVItem;`
This event is called when an item is deleted in the listview. Do not add, delete, or rearrange items in the list view while processing this notification.

`property` **`OnDeleteLVItem:`** `TOnDeleteLVItem;`
Called for every deleted list view item.

`property` **`OnDeleteAllLVItems:`** `TOnEvent;`
Called when all the items of the list view control are to be deleted. If after returning from this event handler event OnDeleteLVItem 274 is yet assigned, an event OnDeleteLVItem 274 will be called for every deleted item.

`property` **`OnLVData:`** `TOnLVData;`
Called to provide virtual list view with actual data. To use list view as virtaul list view, define also lvsOwnerData style and set Count 219 property to actual row count of the list view. This manner of working with list view control can greatly improve performance of an application when working with huge data sets represented in listview control.

`property` **`OnCompareLVItems:`** `TOnCompareLVItems;`
Event to compare two list view items during sort operation (initiated by LVSort 259 method call). Do not send any messages to the list view control while it is sorting - results can be unpredictable!

`property` **`OnColumnClick:`** `TOnLVColumnClick;`
This event handler is called when column of the list view control is clicked. You can use this event to initiate sorting of list view items by this column.

`property` **`OnLVStateChange:`** `TOnLVStateChange;`
This event occure when an item or items range in list view control are changing its state (e.g. selected or unselected).

`property` **`OnDrawItem:`** `TOnDrawItem;`
This event can be used to implement custom drawing for list view, list box, dropped list of a

© 2024 Carl Peeraer

combobox. For a list view, custom drawing using this event is possible only in lvsDetail and lvsDetailNoHeader styles, and OnDrawItem is called to draw entire row at once only. See also OnLVCustomDraw 275 event.

property **OnLVCustomDraw**: TOnLVCustomDraw;

Custom draw event for listview. For every item to be drawn, this event can be called several times during a single drawing cycle - depending on a result, returned by an event handler. Stage can have one of following values:

CDDS_PREERASE
CDDS_POSTERASE
CDDS_ITEMPREERASE
CDDS_PREPAINT
CDDS_ITEMPREPAINT
CDDS_ITEM
CDDS_SUBITEM + CDDS_ITEMPREPAINT
CDDS_SUBITEM + CDDS_ITEMPOSTPAINT
CDDS_ITEMPOSTPAINT
CDDS_POSTPAINT

When called, see on Stage to get know, on what stage the event is activated. And depend on the stage and on what you want to paint, return a value as a result, which instructs the system, if to use default drawing on this (and follows) stage(s) for the item, and if to notify further about different stages of drawing the item during this drawing cycle. Possible values to return are:

CDRF_DODEFAULT	perform default drawing. Do not notify further for this item (subitem) (or for entire listview, if called with flag CDDS_ITEM reset - ?)
CDRF_NOTIFYITEMDRAW	return this value, when the event is called the first time in a cycle of drawing, with ItemIdx = -1 and flag CDDS_ITEM reset in Stage parameter
CDRF_NOTIFYPOSTERASE	usually can be used to provide default erasing, if you want to perform drawing immediately after that
CDRF_NOTIFYPOSTPAINT	return this value to provide calling the event after performing default drawing. Useful when you wish redraw only a part of the (sub)item
CDRF_SKIPDEFAULT	return this value to inform the system that all drawing is done and system should not peform any more drawing for the (sub)item during this drawing cycle.
CDRF_NEWFONT	informs the system, that font is changed and default drawing should be performed with changed font;

© 2024 Carl Peeraer

If you want to get notifications for each subitem, do not use option lvoOwnerDrawFixed, because such style prevents system from notifying the application for each subitem to be drawn in the listview and only notifications will be sent about entire items.
See also NM_CUSTOMDRAW in API Help.

`property` **`OnTVBeginDrag:`** `TOnTVBeginDrag`205`;`
Is called for tree view, when its item is to be dragging.

`property` **`OnTVBeginEdit:`** `TOnTVBeginEdit`205`;`
Is called for tree view, when its item label is to be editing. Return TRUE to allow editing of the item.

`property` **`OnTVEndEdit:`** `TOnTVEndEdit`205`;`
Is called when item label is edited. It is possible to cancel edit, returning False as a result.

`property` **`OnTVExpanding:`** `TOnTVExpanding`205`;`
Is called just before expanding/collapsing item. It is possible to return TRUE to prevent expanding item, otherwise FALSE should be returned.

`property` **`OnTVExpanded:`** `TOnTVExpanded`205`;`
Is called after expanding/collapsing item children.

`property` **`OnTVDelete:`** `TOnTVDelete`205`;`
Is called just before deleting item. You may use this event to free resources, associated with an item (see TVItemData234 property).

`property` **`OnTVSelChanging:`** `TOnTVSelChanging`205`;`
Is called before changing the selection. The handler can return FALSE to prevent changing the selection.

`property` **`OnTBDropDown:`** `TOnEvent;`
This event is called for drop down buttons, when user click drop part of drop down button. To determine for which button event is called, look at CurItem or CurIndex219 property. It is also possible to use common (with combobox) property OnDropDown269.

`property` **`OnTBClick:`** `TOnEvent;`
The same as OnClick271.

`property` **`OnTBCustomDraw:`** `TOnTBCustomDraw;`
An event (mainly) to customize toolbar background.

© 2024 Carl Peeraer

`property` **OnDTPUserString:** `TDTParseInputEvent;`
Special event to parse input from the application. Option dtpoParseInput must be set when control is created.

`property` **OnSBBeforeScroll:** `TOnSBBeforeScroll;`

`property` **OnSBScroll:** `TOnSBScroll;`

`property` **OnDropFiles:** `TOnDropFiles;`
Assign this event to your handler, if You want to accept drag and drop files from other applications such as explorer onto your control. When this event is assigned to a control or form, this has effect also for all its child controls too.

`property` **OnScroll:** `TOnScroll;`

`property` **OnRE_InsOvrMode_Change:** `TOnEvent;`
This event is called, whenever key INSERT is pressed in control (and for RichEdit, this means, that insert mode is changed).

`property` **OnProgress:** `TOnEvent;`
This event is called during RE_SaveToStream 268, RE_LoadFromStream 267 (and also during RE_SaveToFile 268, RE_LoadFromFile 268 and while accessing or changing RE_Text 244 property). To calculate relative progress, it is possible to examine current position in stream/file with its total size while reading, or with rich edit text size, while writing (property RE_TextSize 238[rsBytes]).

`property` **OnRE_OverURL:** `TOnEvent;`
Is called when mouse is moving over URL. This can be used to set cursor, for example, depending on type of URL (to determine URL type read property RE_URL 245).

`property` **OnRE_URLClick:** `TOnEvent;`
Is called when click on URL detected.

© 2024 Carl Peeraer

TControl fields

FormString: KOLString;
String of the current parameter. It is cleared after each call to FormExecuteCommands 246, so no special clearing is required.

fDoubleBuffered: Boolean;
True, if cannot set DoubleBuffered 224 to True (RichEdit).

fClassicTransparent: Boolean;
True, when creating of object is in progress.

fDestroying: Boolean;
True, when destroying of the window is started.

fBeginDestroying: Boolean;
true, when destroying of the window is initiated by the system, i.e. message WM_DESTROY fired

fChangedPosSz: Byte;
Flags of changing left (1), top (2), width (4) or height (8)

fIsForm: Boolean;
True, if the object is form.

fIsApplet: Boolean;
True, if the object represent application taskbar button.

fIsControl: Boolean;
True, if it is a control on form.

fIsMDIChild: Boolean;
TRUE, if the object is MDI child form.

fIsCommonControl: Boolean;
True, if it is common control.

fWindowed: Boolean;
True, if control is windowed (or is a form). It is set to FALSE only for graphic controls.

fCtlClsNameChg: Boolean;
True, if control class name changed and memory is allocated to store it.

fChildren: PList;
List of children.

fTmpBrush: HBrush;

© 2024 Carl Peeraer

Brush 221 handle to return in response to some color set messages. Intended for internal use instead of Brush.Color if possible to avoid using it.

fMenu: HMenu;
Usually used to store handle of attached main menu, but sometimes is used to store control ID (for standard GUI controls only).

fMenuObj: PObj;
PMenu pointer to TMenu object. Freed automatically with entire chain of menu objects attached to a control (or form).

fImageList: PImageList;
Pointer to first private image list. Control can own several image, lists, linked to a chain of image list objects. All these image lists are released automatically, when control is destroyed.

fTextColor: TColor;
Color 221 of text. Used instead of fFont.Color internally to // avoid usage of Font 221 object if user is not accessing and changing it.

fColor: TColor;
Color 221 of control background.

fClientRight: ShortInt;
Store adjustment factor of ClientRect 248 for some 'idiosincrasies' windows, // such as Groupbox or Tabcontrol.

DF: **TDataFields;**
Data fields for certain controls. These are overlapped to economy size of TControl object.

fNestedMsgHandling: SmallInt;
level of nested message handling for a control. Only when it is 0 at the end of message handling and fBeginDestroying 278 set, the control is destroyed.

stdcall;
MDI client window control

4.26 Programming in KOL (without MCK)

Programming in KOL (without MCK). Create a Form and start a Message Loop.

In order to start designing a "clean" KOL project (ie without MCK), it is enough to create a project as usual and remove the first module with a form from it. After that, in the project file, remove the reference to Forms and all other VCL units from uses (replacing them with a reference to KOL), and from the body of the project code, begin ... end. delete all lines (there are two of them, and they refer to Application to initialize and launch the application). Now you can add the first lines of code:

© 2024 Carl Peeraer

```
Applet: = NewForm (nil, 'form title'); Run (Applet);
```

This is a minimal KOL form application. You can try compiling it and running it. If something doesn't work out for you, take a look at the demo project called Empty. If you do nothing else, then the size of this application in Delphi5 is 23 Kilobytes. If, in the project options, add the path to the folder with the replacement of system modules to the list of search paths, and add **SMALLEST_CODE** 38 and **NOT_USE_RICHEDIT** 42 to the list of conditional compilation symbols, then the application size is reduced to 10.5 Kilobytes.

Please note: if you compile a project in Delphi version 6 or higher, then sometimes the compiler has an incomprehensible tendency to add additional functions to the code from the Variants.pas module (which appeared in this version for the first time). The size of the application grows dramatically by several kilobytes, even if you didn't use the options. Sometimes it is possible to get rid of this module by various manipulations (reopening the project in Delphi, restarting the environment, rebuilding the project). The most efficient way is to download the FakeVariants.zip archive, unpack it (the Variants.pas file, from which everything that is not required has been removed) into a directory and specify the path to it in the project options. Well, or just unzip it into your project folder.

The **global procedure Run**, which is called in the above example, receives a window object as a parameter, calls for it to create a window, and then enters a loop of waiting and processing messages. This cycle continues until the application is terminated (i.e., until the global variable AppletTerminated is set to true). Then the global procedure **TerminateExecution** is called (if it is still needed), and this is where the application really ends.

The programmer always has the opportunity to write his own analog of the **Run** procedure, if necessary, and call it exactly. For example, in one of my applications, I changed this procedure in order to process messages from the mouse and keyboard first, before other window messages (otherwise, when multithreaded work with an increased priority, problems with reaction to the keyboard and mouse began to arise).

When programming in KOL without MCK, you should pay attention to how to assign event handlers. The easiest way is to create a custom object (derived from TObj), define a method for it that matches the handler type description, and specify that method as the event handler.

But you can use ordinary procedures instead of methods, turning them into a method using the MakeMethod (data, proc) function and not forgetting to convert the resulting method to the required handler type, for example:

```
Button1.OnClick: = TOnEvent (MakeMethod (nil, @ Button1ClickProc)
```

It should be remembered that in the declaration of the handler procedure, you must add an additional parameter of the pointer type as the first. This pointer corresponds to a pointer to an object instance, that is, the same pointer that you assigned in the first parameter in the call to the MakeMethod function will be

© 2024 Carl Peeraer

passed here. For example, to correctly handle the above button click event, the titlethe handler should look like this:

```
procedure Button1ClickProc (Dummy, Sender: PControl);
begin
 ...
```

In the case of a different type of handler with more parameters, they also follow in the usual order, but the same dummy parameter must be added first.

4.27 MCK Design

- Creation of on MCK project 281
- Form customization 285
- Coding 287

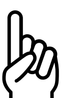
4.27.1 Creation of on MCK project

Oh, how many wonderful discoveries we have
Prepares the spirit of enlightenment!
And experience, son of difficult mistakes,
And a genius, a friend of paradoxes,
And chance, Fortune's half-brother.
(A.S. Pushkin)

Now that you've got the initial idea of the capabilities of forms, applets and - in general - other visual elements, it's time to talk a little about the development of **MCK** projects. The ability to create an **MCK** project, toss a couple of **MCK** components onto a form, configure them and launch the project will allow the reader, upon further reading, to feel what he has read through experience. For, as experience shows, there is no better way to learn than ... experience.
As already mentioned at the beginning of the presentation, the Mirror Classes Kit, i.e. a set of mirrored components for the KOL library did not appear immediately. The ideology of the KOL library denies the very possibility of using components, i.e. classes derived from **TComponent**, classes that could exist both at the stage of development and at the stage of application execution (in the second case, loading its initial state from the resources of the form, during its construction). This is how the Delphi environment works, and I must admit, this is a very convenient approach that significantly speeds up application development, and it is not for nothing that Delphi is proudly called RAD - Rapid Develpment Tool, or, translated into Russian: rapid development tool.
Nevertheless, KOL managed to build a not the worst mechanism for visual programming. **MCK** contains a set of mirror components (or simply - mirrors), roughly corresponding to the set of varieties of simple KOL objects. In the same way as in the development of a VCL project, these mirrors are thrown onto the form from the component ruler, their properties are configured visually (with the mouse, the Object Inspector, calling special component editors). And as a result of the joint work of these components at the development stage, the text of the source files of the projects is modified in such a way that, when compiled, we get a "clean" KOL project, in

which there are no classes, components, and other references to **VCL**, but only simple objects object, and a minimum of code with all the necessary functionality.
Perhaps it is worth highlighting this feature, and emphasizing once again that MCK components do not participate in the working version of the code. Their task is only to generate code for KOL, which is placed mainly in inc files, and when the application is executed, it is called to create the form along with all its children. That is, the form at run time does not create itself from the form resource, as is done in a **VCL** application, but is built dynamically by calling the appropriate NewXXXX functions, assigning initial values to properties and events in accordance with the settings of MCK components made at the design stage. ...

In fact, the **MCK** idea is not so trivial. The mirrored component code contains a number of tricks designed to "trick" the development environment. As a result, the Delphi IDE thinks that it is dealing with a regular VCL project with classes and forms loaded from dfm resources, although this is far from the case.



Creating an **MCK** project starts with the same thing as creating a VCL project, namely: in the File menu of the development environment, select the **New Application** item. As a result, the project **Project1** is created, containing three files (so far they are stored in memory): this is **Project1.dpr** - the source file of the project, **Unit1.pas** - the source file of the only form module so far, and **Unit1.dfm** - the form file. The next step that now needs to be done is to save the project in a folder. To do this, we select **File | Save All** in the menu, and we are sequentially prompted to save **Projec1.dpr** and **Unit1.pas** (the **Unit1.dfm** file is saved automatically in the same folder where the Unit1.pas file is also saved). When saving, it is worth changing the name of the **Unit1.pas** module (unless you intend to leave this name forever: Renaming modules in an **MCK** project can be a bit tricky, so it's best to think about module names in advance). However, you should not change the name of the project for now, let it remain **Project1** (below I will explain why).
Another important point: keep all source files of the **MCK** project (at least the form modules and the project file itself) in one directory. If you put them in different folders, MCK may not be able to detect them and make the necessary modifications.

Now that the project has been saved, we begin to "convert" it to an **MCK project**. At this point, the **MirrorKOLPackageXX package** for the corresponding Delphi version should already be installed. Since I work with **MCK components** most of the time, after installing this package, I immediately go to the **Component | Configure Palette menu** and drag this set of components closer to the beginning, so as not to scroll through all the palette tabs in a row.

Conversion consists of four very simple steps. But they must be executed exactly, otherwise question # 1: "why is KOL / MCK not installed?" - is inevitable. Make your life easier, read the instructions very carefully !.

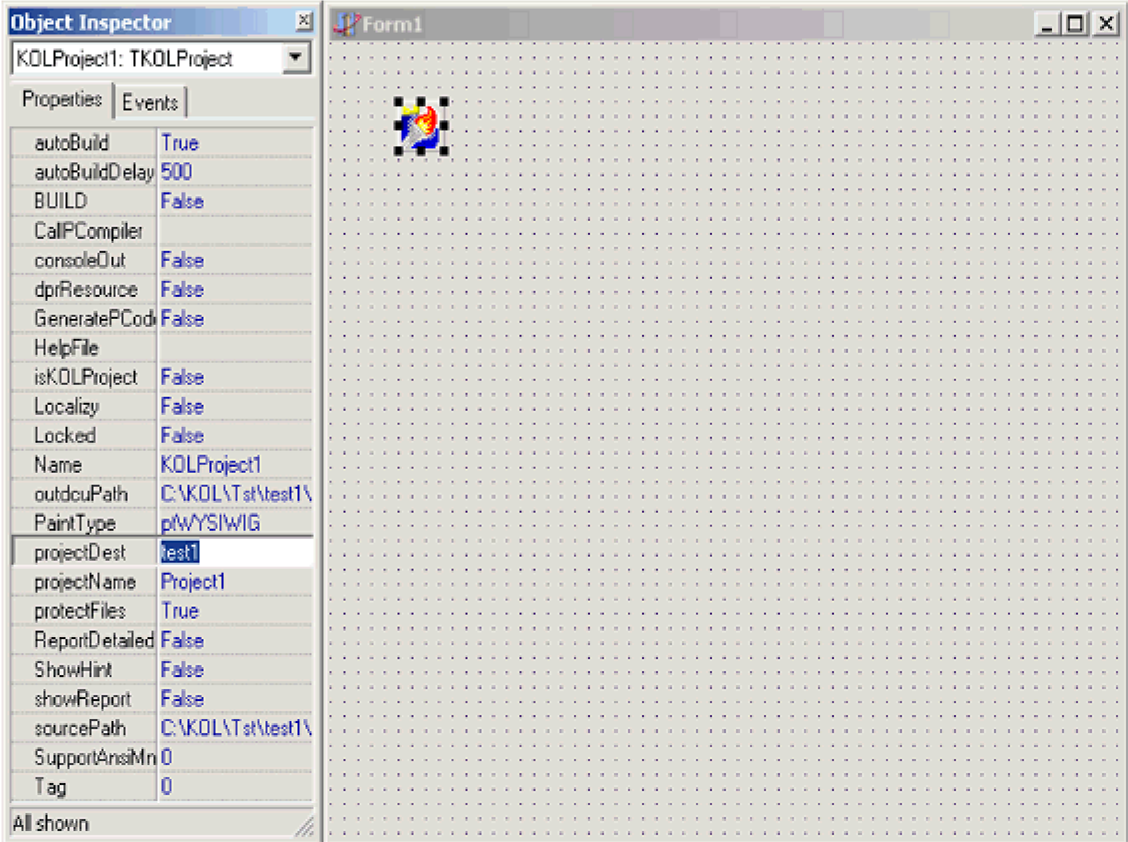
© 2024 Carl Peeraer

TKOLProject component

Firstly, you need to find it on the component ruler in the KOL tab and drop the **TKOLProject** component onto the form.

One such component must always be present in an **MCK project**. By the way, you shouldn't use Delphi's ability to open more than one project at the same time. As soon as two **TKOLProject** components are loaded simultaneously, they will start loudly complaining about the inadmissibility of the situation that has arisen.

When the **TKOLProject** component is created for the first time, or when opening an **MCK project**, it "comes to life" and starts checking regularly (by timer, the checking period is regulated by the **autoBuildDelay property** - in milliseconds) whether **MCK** components have requested code regeneration due to changing a property in any **MCK mirror** (including in the **TKOLProject** component itself). As soon as such a change is committed, by the next event from its timer, all forms of the project are searched for, and for those that have changed, a code regeneration is called, and, if necessary, a new version of the code for the project itself is generated.



TKOLProject component - projectDest property

Secondly, you must now select the **TKOLProject** component on the form, switch to the **Objects Inspector (F11 key)**, find and change the **projectDest property**. Enter the "real" name of the

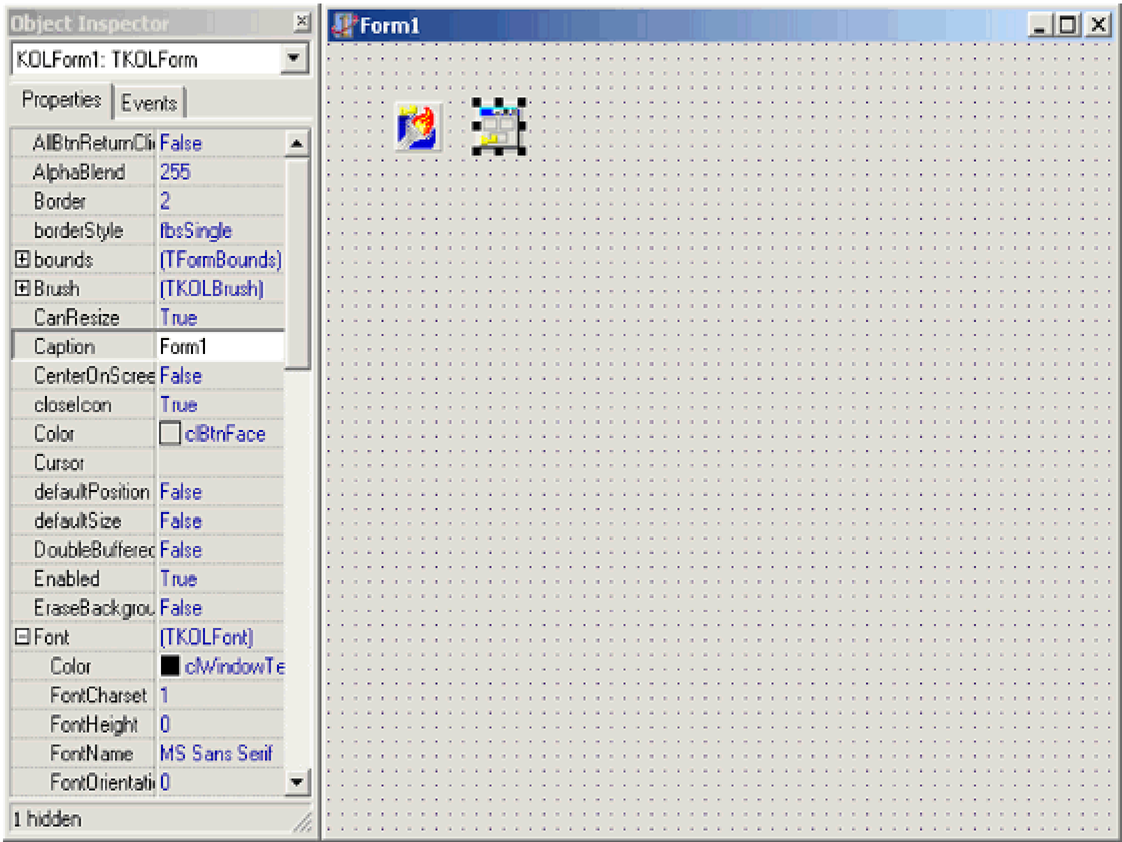
© 2024 Carl Peeraer

project in this field, other than Project1. For example, test1 is a perfectly valid name for a test project. (The project is ready for conversion, there is only a small step left to do).

TKOLForm component

Thirdly, find on the ruler, also in the KOL tab, the **TKOLForm** component, and also drop it on the form. Didn't you notice anything? In vain: you need to look carefully. If you had looked at the source code of the module before you dropped that component on the form, and compared it with what happened after that, you would have noticed that there were quite a lot of changes. But more on that later. There is one more step to take.

A number of changes are made to the code to ensure (through conditional compilation directives) that the compiler is invisible to the code that should not be compiled. At the same time (fortunately for **MCK**) the Delphi IDE itself "does not notice" the conditional compilation directives, which make undesirable code fragments invisible to the compiler. As a result, the form file, mirrored components, and form declaration text remain "visible" to the Object Inspector and the IDE, which continues to treat the project as a normal VCL project. In particular, even the Code Completion tools continue to work, and the navigation tools between methods and their declarations in a fictitious form class from now on continue to work. At least up to and including Delphi 2010.
In addition, the **{$ INCLUDE ...}** directive is added to link to the newly created file **<module_name> _1.inc**, which contains the procedure for initializing the form **New <form_type_name> (Form_Var, Aparent)**.



© 2024 Carl Peeraer

Save the MCK Project

Fourth, you need to **save the result** (**File | Save all**), and **close the project**. **Why close**? Because now in Delphi, the loaded is not the MCK project that we created as a result of the previous four steps, but a VCL project named Project1. We now need to open our project. "Test1" seems to be what we called it, right? Not? Well, you know better.
In principle, you can not close the project, but immediately select the File | Open Project menu item, and select the test1.dpr file for loading (well, or whatever you called it there). Moreover, at this moment I usually still pre-select the Project1.dpr file in the open dialog and press the <Delete> button - this file will no longer be needed (as the great Shakespeare said, "the Moor has done his job, the Moor can leave").

4.27.2 Form customization

Now that the newly created MCK project is open, you can try to **compile it and run it**. As usual, press the **green arrow on the Delphi toolbar**, or press the **F9 key**.

Did not work out? Probably, it is necessary to **register the path to the KOL.pas** file in the **project options**. In the Delphi menu, **select: Project | Options**, then on the **Directories / Conditionals tab**, find the Search Paths field, **enter C: \ KOL** here - well, or the path where you "installed" the KOL library.
The project should compile and run, and the form should appear. So far, there is nothing on it, of course. The form window can be moved, resized, minimized, and then closed.

Look in the project folder (to do this quickly, you can right-click on the **TKOLProject component** and select the Open Project Folder menu item - it is almost at the very top of the list), and select the assembled executable file. See what the size of the executable file is. You can see that the size is small enough compared to a single-shape project that is obtained in VCL (I now have 22 Kilobytes). You can make it even smaller right now if you have already **downloaded and unpacked the files to replace system modules in a folder** (for Delphi 7: https://www.artwerp.be/kol/sysdcu7.zip). Open the **project options** (Project | Options) and in the **Directories / Conditionals tab** in the **Search Paths field**, **add (separated by semicolons) the path to this folder**. Now the project needs to be rebuilt (**Project | Build**). Now see what happened to the size of the exe file.

Perhaps there is a little more that can be done to demonstrate the potential for size savings. If right now, select the **TKOLForm** component on the form, and change the values of the **defaultPosition** and **defaultSize** properties to true in the Object Inspector, add a couple more symbols to the list of conditional compilation symbols, separated by a semicolon after the **KOL_MCK** symbol available there: **SMALLEST_CODE** 38 and **NOT_USE_RICHEDIT** 42, then I get 13.5 Kilobytes. And what's great is that the resulting executable file can be compressed using a

© 2024 Carl Peeraer

compression utility such as **UPX** or **AsPack** up to 8.5 Kbytes, i.e. in about the same ratio (by a third) as a large VCL project.

Next, let's adjust the shape. If we just want to change its size or position, then to change these parameters of the form during development, we should proceed as usual, i.e. grab the mouse, and **resize the form**. Yes, the corresponding properties **defaultSize** and **defaultPosition** will have to be set back to false if you are not satisfied with the system defaults.

All other form properties should now be configured through the TKOLForm component... If you select a form, as you did in a VCL project, and try to change some property, like Caption, or something else, it will not affect the form in any way when it starts. So, here's rule # 1: to change the properties of a form in **MCK**, you need to select the **TKOLForm** component and change the required properties in it (the same Caption159).

Now you can customize the form however you want. **MCK** will generate code in the **Uni1_1.inc** module (this is where the code for initializing the form, automatically generated by the MCK components, is located). There is usually no need to load this module into the IDE editor.
And in any case, do not try to fix anything manually in this inc file.: such fixes will survive only until the next change in any of the properties of any **MCK component**, after which a new version of the form initialization code will be generated, and your work on fixing will disappear without a trace.

If you need to add some code of your own in the initialization of the form, you can do this in the event handler **OnFormCreate**, **OnBeforeCreateWindow**, or in **OnShow**. Note that if some code in the **OnShow** event handler needs to run once, you must ensure that **OnShow** is called the first time. For example, set up your own **boolean** variable **OnShow_Fired**, which should be set to true after the first execution of **OnShow**, and set the appropriate check.

```
if not OnShow_Fired then
begin
 // code that will work
 // only in the first call to OnShow
end;
OnShow_Fired: = true;
```

Be aware in the **OnFormCreate "event"** handler that window handles may still not exist for form visuals. In the **OnBeforeCreateWindow** "event" handler, window handles are not guaranteed to exist. And some objects may not be created as objects at this moment (i.e. pointers to them still contain nil, and an attempt to access their properties, methods, fields will lead to the crash of the application in this case). Those. you need to add the appropriate checks to your handlers code.

© 2024 Carl Peeraer

4.27.3 Coding

Perhaps it is necessary to clarify some of the features of writing code for **MCK**. Unlike "direct" programming in KOL, in MCK a form holder object is created for the form, produced directly from **TObj** 92 (you can do the same with manual KOL programming, but this is entirely at the discretion of the programmer). This object is "parallel" to the VCL form, but is not a visual object. The form itself (as an instance of an object of the PControl type) is represented in it by the Form field. That is, for example, if a **TForm1 form** was created in your VCL project, an object of the **TForm1 type** with the **PForm1 pointer type** is created in the **MCK project "at the same place"**. Those. here it is seen by the compiler. And all the components thrown onto the form during development, including the form itself (mirrored to the non-visual TKOLForm component lying on the form),

*Note. In fact, the commonplace banality of this description of the use of the word Form hides truly fantastic phenomena. For example, if you write the code proposed in the next paragraph in the **OnFormCreate** handler, then at the very moment when you type it from the keyboard, you may find that Code Completion, i.e. automatic code completion system, perfectly "sees" the Form variable. And you can go to the declaration of this field in the field structure of the "former VCL-form" (ctrl + click). The great thing is that this variable is "visible" at the same time as the code is being written, to the compiler when the code is compiled, and to the debugger when stepping through it. Probably should have applied for a patent ... but now it's too late.* *

The conclusion from the above is the following. The difference between writing code for VCL and for KOL + MCK is that in order to access a property, method or event of the form itself in KOL, you must use Self.Form instead of Self. For example, to change the title of a form in the OnFormCreate event handler, the code should be like this:

```
Form.Caption: = 'new caption';
```

4.28 Application graphic resources

Just like any PE * application in Windows32, the KOL application allows you to store the graphics resources necessary for its operation along with the executable file. These resources can be loaded by various methods, including implicit loading of resources by object methods, but ultimately they all boil down to calls to the appropriate API functions that create the required graphical object and return its descriptor.

From an application size perspective, the number and size of these resources can be important. Sometimes you can significantly reduce the size of resources if you use careful selection of the most suitable formats for storing graphics. First of all, notice the number of different colors required to represent a bitmap or icon graphic. Even if the picture uses non-standard colors that are not present in the standard system palette, but the number of these colors does not exceed 256, then it makes sense to use the 8 bits per pixel format. And if there are no more than 16 colors, then 4 bits per pixel is perfect. Even in the case of the transition from the full-color format 24 bits per pixel to the 8-bit format, for one image with a size of 100x100 pixels, a saving of 20,000 bytes will be obtained - 256x4 = 18976 bytes. Here I have subtracted the cost of storing

© 2024 Carl Peeraer

the 256-color palette itself at 4 bytes per color. I will also note that there is a presentation format (considered obsolete, but quite workable), in which not 4, but 3 bytes are spent per color in the palette. When using this format, you can save at least another 256 bytes (the image header in this format is also a few bytes shorter, so the savings will be slightly higher).

For large image sizes, where the total number of different colors used is already large enough to be limited to 256, consider using 16 bits per pixel images. An unpleasant feature of this format is that the R and B color channels (red and blue) discard the least significant 3 bits, and the G channel (green) discards the least significant 2 bits. In some cases, such "rounding" can slightly degrade the quality of the picture, especially if it contains elements of smooth gradient fill. But in general, in many cases, the deterioration in image quality is hardly noticeable to the eye.
Some novice programmers have the misconception that storing images in a compressed format, as opposed to storing them in a bitmap, can save application size. Yes, it can, but only if the total effect of compressing all such resources overrides the negative effect of adding appropriate decompression algorithms to the code. Usually, the presence of an unpacker for any of the most common formats GIF, JPG, PNG requires more than 30KB of additional code for each.
If the total effect of compressing images in resources by using one of these formats exceeds the size of the unpacker added to the program code, then the game is worth a certain number of candles. However, in this case, you should also take into account that if you plan to package the application using some external packer, then the effect of such packaging will be significantly reduced if some of the resources inside the application are already packed. Very often it makes sense in this case not to use any graphics packers at all, and store resources in the form of "flat" uncompressed "bitmaps", and use an external packer or a simple archive of the application file at the time of its distribution to end users.

There is another compression format that is sometimes quite useful (for example, in the case of images containing large areas of one color) that is often forgotten: it is RLE encoding. The TBitmap.LoadFromStreamEx method allows loading such images without any problems (thanks to V. Gavrik, who added this method to KOL). Most likely, you will have to load such a resource with your own code, referring to the above method, but this is no more difficult than using loaders of other formats with compression. You will, of course, have to find software that will RLE compress images before packaging them into resources.

4.29 Graphics Resources and MCK's

If you are using **MCK**, it will be useful to know that when using graphic resources that are automatically added to the application, **MCK** will automatically reduce the number of colors used, if it finds it possible. When generating resources containing images, the number of different colors used is counted, and the lowest possible representation in bits per pixel is used, in order to save the resulting image. Note also that the 16 bits per pixel format will be used automatically only when it does not require discarding the least significant bits in the R, G, B channels (see the previous paragraph). The maximum format used for graphic resources of 24 bits per pixel is used only as a last resort, when all other formats do not fit.

© 2024 Carl Peeraer

In the first versions of **MCK**, to build resources and generate *** .RES files**, the MCK add-in called an external resource compiler, which was BRCC, a Borland resource compiler that comes with the Delphi compiler and development environment. But this compiler still does not know how to work with graphic resources containing more than 256 colors, so in the end I refused to use the BRCC compiler. / Perhaps the limitation is due to the fact that the VCL itself is not able to load such resources, so there is no point in remaking the resource compiler for use with the VCL. But this is my guess, not necessarily true /.

Now **MCK** generates the resource file on its own by simply adding the correct resource header to the graphic file. The result is much faster and less problematic. If you do not use MCK, then to build resources you can try using other resource compilers, for example MS VC ++, or Resource Workshop, by the same Borland company. Although written for Windows 3.1, it works great in XP as well, although its interface is long overdue to send to the museum of antiquities. Or you can watch how MCK performs this task and write your own bitmap-to-res converter.

© 2024 Carl Peeraer

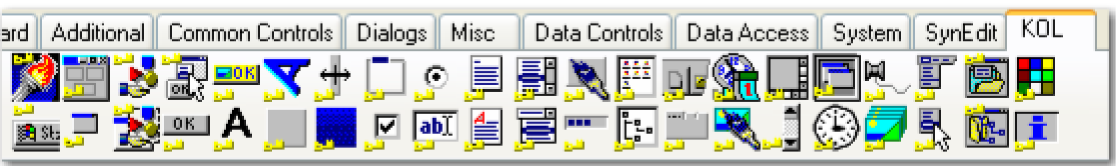
Window Objects

This chapter will be devoted to simple controls with a minimum of specific properties, methods and events

5 Window Objects

This chapter will be devoted to simple controls that contain (even this is not necessary) a single line of text, a minimum of specific properties, methods and events in the **TControl object type** 184 (in which they "live" together with many other kinds of visual objects, without the need to branch out into a separate object type). a type).

It should be noted right away that since the contents of almost all of these visual objects (text, picture, combination of text and picture) have easily defined sizes, then almost all of them can have the property of automatically resizing to the content (see the AutoSize method). Exceptions to this rule are panels (since their contents are, first of all, visual objects child of them).

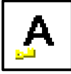


- Labels (label, label effect) 293
- Panel (Panel, Gradient Panel, Gradient Style) 294
- Groupbox 296
- Paintbox 296
- Imageshow 297
- Splitter 298
- Scrollbar 299
- Progressbar 300
- Scrollbox 300
- Buttons 301
- Switches (Checkbox, Radiobox) 304
- Visual objects with a list of items 304
- Text input fields (editbox, memo, richedit) 305
  - Text input field constructors (edit) 306
  - Specifics of using common properties (edit) 306
  - Input field options (edit) 307
  - General properties of input fields (edit) 308
  - Empowering: direct API access (edit) 309
  - Features of Rich Edit 309
  - Mirrored input field classes (edit) 314
- List of Strings (Listbox) 314
- Combobox 316
- General List (List View) 318
  - List Views 320
  - Column management 320
  - Working with items and selection 321
  - Adding and removing items 322
  - Element values and their change 322

© 2024 Carl Peeraer

  - Location of items 323
  - List view 324
  - Sorting and searching 324
- Tree View 325
  - Properties of the whole tree 327
  - Adding and removing nodes 327
  - Properties of parent nodes 328
  - Properties of child nodes 328
  - Node attributes: text, icons, states 328
  - Node geometry and drag 329
  - Editing text 329
- Tool Bar 330
  - General properties, methods, events 333
  - Setting up the ruler 334
  - Button properties 335
  - Some features of working with the toolbar 335
- Tab Control 336
- Frames (TKOLFrame) 339
- Data Module (TKOLDataModule) 340
- The Form 341
- "Alien" Panel 342
- MDI Interface 342
- DateTime Picker 344

5.1 Labels (label, label effect)



So one of the simplest objects of this kind is label. **Label constructor**:

NewLabel(Parent, s) 345 - returns a pointer to the created object of the **PControl** type, which in the constructor receives the visual and behavioral features necessary for the label, and sets the passed parameter as the text (Caption) of the label 346. Almost all constructors of visual elements on a form (and this one as well) have a Parent parameter of the **PControl** type, which indicates which window object is the parent of the object created by this constructor.

To a large extent, the label in KOL is similar to the TLabel class in the VCL, but there is an important "but": in KOL, the label is a window object. There is also a graphical label, but it will be discussed later, along with all the other graphical (windowless) controls. In addition to the above general properties, methods and events, the label actually has nothing more and there is nothing that could be called characteristic only of the label. If it differs from its other **TControl** neighbors in any way, it is its behavior. namely: the label cannot have the input focus, the tab order (TabStop, TabOrder), so the keyboard keypress events do not make sense for it.

© 2024 Carl Peeraer

There is an additional restriction introduced solely for the convenience of manipulating objects at the design stage. The label, like many "leaf" controls, cannot become a parent for other visual objects *. If any window controls could become the parents of any visual objects, then when dropping controls from the component ruler onto the form, you would have to aim very carefully, looking for a place for a new child object. Now, in many cases, it is enough to get somewhere inside the future parent. Of course, when writing code manually (even in MCK), no one bothers to create controls, the parent of which is a label or a button.

There are **two other flavors of window labels** in KOL. **Constructor**:

`NewWordWrapLabel(Parent, s)` 345 - creates a label with text wrapped by words when the text reaches the right border of the label. With the help of such a label, it is convenient to make multi-line explanatory inscriptions and messages on the form. There is a small peculiarity of using the **AutoSize** method for a label with word-by-word breaks: when automatically resizing it, it never changes its width, but only its height (this has nothing to do with alignment with the Align property, it is only about automatically adjusting the object's size to the content, in this case - for multi-line text).

Constructor:

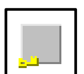
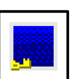
`NewLabelEffect(Parent, s, shadowdeep)` 345 - Creates a custom label with additional visual effects. The text in such a label can have a shadow that is drawn at a specified offset. The color of the shadow text uses the (optional) **Color2 property** of the **TControl** object. The most important feature of this label is that it is the only one capable of correctly displaying text in a font that has a non-zero rotation angle (**FontOrientation**) when rendering its content on its own. You can even animate the rotation of the text using this label (see the ADV demo application).
Note that the label must use a True Type font for the text to rotate. This applies not only to "labels with effects", but also drawing on the canvas with the changed **FontOrientation** property for the font.

Additional properties for controlling text label with effects in TControl:

ShadowDeep - shadow depth in pixels (can be negative or zero). Initially set as a parameter to the **NewLabelEffect** 347 constructor;

TKOLLabel and **TKOLLabelEffect** mirrors are available for Cue and Cue Effects for use in MCK projects. There is no separate mirror for a “word wrap” label, the same TKOLLabel is used, you just need to set the **wordWrap property** to true using the Object Inspector.

5.2 Panel (Panel, Gradient Panel, Gradient Style)

© 2024 Carl Peeraer

Another important and commonly used type of control is panel. Panel constructor:

NewPanel(Parent, edge) 345 - creates a panel of TControl type (by returning a pointer of PControl type to it - I will not dwell on this point anymore). The difference in the parameters is that the caption text string is not passed to this constructor. It's not that the panel can't have text. Maybe the same as the label. But here's the question: how often do you leave the default title (Panel1, Panel2, ...) for the panel in your Delphi project? Personally, in almost all cases, I immediately go to the Caption property in the Object Inspector and press the <Delete> button.

Thus, although the panel can have text, when designing the designer for it, I decided that usually this is, after all, an unnecessary parameter, and it is much more important when creating the panel to set the type of border for it (which is what is done). The panel can be flat, convex or depressed - which is specified by the second parameter of the constructor.

The panel has no other features. Unlike a label, it can (and this is what it is designed to do) parent other visual elements on a form. Including, when designing a form in MCK, the panel "accepts" the controls thrown onto it from the KOL toolbar as child controls.

Additionally, KOL has a special panel with a gradient fill effect for its content.

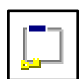
NewGradientPanel(Parent, Color1, Color2) 345 - creates such a panel with the default style gsVertical. This fill style can be further modified by modifying the GradientStyle property, or by using an alternative constructor:

NewGradientPanelEx(Parent, Color1, Color2, style, layout) 345
There are styles of vertical, horizontal, diagonal - left to right top to bottom, and left to right bottom to top, as well as rhombic, elliptical and rectangular fill. It is also possible to control the placement of the conditional center of the fill (the last parameter of the second constructor is layout).

The gradient panel is a full-fledged panel, it can contain arbitrary child visual objects. If you use transparency (but not for all controls this is possible), you can get amazing effects. Of course, using a gradient bar adds a few kilobytes to the size of the application, so the choice between beauty and size is yours.

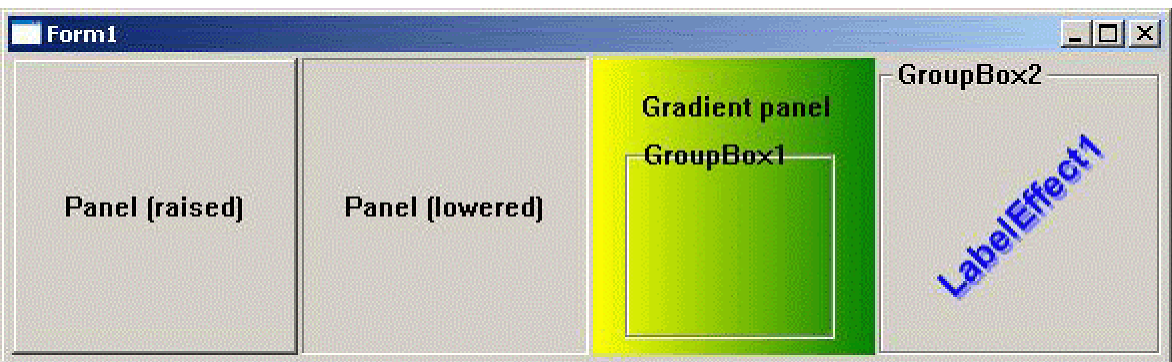
© 2024 Carl Peeraer

5.3 Groupbox



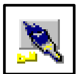
Another important window object that is intended to be used as a panel is the group box. In fact, in the Windows API, a group is a special kind of button. When I implemented this object in KOL, I was, first of all, interested in the fact that it provides automatic rendering of its very characteristic appearance, i.e. independently provided the image of the header, frame, and at the same time could be used exactly as a parent for child controls, like a panel.

Constructor:
`NewGroupBox(Parent, s)` 345. **Actually, the group has no other special properties.**



Screenshot notes: Since the gradient bar cannot display its title, it is labeled with a label placed on it with transparency added. In the figure, after its compression, vertical stripes became noticeable. In fact, such streaks are only noticeable when the desktop resolution is set to 64K colors or less.

5.4 Paintbox



Perhaps a paint box is an even simpler window object than a label or panel.

Constructor:
`NewPaintBox(Parent)` 345

It differs from other interface objects in that its text is not displayed in any way at runtime. This object is not intended to render anything on its own at all: the entire rectangle of its window (the size of which is the same as the client's size, that is, it does not even have a non-client part, or a border) must be drawn by your code in the **OnPaint** event (**OnEraseBkgnd** can also be used if needed). In general, if an **OnPaint** event handler is not assigned, the standard code will erase the rectangle (filled with a Color or Brush).

© 2024 Carl Peeraer

In both **KOL** and **MCK**, the paint box can be the parent of other controls. In fact, introducing a separate type of control for the paint box is redundant, since the panel may also have no border, and its image can also be painted in the **OnPaint** event. But it is more customary and more convenient to have a separate specialized object for drawing purposes.

In order to draw an image in the paint box control, it is enough to assign the **OnPaint** event and call the necessary methods of the picture object to paint on the canvas of our "box". This task is greatly simplified by a specially developed kind of control, which is called so - **image show** 297 ("showing a picture", or "show a picture", whatever you like).

In **MCK**, the mirror for the paint box control is **TKOLPaintBox**.

5.5 Imageshow

In order to draw an image in the paint box control, it is enough to assign the **OnPaint** event and call the necessary methods of the picture object to paint on the canvas of our "box". This task is greatly simplified by a specially developed kind of control, which is called so - image show ("showing a picture", or "show a picture", whatever you like).

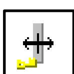
Constructor:
`NewImageShow(Parent, imagelist, i)` 345

Creates a "show" -control, which displays the **i-th picture** from the list of pictures **imagelist** 174. In order to remove the border surrounding it for the "show" control, it is enough to set the **HasBorder** property to false. Among other things, setting automatic sizing for this control will set the size of the control in accordance with the dimensions of the images.

In **MCK**, the mirror for the "show" control it is **TKOLImageShow**.

© 2024 Carl Peeraer

5.6 Splitter



Windows does not have a dedicated window that would deal with "splitting" adjacent elements. Nevertheless, such an element is widely used in applications when it is necessary to provide the user with the ability to dynamically change the width or height of one element due to the corresponding size of the adjacent visual element. Such an object is implemented simply as a regular panel with additional window message handlers that allow you to grab the object with the mouse and drag it to a new location (movement is allowed along one axis, only horizontally, or vertically). In this case, the "splitter" automatically resizes the shared visual elements.

Delimiter constructor:
`NewSplitter(Parent, i, j` 345

Creates an object of the PControl type with the separator functionality, assigning the values i and j to the MinSizePrev and MinSizeNext properties. By default, no division axis is specified. The splitter object "learns" whether to split two adjacent windows vertically or horizontally when it gets the Align property equal to caLeft, caRight (separation of horizontally adjacent objects) or caTop, caBottom (vertical separation).

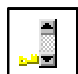
Properties and events:

MinSizePrev - the minimum size of the visual object "before" the separator. I put "before" in quotation marks, because the meaning of this definition depends on the alignment of the dividing window: for caLeft and caTop, "before" means to the left and above, and for caRight and caBottom - to the right and below;
MinSizeNext - the minimum size of the opposite of the two controls shared by the split object;
SecondControl - returns (and allows to set) the pointer of the visual object "after" the separator, the dimensions of which will be tightened when the location of the separator window is changed;
OnSplit - an event that is triggered every time when, in response to a signal about mouse movement (when dragging the separator), the object must make a decision about the admissibility of new coordinates (and, accordingly, new sizes of shared objects).
As you can see, this object is very simple. It is only intended to allow two (aligned) visuals to be resized by moving the border strip between them. If you need to have several such delimiters in a row (that is, in the form <window1> [separator] <window2> [separator] <window3>, then this object will not be able to function normally. In this case, make nested panels, and separate the parents, for example, according to the following scheme: <<> [] <>> [] <>.

MCK has a **TKOLSplitter** component that allows you to customize this type of window at design time.

© 2024 Carl Peeraer

5.7 Scrollbar



A tool window of this type is usually not required on its own. Almost all windows that are containers for lines of text, images, or other objects automatically provide scroll bars when needed. However, a few years after the creation of KOL, such a window object was added. It allows you to organize the scrolling of arbitrary elements without being part of any window object.

Constructor:
`NewScrollBar(Parent, side)` 345 - Creates a scroll element, giving it a direction - vertical or horizontal.

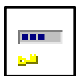
In addition to the general visual properties inherent in all window objects, the scroll element has a number of specific ones only for it. Namely:

SBMin - minimum scroll position (initial value 0);
SBMax - maximum scroll position (initial value 32767, but any integer greater than SBMin is allowed);
SBMinMax - intended for obtaining or changing the properties of SBMin and SBMax in one step, through the TPoint structure;
SBPosition - current scroll position, from SBMin to SBMax inclusive;
SBPageSize - page size. Used to scroll by page as an increment or decrement for SBPosition. In addition, if this value is not zero, then the system automatically calculates the size of the "slider" on the scroll bar, so that it, if possible, demonstrates how large one page of scrolling content is - compared to the entire content (this size cannot visually, however, be less than a certain minimum value determined by the system);
OnSBScroll - an event that is triggered when scrolling is performed. In the case of scrolling by dragging the slider on the ruler with the mouse, this event occurs regularly until the slider is released.

Mirror in **MCK**: **TKOLScrollBar**.

© 2024 Carl Peeraer

5.8 Progressbar



To show how many percent of the data has already been processed, during the execution of any lengthy operations, it is customary to use this element. It is called "progress".

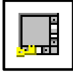
Constructors:
`NewProgressBar(Parent)` 346 - creates an object for the horizontal progress window, and returns a PControl pointer;
`NewProgressBarEx(Parent, options)` 346 - completely similar to the previous constructor, but allows you to set additional options: vertical direction, and solid fill when painting progress (by default, a set of "bricks" is used).

Properties, methods, events:

Progress - a number that defines the current "percentage" of execution. By default, the maximum value for this property is 100, so this is really a percentage. But the maximum value can be changed:
MaxProgress is the maximum value for the Progress property. To visualize the current progress of execution, a part of it is painted in the ruler window, proportional in area to the **Progress / MaxProgress ratio**;
ProgressColor - sets the color for shading (for the rest, the Color is used, as usual);
ProgressBkColor - the same as Color - the color for the window itself.

The **MCK** mirror **TKOLProgressBar** component chooses which constructor to add to the code to initialize the form, based on design-time options.

5.9 Scrollbox



Sometimes there is a need to be able to accommodate a very large number of visual elements on a form, so large that it will almost never be possible to see them all, even if the form is expanded to full screen. Sometimes it is also necessary to ensure the scrolling of a large work plane, such as a drawing box, the dimensions of which are large, and sometimes unknown in advance.

In the VCL, for the first time, the form itself has the ability to scroll through its content using standard scroll bars. But this feature is implemented so strangely that sometimes you just wonder when looking at applications developed in Delphi and running on another computer. For example, if the application itself was developed on a machine with completely different desktop

© 2024 Carl Peeraer

font size settings, then even on a small form its elements begin to not fit (although if they had a normal font, everything would fit perfectly, and there would still be space). And that's when the scroll bars appear. Moreover, the developer did not even order such an opportunity, just the automatic inclusion of scroll bars for the VCL form is built in by default, and on his machine everything fit perfectly without them. He doesn't even know

The KOL form, in principle, does not have any scroll bars. But, if you wish, you can use a specially designed version of the TControl object for this purpose: a container, or a scrolling box. Its constructors:

NewScrollBox(Parent, edgestyle, bars) 345 - creates a universal "scroll box" for scrolling some geometrically large object;
NewScrollBoxEx(Pafrent, edgestyle) 345 - creates a scroll box that automatically scrolls child visuals (if any).

In general, this visual element no longer has any other specific properties. Otherwise, it can be considered a panel "with edges extending beyond the horizon". In MCK, the mirror for this object

5.10 Buttons

What is an application without buttons? There are two main types of buttons in KOL.

Constructors:
NewButton(Parent, s) 344 - creates a regular button that cannot be changed in color (this is how Windows works: someone once decided that all buttons must have a standard mouse color, and since then it has been so, only not all programmers use standard buttons as a result in their applications).
NewBitBtn(Parent, s, options, layout, bmp, n) 344 - creates a "hand-drawn" button (something like TBitBtn in VCL, but windowed). This type of button has much more options and settings that allow you to set images for it (for several states: the button is not pressed, pressed, inaccessible, the button is by default, or under the mouse cursor), drawing options (flat, without borders, with fixation, with autorepeat, etc.). A significant drawback of such a button is that its appearance is poorly compatible with XP themes.

In fact, since Windows XP appeared and became widespread, the use of "self-drawn" buttons is strongly discouraged, as they can seriously spoil the appearance of the application, if not in the standard XP theme, then certainly in some additional. Instead of the **BitBtn button**, it is quite possible to use a standard button, placing the necessary child controls on it to display the button title, image and any other desired visual elements. This is possible because there is no restriction on how a button window can become parent to other windows. The limitation that is set on the **MCK** mirror of the **TKOLButton** can be removed by setting the **AcceptChildren property** to true. You just need to remember to set these child elements to the transparency of the mouse (MouseTransparent).

© 2024 Carl Peeraer

There are a number of methods and properties specific to buttons:

IsButton - returns true for all kinds of buttons (including the radio buttons discussed in the next chapter);
Click - a method that makes the button click and release as if it were clicked with the mouse. In fact, the same is done for any control when calling this method, but it is most used for a button, because it is for a button that such software clicks are visually observed by the user. For all other controls, it is almost always much easier to call the associated OnClick event handler from your code;
LikeSpeedButton - this property was already mentioned as a property that allows you to prevent the control from capturing focus, and for a button it makes it look like a TSpeedButton in VCL;
OnClick - this is the most important thing for a button (it is also present in the general description for all visual objects): because, why do we need a button at all, if we do not handle the events of pressing the button;

In addition to the general properties of the button, for bitbtn, i.e. for a drawn button, TControl has a whole set of additional properties, methods and events:

OnBitBtnDraw - a special drawing event, allows you not only to completely replace the drawing procedure, like OnPaint, but to complete its completion;
BitBtnDrawMnemonic - setting this property to true provides an image of an underline in the text on buttons on a mnemonic symbol that has a prefix '&' (the ampersand '&' itself is not displayed - this is the style that corresponds to the standard behavior of a regular button, with the difference that all drawing bitbtn-buttons are executed by the library code, not the system);
Flat - flat button (borders appear only when the mouse enters the button);
TextShiftX - horizontal displacement of the text on the button when it is pressed;
TextShiftY - vertical displacement of the text when the button is pressed;
BitBtnImgIdx - the index of the image in the image list associated with the button (if only the list of images is used, and not its own bitmap with several reliefs - glyph);
BitBtnImgList - a list of images for the **bitbtn** button (if used);
OnTestMouseOver - this event is used and generated only for the bitbtn-button (if set), so that the user, with his own code, can set the area in which the mouse is considered to fall on the button. This event allows you to form bitbtn-buttons of a completely arbitrary shape, in which the clicking occurs not on the entire rectangle of the button, but only in the active zone allocated by the user handler;
BitBtnInterval - interval of autorepeat of pressing the button, when the mouse is held down after pressing the button for some time (if 0, i.e. the property has not changed, autorepeat does not work).

© 2024 Carl Peeraer

Additional properties of all buttons:

DefaultBtn - should be set to true for the button to become the default button. Only one button on a form can be the default button. If there is such a button, pressing the <Enter> key causes this button to be "pressed" when the focus is on the form control (unless this control has the **IgnoreDefault** property set to true);

CancelBtn - cancel button. Similar to the **DefaultBtn** property, it allows you to define (the only one on a form) button as the "cancel" button that will be triggered when the <Escape> key is pressed on the keyboard when this form is active;

For a button, both the **DefaultBtn** and **CancelBtn** properties can be set to true at the same time, allowing you to "press" both the <Enter> key and the <Escape> key. But on the form there can be only one button with the **DefaultBtn** property, and only one button with the **CancelBtn** property.

In the case of a "hand-drawn" button (bitbtn), either a **single image** or an **image from an image list (imagelist)** 174 can be used. And in any of these cases, up to 5 "glyphs" can be provided - one for each of the states: **normal**, **pressed**, **disabled**, **focal**, and **highlighted**. In the case of a single image, the glyphs are arranged horizontally in the drawing, and the number of glyphs provided is based on the size of the drawing (assuming all images are square). In the case of a list of images, the glyphs themselves must be in consecutive index positions in the list, and the number of glyphs provided is specified separately. When directly creating a button with a list of images in Run-time, the number of glyphs is passed in the high word of the GlyphCount parameter (in this case, the low word is used to pass the starting index of the glyph set for the button in the list of images). When configuring such a button in MCK, there is a special design-time pseudo-property for this: **glyphCount**.

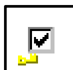
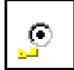
Note: Prior to version 2.42, the order of glyphs for states was exactly this: normal, pressed, disabled, focal and highlighted. Since version 2.42, the order has changed slightly, the "button is not available" state now has an index of 1. But this can be reverted to its original state by adding the conditional compilation symbol **BITBTN_DISABLEDGLYPH2**.

It is recommended that **BitBtn** buttons not be used in new applications, see the alternative method for creating arbitrarily complex buttons at the beginning of the paragraph.

In **MCK**, both buttons have their own mirrors: **TKOLButton** and **TKOLBitBtn**. When setting up a mirror for a regular button, you can find in the list of properties, including the image property. Yes, KOL implements the API-supported ability to display some image (icon) instead of text on a button. Although, for me, the text as an image on the button is cheaper in terms of the size of the code.

© 2024 Carl Peeraer

5.11 Switches (Checkbox, Radiobox)

Checkboxes and radio boxes in Windows are considered a kind of button. The TControl object has three main flavors of switches.

Their constructors:
`NewCheckBox(Parent, s)` 345 - creates a switch window object with two states, immediately setting a title for it;
`NewCheckBox3State(Parent, s)` 345 - creates a window switch with three states (checked, unchecked and in an undefined state - it is painted over in dark gray in the standard color scheme);
`NewRadioBox(Parent, s)` 345 - creates a radio switch, i.e. if the parent has multiple such radio switches (or radio buttons), then only one of them can be "checked" (checked). The essential difference from the TRadioBox component in the VCL lies precisely in the way of dividing radio buttons into groups, for which the RadioGroup property was responsible there.

Since all these radio buttons are buttons, they have all the general properties of controls, buttons, and, in addition, several properties specific to radio buttons are added:

Checked - reads or changes the state of the switch. To switch the radio button to the "on" state, use the SetRadioChecked method;
SetChecked(on) - "pass-through" method for initial initialization of the switch state when it is created;
SetRadioChecked - method exclusively for radio switch. Sets the radio button to checked = true, while disabling the currently enabled radio toggle on the parent window (if any). In this case, the OnClick event is triggered for both switches: for enabled and for disabled;
Check3 - the state of the three-position switch (created by the NewCheckBox3State constructor).

In **MCK**, these types of controls are represented by two mirrors: **TKOLCheckBox** and **TKOLRadioBox**. In this case, the **TKOLCheckBox** component has a design-time property **Auto3State**, which, if set, tells MCK that the code for the triple radio button should be generated in the form initializer.

5.12 Visual objects with a list of items

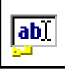
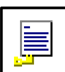
The next large group of objects are visual objects designed to represent a set of items (for example, lines of text) that can be manipulated through the corresponding General Items [] property. These can be windows for editing multi-line text, for viewing lists of texts, images, for presenting a line of buttons, etc.

© 2024 Carl Peeraer

The common set of properties, methods and events for all of these kinds of objects in **TControl** include:

Count - the number of items (lines in a multi-line input field - memo or rich edit, items in the list box, combo box and list view lists, top-level nodes in the tree view, pages in panels with tab controls, buttons in the tool bar) ;
Items[i] - access to the text of the element with index i (does not work for all list controls, for some of them - for example, list view - you need to use specialized properties);
ItemData[i] - access to an additional number or pointer associated with an element (not typical for all list controls);
IndexOf(s) - returns the index of elements with the specified text, or -1 if no such element was found;
SearchFor(s, i) - similar to the IndexOf function, finds an element containing the specified text, but the search starts from the element with index i;
Add(s) - adds a given string to the end of the "list" (just about the peculiarities of using for multi-line text input fields: the string must contain line termination characters # 13 # 10, otherwise, when the next lines are added, their text will be concatenated with the last line without moving to a new one string);
Insert(i, s) - inserts a new element with the specified text in the specified position of the "list";
Delete(i) - removes the element with index i;
Clear - clears the list of items (or text in the input field)
CurIndex - the current item in the list is also valid not for all list controls;
ItemSelected[i] - checks that the element is "selected" (typical for those list controls in which there is the concept of "selected element" or "set of selected elements");
OnMeasureItem - an event that allows you to set the dimensions (height) of an element in your handler. For more details, see below in the description of the controls for which this event works (list box, combo box, list view);
OnDrawItem - the event of drawing an item, allows you to set your own procedure for the image of a separate item in the list.

5.13 Text input fields (editbox, memo, richedit)

- Text input field constructors (edit) 306
- Specifics of using common properties (edit) 306
- Input field options (edit) 307
- General properties of input fields (edit) 308
- Empowering: direct API access (edit) 309
- Features of Rich Edit 309
- Mirrored input field classes (edit) 314

© 2024 Carl Peeraer

5.13.1 Text input field constructors (edit)

The first group of such window objects, which can be considered as consisting of a certain set of elements (lines of text), includes input fields.

They are created by constructors:

`NewEditBox(Parent, options)` 345 - Creates a **single-line** or **multi-line** input field object. A single-line input field always contains one element, and most of the above properties and methods are uncommon for it (although they work). But this field is actually a special case of a **multi-line text input field** (**memo**) created by the same constructor (but with the **eoMultiline** option in the options parameter);
`NewRichEdit(Parent, options)` 345 - creates an object for editing text with advanced formatting (rich edit, allows you to edit .rtf files);
`NewRichEdit1(Parent, options)` 345 - similar to the previous one, but not the latest version of the rich edit editor available on the system is used, but version 1 (which may be needed for compatibility purposes, for example).

Please note that when using rich edit controls in a project, the **NOT_USE_RICHEDIT** conditional compilation symbol should not be present in the project options.

5.13.2 Specifics of using common properties (edit)

The features of the above properties and methods for working with visual list objects when working with input fields are as follows:
editable multi-line text is conditionally divided into separate elements (lines). Namely, the line separator is a combination of # 13 # 10 characters (carriage return and line feed characters). This, in particular, means that when using word-by-word hyphenation (eoNoHScroll option), the line can visually be located in several lines, nevertheless, it continues to be a single "element" of the list of lines.

The **CurIndex property** does not make sense for input fields, since the selection in them is not made line by line, but character by character (a continuous piece of text can be selected, including one that starts on one line and ends on another);
for the same reason, the **ItemSelected [] property** has a slightly different meaning: it checks that the given row is in whole or in part in the selection.

To **add** or **insert** a whole line using the **Add** and **Insert** methods, the added line must be terminated with characters # 13 # 10 (otherwise, these characters themselves are not inserted into the text, and the line is combined with subsequent characters into one element).
Events **OnMeasureItem**, **OnDrawItem** for input fields are not applicable.

© 2024 Carl Peeraer

5.13.3 Input field options (edit)

Input fields have a number of properties, methods and events that are specific to working with text input fields. But first, it is still worth considering in more detail the options (the **Options property** - for mirrored MCK objects, or the options parameter, when manually calling the constructing functions) of such editing objects that are set when they are created:

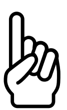
eoNoHScroll - prohibits the use of the horizontal scroll bar, as a result, too long lines of text are broken into separate lines along word boundaries;
eoNoVScroll - prohibits the use of the vertical scroll bar (as a result, the field does not allow entering more lines than can be displayed in the text field without using vertical scrolling);
Note: The previous two properties are ignored for a single-line input field that never has scroll bars.
eoLowerCase - all letters in the text are shown (and entered) in lower case;
eoMultiline - creates a multi-line text input field (including, it is possible to create a single-line input field for rich edit text, if this option is omitted);
eoNoHideSel - Prevents hiding of text highlighting with color when the object window is not in the focus of keyboard input;
eoOemConvert - allows you to display correctly the characters of the national OEM encoding (this is the character set that was used in DOS);
eoPassword - all characters in the input field when displayed are replaced by a substitute character assigned by the system by default or an additionally specified character. Used in non-professional password fields;
eoReadOnly - the field is used only to display the text placed in it from the program, the user cannot edit such text (but getting into the keyboard focus, if not prohibited, is still possible - to select fragments of text, and, for example, copy them to the clipboard) ;
eoUpperCase - all letters in the text are shown (and entered) in upper case;
eoWantReturn - a multi-line input field will, when the <Enter> key is pressed, feed the carriage, completing the input of the current line (in fact, the characters # 13 # 10 are inserted into the text);
eoWantTab - when the <Tab> key is pressed, the input field will insert a tab character (with code # 9) into the text; if this option is absent, this key is used to tab between the elements receiving the input focus;
eoNumber - a field for entering numbers (i.e. only numbers are accepted).

© 2024 Carl Peeraer

5.13.4 General properties of input fields (edit)

Now I will list the common methods, properties and events of all input fields (i.e., related mainly to objects for editing plain text - memo, and rich text - rich edit):

Text - (same as Caption). Yes, I already mentioned this property among the most common for all visual objects. But for the input field, it will not be superfluous to remind about the existence of such a property that provides access to all editable and displayed text at once, returning it on one line (and allowing you to change all the text by assigning a line to this property);
TextSize - the size of the text in bytes. For formatted text, returns the size of the text without regard to formatting. (See also the **RE_TextSize** property);
SelStart - the position of the beginning of the selection area (character index in the general array of text characters, starting from zero). If the selection is not empty, then this will be the index of the first selected character. With an empty selection, the **SelStart** property still has a value - as the current input position in the text (visualized by a blinking caret). And then it indicates the index of the character, before which the new characters entered from the keyboard will be typed;
SelLength - the length of the selected text fragment (I don't need to explain here why the text is selected in general, right?);
Selection - a string property representing the current selection (you can read it to get the selection, or assign another string to this property to replace the entire selection). For a formatted text input field, this property represents the selected text in unformatted form; to get a formatted selection, use the specialized **RE_Text** property;
ReplaceSelection(s, canundo) - allows you to replace the current selection with string s, and additionally indicate that this operation will go to the rollback stack (i.e. if the parameter **canundo** = true, then this operation can be canceled);
SelectAll - when called, makes all the text in the field selected;
DeleteLines(i1, i2) - deletes lines in the specified range;
Item2Pos(i) - for a given line index, returns the position of its first character in the text (0 is returned for a single-line input field);
Pos2Item(i) - for a given position in the text, the index of the line to which this character belongs is returned;
EditTabChar - "pass-through" method, which ensures the typing of a tab character in the text when the Tab key is pressed (including for fields for which the eoWantTab option was not specified during creation);
Ed_Transparent - allows you to make the input field of plain text (not formatted) almost completely "transparent". Can be used to achieve special visual effects (for example, if the parent of the input field is a gradient bar);
OnChange - this event is triggered when there is any change in the text.



Note that the **OnSelChange** event does not make sense for an input field and does not fire; if you need to track changes in the carriage position and text selection, you should process keyboard and mouse messages, as a result of which the specified parameters change. Moreover, since these changes occur after the event has been processed, in the handler itself, you should send a user message to yourself using **PostMsg**, and catch this message;

© 2024 Carl Peeraer

CanUndo - checks if the undo operation can be performed now;
EmptyUndoBuffer - clears the rollback stack (after which it becomes impossible to rollback the changes made, and the **CanUndo** method returns false);
Undo - rolls back one performed change (see also **RE_Redo**).

5.13.5 Empowering: direct API access (edit)

In fact, the API has a much broader set of functions for working with input fields, and you can do a lot of extra work using the Perform method on an edit object. For example, scrolling text down one line can be done with the following call:

```
MyEdit.Perform (EM_SCROLL, 1, 0);
```

And to move the carriage to the visible area of the field:

```
MyEdit.Perform (EM_SCROLLCARET, 0, 0);
```

5.13.6 Features of Rich Edit

And now about the additional properties and methods of the text input field with formatting (rich edit), which are much more than the properties and methods common to all edit fields.

And, first of all, I immediately draw your attention to the **RE_Font** property, namely, that this property should be used to change the font parameters instead of the usual Font property.

MaxTextSize - the maximum size of the text in the input field. The default is 32767 (which is the maximum possible for an unformatted text input field - memo). In order to allow entering editing of large-sized texts, this property should be changed (the maximum allowable value is 4 Gigabytes, without one byte);
RE_TextSize[units] - returns the current formatted text size in the specified units of measurement (in the units options it is even possible to specify whether to take into account the characters # 13 # 10 at the end of lines);
RE_TextSizePrecise - returns the exact size of formatted text in characters;
RE_CharFmtArea - sets the character formatting area (current selection, current word, or all text) that is used when formatting characters (**RE_CharFormat**, and many properties that control font styles and colors). defaults to raSelection, i.e. formatting is applied to the current selection in the text. If you change the value to raAll, then the formatting will apply to the entire content, and in the case of the value of raWord, the changes will affect only the word to which the caret is set (SelStart);
RE_CharFormat - the lowest level of access to character formatting properties. Allows you to read the current formatting (this returns the structure), change this formatting (by changing the structure fields), and assign a new value for the formatting settings to this property. It is better to use the corresponding properties below to change individual formatting settings:
RE_Font - font settings, when reading this property, the settings for the first character in the formatting area (**RE_CharFmtArea**) are returned, when changed, the new font is applied to the entire formatting area. Sometimes it is necessary to change only one of the formatting styles in the formatting area (for example, italic), without affecting all the others, for this you should use the properties **RE_FmtItalic**, **RE_FmtBold** and other similar properties below.

 © 2024 Carl Peeraer

Note when changing the font for the rich edit control, you should be aware that the font height for this object is set in twips, not in pixels, which is 1/20 of a point, where point is a logical unit equal to 1/72 inch on the display screen (see the font properties in the description of the **TGraphicTool** 151 object).

RE_FmtBold - style "bold" for a font in the formatting area, in order to find out if the returned value refers to all characters in the selection area or only to the first character, use the property: **RE_FmtBoldValid**;

And similar properties for other character styles:

RE_FmtItalic and **RE_FmtItalicValid** for italic font style;
RE_FmtStrikeout and **RE_FmtStrikeoutValid** for strikethrough style;
RE_FmtUnderline and **RE_FmtUnderlineValid** for the underlined style, in addition, there is an additional property for the underline:
RE_FmtUnderlineStyle - allows you to set the style of the underline (single, double, word by word, dots, dashed, wavy, mixed dash-dot, dash-dot-dot, thickened, ...);
RE_FmtProtected and **RE_FmtProtectedValid** - protection of a piece of text from being changed by the user;
RE_FmtHidden and **RE_FmtHiddenValid** - hiding a piece of text from the user;
RE_FmtLink and **RE_FmtLinkValid** - allows you to mark a part of the text as a link (URL - Universal Resolve Link, usually used as a link to Internet pages, and to highlight email addresses);
RE_FmtFontSize and **RE_FmtFontSizeValid** - font height in twips (see above);
RE_FmtFontColor and **RE_FmtFontColorValid** - the color of the symbols;
RE_FmtAutoColor and **RE_FmtAutoColorValid** - specifies that the default color for symbols is used;
RE_FmtBackColor and **RE_FmtBackColorValid** - background color;
RE_FmtAutoBackColor and **RE_FmtAutoBackColorValid** - determines that the default color for the background is automatically used for the formatting area;
RE_FmtFontOffset and **RE_FmtFontOffsetValid** - the offset of the font from the baseline down (negative values - up), as well as the font height, is set in twips;
RE_FmtFontCharset and **RE_FmtFontCharsetValid** - a set of font characters;
RE_FmtFontName and **RE_FmtFontNameValid** is the name of the font.

© 2024 Carl Peeraer

In addition to character formatting, rich edit also has paragraph (paragraph) formatting.

It applies (is set) to paragraphs that fall within the selection area (and is returned for the first paragraph in the selection area). Here the phrase "in the selection area" means even a partial hit of the paragraph in the selection area, including in the case when there is no selection - to the paragraph in which the input caret position is located.

RE_ParaFmt - similar to **RE_CharFormat**, provides low-level access to paragraph formatting parameters: the structure should be retrieved, modified and reassigned to this property to change the paragraph formatting parameters. Other paragraph formatting properties, when changed, affect only the corresponding formatting style, without affecting other styles;
RE_TextAlign - text alignment (left, right, center, or width). Justification is performed and is preserved in the text, but, unfortunately, it cannot be displayed by the rich edit window itself, i.e. to see that the text in the paragraph is indeed aligned along the edges, you can only save this text in an rtf file, and load it for viewing in a word processor like MS Word or Write. Similar to the character formatting properties, there is also a corresponding validation property: **RE_TextAlignValid**, which indicates that this formatting takes place for all paragraphs that fall into the selection area, and not only for the first paragraph;
RE_Numbering - sets the use of paragraph numbering, see also properties **RE_NumStyle** and **RE_NumStart**;
RE_NumStyle - sets the numbering style (no numbering, unnumbered list, Arabic numerals 0, 1, 2, ..., letters a, b, c, ...; letters A, B, C, Roman numerals i, ii, iii, iv , ..., and uppercase roman numerals I, II, III, IV, ...);
RE_NumStart - sets the initial number for numbering (for letter numbering, the number 1 corresponds to the letter A or a);
RE_NumBrackets - sets the separator between the numbering sign and the paragraph text (brackets on the right: 1), 2) ..., brackets on both sides (1), (2), ..., point on the right 1., 2., ..., and a regular space);
RE_NumTab - width of the field reserved for the number (if the field is too narrow, the number is not displayed);
And all these numbering properties can be checked with the RE_NumberingValid property to ensure that the numbering is the same for all paragraphs in the selected area;
RE_Level - nesting level (read-only);
RE_SpaceBefore and **RE_SpaceBeforeValid** - the space (vertically) before the paragraph;
RE_SpaceAfter and **RE_SpaceAfterValid** - the space after the paragraph;
RE_LineSpacing - interline skip within a paragraph;
RE_SpacingRule - rule (units of measurement?) for the **RE_LineSpacing** property, no other information;
RE_LineSpacingValid - checks that the properties **RE_LineSpacing** and **RE_SpacingRule** are the same for all paragraphs in the selection area;
RE_Indent and **RE_IndentValid** - left indent for text in a paragraph;
RE_StartIndent and **RE_StartIndentValid** - indentation for the first displayed line of text in a paragraph ("red line");
RE_RightIndent and **RE_RightIndentValid** - indent from the right edge of the sheet for the text in the paragraph;

© 2024 Carl Peeraer

RE_TabCount - the number of tab stops in the RE_Tabs array;
RE_Tabs[i] - tab stops for text;
RE_TabsValid - checks that the properties RE_TabCount and RE_Tabs are the same for all selected paragraphs;

This completes the list of formatting properties, although it is not complete yet. If you need table formatting, you can see the corresponding properties in the **TControl help** 203, or by looking at the source code in **KOL.pas**.

It would be better to pay more attention to other properties with the RE_ prefix:

RE_FmtStandard - calling this method attaches an additional handler for pressing the keyboard, which provides the ability to format text using control keys. For example: ctrl + B - turn bold on and off, ctrl + I - italic, ctrl + U - underlines, ctrl + O - strikethrough, ctrl + L - left alignment, ctrl + R - right align, ctrl + E - align center, ctrl + J - edge alignment, ctrl + N - numbering style selection, ctrl + '+' - font increase, ctrl + '-' - font decrease, etc .;
RE_AutoKeyboard - this property controls the automatic switching of the keyboard layout when the caret enters the text written in the corresponding language (moreover, if this property is initially enabled for rich edit windows in Windows9x, then it is disabled in Windows NT);
Note: There is also an additional design-time property **RE_AutoKeybdSet** for the **MCK mirror TKOLRichEdit**, which controls whether code should be generated to set the **RE_AutoKeyboard** property to ensure the same behavior of the **RE_AutoKeyboard** property on all operating systems. Or, on the contrary, you should not create such code, and then such behavior will be determined by the peculiarities of the OS version.
RE_OverwriteMode - turns on the "in place" recording mode, in which, from the usual "insert" mode, the characters typed on the keyboard are not inserted into the carriage position, but replace the characters after the carriage (by default, this mode is automatically enabled when you press the Insert key on keyboard when the rich edit element has input focus);
OnRE_InsOvrMode_Change - this event occurs when changing the mode from "Insert - Insert" to "Replace - Overwrite" and vice versa;
RE_DisableOverwriteChange - allows you to prohibit changing the above indicated mode (however, the **OnRE_InsOvrMode_Change** event still continues to fire on pressing the Insert key, if assigned);
RE_LoadFromStream(strm, i, fmt, selonly) - loads text in fmt format of length i characters from the strm stream, replacing the entire text or selection, depending on the selonly parameter;
RE_SaveToStream(strm, fmt, selonly) - saves all text or only the selection in the specified data stream;
RE_LoadFromFile(s, fmt, selonly) - loads text from a file (the whole file is loaded, so there is no length parameter, as opposed to the RE_LoadFromStream method);
RE_SaveToFile(s, fmt, selonly) - saves the text (all or only the selected portion of the text) from rich edit to a file;
OnProgress - this event fires regularly when saving or loading text from a rich edit window using the **RE_LoadFromStream**, **RE_LoadFromFile**, **RE_SaveToFile** and **RE_SaveToStream** methods;

© 2024 Carl Peeraer

RE_Text[fmt, selonly]- allows you to access the entire text or only the selected part of the text in a rich edit control as one line. The fmt parameter specifies how to receive a string when reading, and when writing, tells how to interpret the assigned string (plain text without formatting, text with formatting, and other types - see the description in the code or automatic help for a list of all different types of representation). If this method does not work, during debugging, you can use the property
RE_Error - contains the error code (returned by the OnProgress handler);
RE_Append(s, CanUndo) - adds the string s to the end of the text, allows, if the value CanUndo = true, add this change to the rollback stack, which allows you to later undo this change;
RE_InsertRTF(s)- inserts string s, assuming it is the internal representation of rtf-formatted text. This method can be useful for quick insertion of pre-prepared formatted fragments of text (for example, when generating rtf documents programmatically);
RE_HideSelection(b) - allows to hide or show selection in rich edit control;
RE_SearchText(s, case, word, fwd, i1, i2) - performs a search for the text specified by string s in the region from position i1 to i2 (must be rearranged to search "back"), using additional search parameters: case - take into account the case of letters when comparing; word - search only whole words; fwd - search forward. Note: already in the Windows XP operating system, the rich edit library version 5.0 is used by default, for which this method does not work if the project is compiled without the **UNICODE_CTRLS** option. In this case, you should use the **RE_WSearchText** method, which transfers the string in Unicode;
RE_WSearchText(s, case, word, fwd, i1, i2) - similar to the previous method, but accepts a Unicode string (cannot be used for this reason in Windows9x);
RE_AutoURLDetect - this property determines the need for automatic recognition of Internet addresses and e-mail addresses in the text;
RE_URL - the last URL "visited" by the mouse cursor. Can be parsed by the OnRE_OverURL and OnRE_URLClick event handlers;
OnRE_OverURL - an event that is triggered when the mouse cursor is over an automatically determined URL, i.e. the address of the WEB-page on the Internet, or above the email address. To get the URL itself, an event handler can read the RE_URL property;
OnRE_URLClick - an event that occurs when the mouse button is clicked while the mouse cursor is over a URL. The handler can also read the RE_URL value to get the clicked address;
RE_NoOLEDragDrop - this method prohibits the use of the built-in rich edit (and always available without calling this method) ability to use drag-n-drop to drag text fragments between windows (including between windows of different applications);
RE_BottomLess - this "end-to-end" method sets the style to "bottomless" for the control, which allows it to scroll down outside the text boundlessly;
RE_Transparent - this property allows you to make rich edit partially "transparent" (performance is not guaranteed for all cases!);
RE_Redo - allows you to return a "rollback" of a number of recent operations. Unlike the usual control for editing unformatted text, rich edit remembers all the operations performed on the stack, and allows you to roll them back using the Undo method (and not just the last operation), and it is also possible for it to perform rollbacks back.

Additionally, I note that the set of visual extensions contains a special extension KOLOLERE2 (by Alexander Shakhaylo), which provides the ability to work with rich edit interface extensions. In

© 2024 Carl Peeraer

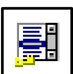
particular, it allows you to insert images, tables and other OLE-objects into the text of "rich" input fields.

5.13.7 Mirrored input field classes (edit)

There are three mirror components in the Mirror Classes Kit for editing controls:

- **TKOLEditBox** - one-line input field (created at runtime by the **NewEditBox** 345 constructor **without eoMultiline** in the options);
- **TKOLMemo** - multi-line unformatted text input field (**NewEditBox** 345 **with** the **eoMultiline** option);
- **TKOLRichEdit** - multi-line or one-line formatted text input field (**NewRichEdit** 345).

5.14 List of Strings (Listbox)



This kind of window object allows you to display a list of strings, which are handled exactly as with a list of elements. In particular, selection is performed exactly element by element (moreover, if multiple selection is allowed, then arbitrary row elements can be selected, not necessarily following one after another. Lines in such a list are never wrapped, and if they are not included in the width of the client part control windows are displayed truncated (if you do not prohibit horizontal scrolling, then by scrolling it is possible to read the entire line).

Constructor:
`NewListBox(Parent, options)` 346 - creates an object of type TControl, as usual, returning a pointer to it of type PControl. In the **options**, you can set the behavior and appearance for the created list:

loNoHideScroll - do not hide the selection when the window is not in focus;
loNoExtendSel - do not allow the selection of arbitrary elements, even with the loMultiSelect option (for example, by clicking the mouse while holding down <Ctrl>);
loMultiColumn - use several columns for display (to separate columns, the tabulation symbol # 9 is used in the text of elements);
loMultiSelect - multiple lines are allowed;
loNoIntegralHeight - when this option is enabled, any size of the window in height is allowed, and not only such that an integer number of lines can fit into the client part of the window;
loNoSel - does not allow line selection at all;
loSort - the list is always sorted;
loTabstops - uses tab stops to set the width of each of the columns in a multi-column list;
loNoStrings - the list is not intended for storing strings;
loNoData - a virtual list, in which the data is not stored or displayed by the window itself, but is provided by the OnDrawItem event handler. The loOwnerDrawFixed option must also be present in this case. The programmer must in his code provide storage and display of items for

© 2024 Carl Peeraer

the virtual list, the window in this case provides only scrolling, selection of items, keyboard and other functionality. Working with virtual lists is usually speed efficient, it is recommended to use this mechanism for large lists (over 1000 items);
loOwnerDrawFixed - the list is drawn by its own OnDrawItem handler, all lines in the list have the same height (if the OnMeasureItem event is present, it is called once to set the height of all lines). Note: the presence of the OnDrawItem event handler does not yet provide the ability to programmatically render list items: it is required that the loOwnerDrawFixed or loOwnerDrawVariable option be specified when creating an object;
loOwnerDrawVariable - similar to the previous one, but if there is an OnMeasureItem event, it is called for each row to determine its height.

Among the properties, methods and events common to all visual objects with lists, only a few specific features of properties should be noted:

SelStart - index of the first selected element (may differ from CurIndex - the current element in focus);
SelLength - the number of selected elements (including in the case when the selected elements are not always adjacent);
OnChange - the event is triggered when the set of selected items in the list changes. Same as OnSelChange for listbox;
OnSelChange - fires when the selection changes, like OnChange.
LVItemHeight- this property is common for list view (general list) and list box (simple list). It allows you to set the height of an element, which is accomplished by adding a window message handler WM_MEASUREITEM.

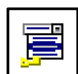
There is also a special method for use only in lists (also in combo boxes, see the next chapter):

AddDirList(s, attrs)- adds a list of files, subdirectories, from the specified path (the path must also contain a file mask, for example, 'C: \ Temp \ *. txt'). Directory names are appended in square brackets.

There is a **TKOLListBox mirror** for this control in **MCK**. Among other properties of this component, it should be noted that there is a design-time property Items, by editing which it is possible to prepare a list of items that will be added immediately after the creation of the object in the form initialization code.

© 2024 Carl Peeraer

5.15 Combobox



This window object combines two entities into one: a one-line text box, and the above list of strings. Very often this visual element is also called a "drop-down list", but this is wrong. Now this object is rarely used in the mode when the combo box is exactly the combo box, and the input field with the list is displayed at the same time (that is, the list seems to be "dropped out" forever and not hidden).

Most often, combo lists are still made drop-down (due to space saving on the form, probably). But the "simple" combo box mode, when the list is always displayed, still exists and can be used in applications.

It goes without saying that only one item can be selected in a combo box at a time. When you select (select) an item from the list, its text is copied to the input field. To add new elements, you need to call the Add and Insert methods in your code (the elements themselves are never automatically added to the list, including when they are entered into the input field, when it can be edited).

One small (but not insignificant) detail. An object of this type (when it is just a drop-down list) should not try to "align" (Align) so that its height tries to automatically adjust to the parent's height (or the height remaining on the parent window). That is, the styles caLeft, caRight, and caClient are undesirable. The problem might be that there will be a conflict between the system and the alignment code. In the worst case, the application will freeze trying to align an object that refuses to change its height, in the best case, the change in height will simply be rejected. The ways I know of to change the height of the input field for the combi-list is to change the font in it, or to use the OnMeasureItem event and the coOwnerDrawVariable style (see below).

There is a little more to know about the combo box. Unlike the vast majority of other window objects, a combo box does not allow you to dynamically change its parent (i.e. the window it is on). You can't just take and assign another object as Parent. The combined list receives a parent once, and will not be able to leave him for another until his death. (Generally speaking, changing the parent for a control dynamically is difficult to call a frequently requested feature, but you never know what cases happen).

© 2024 Carl Peeraer

Constructor:
`NewComboBox(Parent, Options)` 346; Combi-list options are largely correlated with simple list options, but there are also special ones:

coReadOnly - read only, refers to the input field, prohibits entering text here (an element can only be selected from the available list);
coNoHScroll - horizontal scrolling is prohibited, the option refers to the list of elements (if there are elements with too wide text, the text will be cut off, and it will never be possible to read it to the end if the user cannot dynamically increase the width of the list);
coAlwaysVScroll - always show the vertical scroll bar (even if the number of elements is small enough to show them all without using scrolling);
coLowerCase - the text both in the input field and in the list is displayed in lower case;
coNoIntegralHeight - similar to loNoIntegralHeight for a simple list object, refers to a list;
coOemConvert - for OEM-text (i.e. for text in DOS encoding), conversion to ANSI code is performed;
coSort - the list is always sorted;
coUpperCase - the text is displayed and entered in upper case;
coOwnerDrawFixed and **coOwnerDrawVariable** - similar to the corresponding options for a simple list, tell the system that the contents of the list will be drawn using an additionally assigned procedure (OnDrawItem event);
coSimple - the style of a "simple" list, in which the list is not drop-down, but constantly lies on the form directly below the input field, forming a single whole with it.

Properties and methods SelStart, **SelLength**, **CurIndex**, **Count**, **Items** [], **IndexOf**, **SearchFor**, **ItemSelected** [], **ItemData** [] - refer specifically to the list part of the combined control. The input field has a **Text** property (same as Caption). If you need to work with text selection in the input field, you should use the Perform method with the corresponding CB_XXXX window messages.

Properties common to the entire combo box:

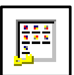
OnDropDown - this event is triggered before the list is dropped by pressing the F4 button, the Alt + <down arrow> key combination, or by clicking on the button with a triangle to the right of the input field, - for a drop-down combo box. In the handler of this event, it is possible, among other things, even to organize a change in the contents of the list - since it is not yet displayed on the screen, but you should not do that - for the reason that this may take some time, and it is better not to keep the user waiting for a long time, after every mouse click or keystroke;
OnCloseUp - this event is triggered when the drop-down list is closed - for any reason (an element is selected, the window has lost focus, the <Escape> key was pressed, etc.);
DroppedWidth - refers to the drop-down list, allows you to set a different width for it than for the entire window (in pixels);
OnSelChange - the same event as for a simple list, fixes the change of the selected item (with the difference that the combo box does not allow multiple selection, and only one item can be selected - the one that is currently displayed in the text input field);

© 2024 Carl Peeraer

OnChange - the event of changing the text in the input field, including triggered when the text has changed as a result of selecting another element in the combi-control list part;
AddDirList(s, attrs) - as well as for the list box-list, allows to add a list of files and subdirectories of the specified directory.

In **MCK**, the combo box is represented by the mirrored component **TKOLComboBox**. Similar to **TKOLListBox**, the Combobox mirror also has a design-time Items property that allows you to edit the list of strings to initialize the control.

5.16 General List (List View)



Starting with Windows 95 and NT 3.51, the so-called common controls ("common" window controls, or controls) have been added to the main set of windowing classes in Microsoft operating systems. Among them there is also a window for viewing lists of arbitrary elements, texts and images - list view. In the KOL library, all "common" controls are also implemented inside the TControl object type. Moreover, in many cases the polymorphism of methods, properties and events is preserved, so that in terms of the external interface for the programmer, the "general" controls practically do not differ from the original window objects (the so-called GUI windows, GUI - Graphic User Interface, or graphical user interface) ...

Shared List Constructor:

`NewListView(Parent, style, options, IL_normal, IL_small, IL_state);` 346

The constructor immediately sets the style (lvsIcon - icons, lvsSmallIcon - small icons, lvsList - list, lvsDetail - detailed, lvsDetailNoHeader - detailed without a header), lists of images (IL_normal - for the "icon display" style, IL_small - - to store the "state" icons displayed in a separate column in the lvsDetail and lvsDetailNoHeader styles; nil can be passed in place of any lists if the list is not used), as well as options:

lvoIconLeft - in the lvsIcon, lvsSmallIcon modes, place the icon to the left of the text (and not above the text, as by default);
lvoAutoArrange - automatic ordering of items in the lvsIcon and lvsSmallIcon view modes;
lvoButton - icons are displayed as buttons (for lvsIcon view mode);
lvoEditLabel - editing of the text of labels is allowed (the first column of the element);
lvoNoLabelWrap - the text is always displayed in one line (for the lvsIcon view mode);
lvoNoScroll - no scrolling in the window;
lvoNoSortHeader - do not try to sort items when you click on the column header button;
lvoHideSel - hide selection when the window is not in focus;
lvoMultiselect - allows multiple selection;
lvoSortAscending - Sort Ascending;

© 2024 Carl Peeraer

lvoSortDescending - sorting in descending order (if neither ascending nor descending sort is specified, then automatic sorting is not performed);
lvoGridLines - a grid of rulers between columns and rows;
lvoSubItemImages - columns can contain their own icons;
lvoCheckBoxes - system switches are used as images;
lvoTrackSelect - tracking the mouse cursor, and additional visual effects when the cursor crosses the elements;
lvoHeaderDragDrop - it is allowed to grab and drag column headers (lvsDetail mode), changing the display order columns;
lvoRowSelect - the entire line is selected, with all columns;
lvoOneClickActivate - single mouse click activates the element;
lvoTwoClickActivate - double click of the mouse activates the element;
lvoFlatsb - flat scroll bars;
lvoRegional - a special "transparency" mode, in which all client space, except for the elements themselves and their icons, is excluded from the window region;
lvoInfoTip - automatically create and display a window, which reflects the entire text of the column under the mouse cursor, if this text is not fully visible in the column itself;
lvoUnderlineHot - underline active elements (under the mouse cursor);
lvoMultiWorkares - use multiple working areas in the window (for viewing and automatic ordering in lvsIcon mode);
lvoOwnerData - the list is virtual, i.e. initially does not store any data itself, but in the **OnLVData** event handler receives their custom code;
lvoOwnerDrawFixed - a list of elements of the same height, displayed by the custom **OnDrawItem** handler (this style should not be used if the **OnLVCustomDraw** event is used).

At runtime, the view style can be changed (or obtained) by the **LVStyle** property. Moreover, it is possible to change any options of the general list dynamically using the properties of **LVOptions**.

Regarding the lvsDetail and lvsDetailNoHeader viewing styles, it should be noted that the text and images of list items in these modes are displayed in columns. But you need to create columns with your own code (or use the column editor at the development stage). If this is not done, the list box will remain empty, even if there are items in it!

For a general list, including those with multiple selection, including for the case when the selected elements are not adjacent, the **SelLength** property continues to work - it returns the number of selected elements. But in code it is better to use the **LVSelCount** property.

As usual, for list controls, the Count property also works (there is a synonym for it - the LVCount property), but for a virtual list this property can be set by telling the object window how many elements there are (shown) in the list.

Note. The effect of the "erroneous" appearance of empty (inaccessible to the user) lines before the very first line of the virtual list is known, if at the moment of changing the number of elements the scroll bar was not in the top position

© 2024 Carl Peeraer

(lvsDetail and lvsDetailNoHeader modes). A completely similar effect can be obtained for the virtual list TListView and in the VCL. You can avoid the appearance of such empty lines if, before changing the Count property, you put the object in lvsList view mode (for example), set the Count value to 0, and then return the image to lvsDetail or lvsDetailNoHeader mode; this problem is solved in a similar way for VCL applications. At least in my applications, this was the way to fix this failure.

The **OnMeasureItem** event (see the **Set_LVItemHeight** method) can be used if the lvoOwnerDrawFixed style is present in order to programmatically set the height of an item if the system's default height is not satisfactory. Personally, I often use another method: assign an object a list of images (in accordance with the view mode - a list for large or small icons, depending on the view modes used). The size (height) of the icon in such a list of images also uniquely determines the height of the elements, as long as it is larger than the height of the font used (if the font is larger, then the height of the element is set by the system so that the text fits completely in height). If the list of images itself is not used, it is not necessary to fill it with anything. An empty list is enough to set the required element height.

As with other list controls, the Clear method continues to work (perhaps the most polymorphic method for all kinds of window objects).

The general list is characterized by the property **SetUnicode(b)**- puts the object window into the mode of processing Unicode strings. To provide the ability to work with Unicode strings in list items, you must also add the conditional compilation symbol **UNICODE_CTRLS** * to the project options;

5.16.1 List Views

Additional properties of common controls, which can be the same as the general list view, are inherent in the tree view, tool bar, tab control, are:

ImageListSmall - allows you to set a "list of icons" object for the general list (for the lvsDetail, lvsDetailNoHeader, lvsList and lvsIconSmall view modes). Among other things, this property allows you to change this object dynamically at runtime.
ImageListNormal - for the general list, sets the list of icons for the lvsIcon mode;
ImageListState - a list of images for the states of elements. An icon from this list is displayed in a separate column (space for the column is reserved if there is a non-empty ImageListState assigned);

Next, let's look at some of the special properties of the general list.

5.16.2 Column management

Column management (list view in lvsDetail, lvsDetailNoHeader display modes)

LVColCount - the number of columns;

© 2024 Carl Peeraer

LVColAdd(s, textalign, i)- adds a column, setting the text alignment (left, right, center) and width for it. For automatic width control, pass a negative number as the width;
LVColAddW(s, textalign, i) - similar to the previous method, but working with a Unicode string;
LVColInsert(i, s, textalign, i1) and **LVColInsertW (i, s, textalign, i1)** - inserts a column at position i in the list of columns;
LVColDelete(i) - deletes the column with index i;
LVColWidth[i] - column width property;
LVColText[i] and **LVColTextW [i]** - column heading;
LVColAlign[i] - text alignment in the column;
LVColImage[i]- image (pictogram) for the column heading. If a number (greater than or equal to zero) is assigned, then the icon with that index from ImageListSmall is used;
LVColOrder[i]- the order of the columns does not have to coincide with the visual order of their display in the general list. This property sets the visual order of the column;
OnColumnClick - an event that is triggered when the mouse is clicked on the column header (lvsDetail view mode);
To analyze which mouse button was pressed in the **OnColumnClick** event on the header of the general list, you can use the property:
RightClick - returns true if the right mouse button was pressed.

5.16.3 Working with items and selection

LVCurItem - should be used to determine the index of the "current" element. I have put in quotation marks the word "current", since with this concept in relation to the general list, different interpretations may appear. The fact is that to provide an opportunity to work with the general list not only with the help of the mouse (but also the keyboard), the concepts of "selected items" and "item in focus" are different for it. Just as in the VCL, I decided to use the word "current" to mean the "first selected" item in the list. Probably because when the user asks for any actions to be performed with respect to individual list items, it is most often the "selected" items that are meant. The element in focus is primarily used to to be able to move between list items using the arrows on the keyboard, and highlight the desired items by pressing the appropriate keys. (However, there is a special property to get and work with it, see **LVFocusItem** below).
If the selection contains no elements, **LVCurItemreturns** the number -1. The same value should be assigned to this property to deselect and bring the list into a state in which there are no selected items in the list. Assigning a non-negative value for a list with multiple selection of elements adds an element with such an index to the set of selected elements, for lists in which only one selected element is allowed, this one element is selected, and then the selection is unselected;
LVFocusItem - the index of the element in the keyboard focus;
LVNextItem(i, attrs) - returns the index of the next element after i, which has the necessary attributes (they can set the search direction, as well as a combination of attributes **LVNI_CUT** - marked for cutting, **LVNI_DROPHILIGHTED** - highlighted for "dropping" objects dragged by the mouse on it, **LVNI_FOCUSED** - is in the input focus, **LVNI_SELECTED** - dedicated;
LVNextSelected(i) - finds the next selected (after index i, where i can be set to -1 to start the search from the beginning of the list);
LVSelectAll - a method for selecting all items in the list;

© 2024 Carl Peeraer

LVSelCount - returns the number of selected items in the list (the general property SelLength does the same);
OnLVStateChange - an event that is triggered when the state of the elements changes (the element is selected, in focus, etc.). The event handler is called for each element in which the state changes, or when the state of several elements changes simultaneously - once for the entire group of elements. In order, for example, to perform some actions only when a certain element is selected, the condition should be checked in the handler:

```
if (OldState and lvisSelect = 0) and (NewState and lvisSelect <> 0) then ...
```

5.16.4 Adding and removing items

LVAdd(s, ii, state, sii, oii, data) - "universal" addition of an item (it is recommended to use LVItemAdd, see below);
LVInsert(i, s, ii, state, sii, oii, data) - similar to LVAdd, inserts an item at the position with index i (I also recommend using the LVItemInsert method that appeared later);
LVItemAdd(s) and **LVItemAddW (s)** - adds an element, setting only the text for it (other properties of the element may well be set by the properties intended for this);
LVItemInsert(i, s) and **LVItemInsertW (i, s)** - inserts an element at the specified position;
LVDelete(i) - removes the element at index i;
OnDeleteLVItem (synonym for **OnLVDelete**) - this event is triggered for each deleted item (can be used to release the resources associated with the item through the **LVItemData** property, for example);
OnDeleteAllLVItems - is called before deleting all elements of the list at once. If, after returning from the handler for this event, the OnDeleteLVItem event handler remains assigned, then it will also be called for each deleted item;

5.16.5 Element values and their change

LVSetItem(i, j, s, ii, state, sii, oii, data)- allows you to assign specified attributes (text, icon, state, etc.) to an element or any of its subelements (columns) - in one call. If the index of the subelement (column) is 0, then the element itself is meant, and from index 1 the columns begin to be numbered (and for them the data parameter is ignored);
LVItemState[i] - "state" of the element (combination of flags lvisBlend - partial coloring of the icon, lvisHighlight - bright selection of the element, lvisFocus - focus frame around the element, lvisSelect - the element is selected). If, during assignment, -1 is used as an index for a list with multiple selection, then the state will change for all elements (for a list that does not allow multiple selection, index -1 refers to the last element in the list);
LVItemIndent[i] - indent from the left edge (for lvsDetail and lvsDetailNoHeader view modes). One corresponds to an indentation with a width equal to the width of the thumbnail in the ImageListSmall;
LVItemImageIndex[i] - index of the thumbnail for the main image in the element (taken from the ImageListNormal list of images for the lvsIcon view mode, and in all other modes - from the ImageListSmall list);
LVItemStateImgIdx[i]- the index of the image in the field for the "state" icon (should not be confused with the state of the element itself). The column with the status icon is displayed in the

© 2024 Carl Peeraer

lvsList, lvsDetail, lvsDetailNoHeader view modes. For state icons, a specially designed list of images ImageListState is used;
LVItemOverlayImgIdx[i] - overprint index for the image of the main icon of the element. This property can take values from 0 to 15, and the value 0 corresponds to the absence of overlays, and for values from 1 to 15, the corresponding "overlays" (Overlay) from the ImageListSmall images are used;
LVItemData[i] - associates a number (or pointer) with an element;
LVItems[i, j] and **LVItemsW [i, j]** - properties for accessing the text of items and subelements (columns). Column j with index 0 is the element itself, the indices of other columns start at one;
LVEditItemLabel(i)- initiates editing of the text of the specified element. Only the element itself (column 0) can be edited in the list view-control by its own means, but not sub-elements. Of course, lvoEditLabel must be present in the control options for this call to work (in this case, the user usually has the opportunity to start editing the text of any element using standard Windows tools: by pressing the F2 key, or by clicking the selected element with the mouse);
OnEndEditLVItem - this event is triggered when the user has finished editing the text of an item, for any reason (editing canceled or completed), and regardless of how the editing was started - by the user or programmatically, by the LVEditItemLabel method. The handler receives a new text as a parameter, and has the ability to substitute any other text for it (including returning the previous value);
OnLVData and **OnLVDataW** is a special event for the virtual list (with the lvoOwnerData option). The handler for this event is called every time when, when drawing a list box, the system needs to get text and images for display (virtual lists of topics differ from the usual ones that the program stores text and images, and as a result, it becomes possible to quickly work with huge lists of data) ;

5.16.6 Location of items

LVItemRect[i, part]- returns the rectangle occupied by the element (or its part specified by the part parameter) in the window. If the element is not currently visible in the window, then a rectangle with all coordinates equal to zero is returned;
LVSubItemRect[i, j]- returns the rectangle in the window occupied by a specific column of the element (for the lvsDetail and lvsDetailNoHeader modes). If the element itself is visible in the window, but its column j is not fully visible, then a rectangle with side borders extending beyond the left and / or right edges of the window will be returned;
LVItemPos[i] - returns the position of the element in the window (for the lvsIcon and lvsSmallIcon modes, this property also allows you to change the position);
LVItemAtPos(X, Y) and **LVItemAtPosEx (X, Y, where)** - return the index of the item located in the given coordinates in the window (or -1 if there are no items at this position). In the LVItemAtPosEx method, the where parameter returns exactly what part of the item ended up at the point (X, Y): icon, text of the item itself, status icon, another column;
LVTopItem - index of the element displayed in the first line of the list (modes lvsDetail, lvsDetailNoHeader, lvsList);
LVPerPage - the number of elements that fit into one page (lvsDetail, lvsDetailNoHeader modes);

© 2024 Carl Peeraer

LVMakeVisible(i, partialOK)- scrolls the window so that the element with index i is in the scope. If the partialOK parameter is true, and the specified element is already partially visible in the window, this call does not change anything;

5.16.7 List view

Set_LVItemHeight(i) - allows you to set the height of items (this method must be called before creating a window for an object so that the system can call the OnMeasureItem event);
SetLVItemHeight(i) - similar to the previous one, but the "through" method;
LVItemHeight - a property that allows you not only to set the height of the item (using Set_LVItemHeight), but later "remember" what height was set for the items;
LVTextColor - sets the color for the text font (should be used instead of changing Font.Color);
LVTextBkColor - sets the color for filling the background of the text in the elements;
OnDrawItem- called for each list item to display it (when lvoOwnerDrawFixed is present in the options). The handler should render the entire content of the element, including subelements - in the lvsDetail and lvsDetailNoHeader view modes;
OnLVCustomDraw - called to perform more detailed custom drawing of elements and / or subelements (as well as the title, and parts of the client area that does not contain elements). This handler will only work if the options lack the lvsOwnerDrawFixed style. Each time the window is drawn, this handler is called repeatedly, for the entire list, for the entire element, and (possibly) for each sub-element of each element, and at each of the drawing stages: prepaint (preparation for drawing), preerase (preparation for erasing the background), erase (erasing the background), paint (drawing the element), posterase (after erasing the background), postpaint (after painting the element). On each of these calls, the handler must return some flags that tell the system when else to call this handler while drawing continues. On the one hand, it is the most flexible and powerful tool for performing any custom control of the drawing process in the list box, on the other hand, everything is so complicated that it is rather difficult to understand all the twists and turns at once. The set of demo projects contains a special project (DemoLVCustomDraw) with a demonstration of the code of this handler. When writing your own handler, you can take it as a basis.

5.16.8 Sorting and searching

LVSort - starts sorting items (this method works if Microsoft Internet Explorer 5.0 or higher is installed in the operating system, see also **LVSortData**);
LVSortData - starts sorting items. Unlike the LVSort method, it works on all Windows systems starting from Windows 95, but the OnCompareLVItems event handler does not receive the indexes of the compared items, but the **LVItemData** field of these items;
OnCompareLVItems - an event for comparing two items during sorting (LVSort and LVSortData). See the LVSortData description for details on accepted parameters;
LVSortColumn(i)- performs sorting by column. Works for Windows 98 and 2000 (and above, of course), and provides automatic sorting of strings in the order of the lexicographic order of the text in a given column;
LVIndexOf(s) and **LVIndexOfW (s)** - returns the index of the first element with the specified value of the label (element text), or -1 if no such element was found;

© 2024 Carl Peeraer

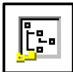
LVSearchFor(s, i, partial) and **LVSearchForW (s, i, partial)** - similar to the previous methods, it searches for an element with a specified string in the label, but allows you to specify after which element to start the search, and whether to compare strings in full, or only partially, by the first characters in a given search pattern;

In the MCK package for the general list, the **TKOLListView** mirror has a special editor for customizing the list of columns at the design stage (called, for example, by double-clicking on the list view rectangle on the form). But there is no editor for adding elements, you should add elements only with your own code at runtime.

If the columns for the list view object are created during the development of the form, then by default MCK provides an additional service: for each column, a symbolic constant with the name of the column is created, storing its index. It is convenient to use these constants in the code when referring to columns instead of directly specifying numeric indices. If you change the composition or order of the columns, in this case, you do not have to change the entire code.

However, if the column names are left as they are obtained by default (Col1, Col2, etc.), then if there are several objects of common lists on the form, there will certainly be a name conflict due to the redefinition of constants. To avoid this, you should either give your columns more meaningful names, or turn off the design-time generateConstants property (which makes sense if constants are not used anyway).

5.17 Tree View



Although the window object for the visual presentation of tree-like data structures (tree view) is somewhat apart from other list views, I nevertheless decided to place it in the place that it usually occupies on the component bar - after the general list view.
I note right away that the performance of the system tree view when working with a large number of elements leaves much to be desired. In addition, there is a system limit on the maximum number of nodes in a tree (65536). As more nodes are added, the window cannot even show the scroll bars, and tree viewing becomes problematic.

© 2024 Carl Peeraer

There are a number of recommendations (in terms of speed optimization) for programmers who use this visual element in their interfaces. Namely:

- avoid loading the whole tree. Nodes should be loaded as needed, when plowing up their parents;
- when constructing tree-organized data, one should prevent each individual node from containing too many slave nodes. Unfortunately, this very recommendation is rarely enforceable: the data is usually organized not by the programmer, but by the user. Example: file structure on disk.
- do not use this object for viewing trees, if you need a really high speed of work (or the total number of nodes cannot be limited to 65536). It is quite possible to display data in the form of a tree using the same general list from the previous chapter. This work can be done especially efficiently if you use a shared list in virtual list mode. At the same time, node data can be stored in memory using a simple TTree object (described above, in the section on simple, non-visual objects).

Nevertheless, a specially designed object for visual work with trees works well if the first two of these requirements are met, or in the case of small trees. Constructor:

NewTreeView(Parent, Options, IL_Normal, IL_State); 346

The following set of flags can be specified in the options:

tvoNoLines - do not show lines connecting nodes in the tree;
tvoLinesRoot - do not show lines for top-level nodes;
tvoNoButtons - do not show the buttons ("+" and "-") used to expand and collapse nodes;
tvoEditLabels - allows you to edit the text of nodes "in place" (F2 key or one more left-click when the mouse cursor is positioned on the text of the node;
tvoHideSel - hide the selection of the current node when the tree is not in the focus of keyboard input;
tvoDragDrop - automatically start the operation of "dragging" nodes with the mouse (by pressing the mouse button on the node and moving the cursor while the mouse is held down);
tvoNoTooltips - automatic display of the full text of nodes in a pop-up window when hovering over them with the mouse, if the text of the node does not completely fit into the client part of the window;
tvoCheckBoxes - system switches are automatically used as status icons for nodes;
tvoTrackSelect - visual tracking of mouse movement over nodes in the tree;
tvoSingleExpand - to expand only one node in the tree (all other nodes automatically collapse when another node is expanded, unless you clicked the mouse while pressing the Control key;
tvoInfoTip - sends a request to the object to get the text to show in the pop-up window for each node in the tree;
tvoFullRowSelect - full line selection;
tvoNoScroll - lack of scrolling in the window;
tvoNonEvenHeight - odd height for elements (by default, the line height must be an even number).

© 2024 Carl Peeraer

The main way to identify nodes in a tree is not their index, but node descriptors, i.e. integers that are one-to-one assignments to elements when they are added. Unlike the TTreeView component in the VCL, the tree view object in KOL does not create an in-memory object for each node in the tree, similar to the VCL's TTreeNode.

The general properties of the list window objects also work for the tree:

Count - returns the number of nodes in the tree (however, if the number of nodes exceeds 65536, the system always returns 0, and cannot show the scroll bar);
ImageListNormal - a list of images for storing thumbnails of the main images for elements;
ImageListState - a list of images for storing state icons;
OnSelChange - an event that is triggered when another node becomes selected in the tree;

Properties, events, and methods that can be considered specific to the treeview object begin with the TV prefix.

5.17.1 Properties of the whole tree

TVSelected - returns a handle to the current node in the tree. This node is also selected. If there is no such node in the tree, then 0 is returned. By assigning a descriptor of some node to this property, thereby it can be made current (other selected ones are no longer selected);
OnTVSelChanging - the event occurs before the node is allocated. The event handler can cancel the selection operation by returning true, and thereby prevent the selection of any nodes in the tree;
OnSelChange - this event is triggered to view the tree after changing the currently selected node;
TVRightClickSelect - this property determines for the tree window whether right-clicking on an unselected node will make it selected;
TVDropHilighted and **TVDropHilited** are synonyms for the same property. Return (and set) a node that is designated (and visually rendered differently) as the target node for dropping an object or node being "dragged" by the mouse (drag and drop operation);
TVRoot - descriptor of the first top-level node (0, only if the tree is empty);
TVFirstVisible - returns (and allows to set) the first visible node in the client side of the tree;
TVIndent - the size (in pixels) of the padding of child nodes in relation to the parent. Allows you to set the desired value other than the default;

5.17.2 Adding and removing nodes

TVInsert(Parent, After, s) and **TVInsertW (Parent, After, s)** - creates a node that is a child of Parent, in order following the After node, and sets the string s as text for it;
TVDelete(Node) - deletes the specified node (together with the node, all subordinate nodes are automatically deleted - recursively);
OnTVDelete - an event that is triggered when each node in the tree is deleted. A handler for this event can, for example, release memory, objects, or other resources associated with the item being removed (TVItemData);
Clear - the same as for other visual elements, it works for viewing the tree, deleting all its nodes.

© 2024 Carl Peeraer

5.17.3 Properties of parent nodes

TVItemChild[Node] - returns the descriptor of the first child node for the given Node (or 0 if the Node has no subordinates in the tree);
TVItemHasChildren[Node] is not just about checking that a Node has child nodes. It is possible to use this property to indicate to the window that the Node node "has" child nodes, and from that moment on, a "+" button will be displayed opposite this element, allowing you to expand this node, even if there were no subordinate nodes in it. At the same time, at the time of plowing, you can add these same nodes. This feature can be used to postpone the loading of those nested nodes that are not yet required for the future, and thereby increase the speed of the initial filling of the contents of the tree;
TVItemChildCount[Node] - returns the number of child nodes for a Node directly nested within it;
TVItemExpanded[Node] - Determines if the node is open. By assigning the value true to this property, you can ensure its expansion (false - closing);
TVItemExpandedOnce[Node] - returns true if the node has been expanded at least once (after which this flag can be cleared only by deleting all child nodes);
TVExpand(Node, flags) - expands or collapses the Node, depending on the flags. You can pass a combination of flags in the flags parameter:

- **TVE_COLLAPSE** - collapses the node;
- **TVE_COLLAPSERESET** - in addition to collapsing, removes children;
- **TVE_EXPAND** - opens the knot;
- **TVE_TOGGLE** - if the node is open, closes it, and if it is open, then it slams;

TVSort(Node) - Sorts the tree starting from the specified node. If 0 is passed as the Node parameter, the entire tree is sorted;

5.17.4 Properties of child nodes

TVItemParent[Node] - returns the parent node for the given one;
TVItemNext[Node] is the next sibling node in the tree after Node (a child for the same parent). Returns 0 if Node is the most recent child in the same parent;
TVItemPrevious[Node] is the previous node in the tree. Returns 0 if Node is the first top-level node;
TVItemNextVisible[Node] - the next node displayed in the window;
TVItemPreviousVisible[Node] - the previous element displayed in the tree window;
TVItemVisible[Node] - Checks if a node is "visible", in the sense that all its parents in the tree hierarchy are open. By setting this property to True, it is possible to ensure the "visibility" of the node (all of its unrevealed parents are opened at the same time);

5.17.5 Node attributes: text, icons, states

TVItemText[Node] and **TVItemTextW [Node]** - node text;
TVItemPath(Node, s) and **TVItemPathW (Node, s)** - returns the "path" from the root node to the specified Node as a concatenation of the text of all nodes on this path (strings are separated by s);

© 2024 Carl Peeraer

TVItemImage[Node] - the main icon for a tree node when it is not selected (for the selected state, another icon is used, set by the **TVItemSelImg** property). Use a value of -1 to be able to set the icon in the **TVN_GETDISPINFO** message handler. In order to have no pictogram at all, the value -2 should be used. When assigning a value to this property, the same value is simultaneously assigned to the TVItemSelImg property;
TVItemSelImg[Node] - an icon for a node in its selected state;
TVItemOverlay[Node] - drawing on the main icon. A value of 0 means no overlays, values from 1 to 15 use overlay icons from the same ImageListNormal list of images where the main icons come from (see the Overlay [] property for a list of icons);
TVItemStateImg[Node] - "state" icon, which is taken from a separately specified list of images ImageListState;
TVItemData[Node] - number or pointer associated with a node in the tree;
TVItemBold[Node] - a special property that allows you to display the text of a node in a bold font;
TVItemCut[Node] - a special visual effect for a node in a tree, which is usually used to display the nodes selected for the operation of "cutting" and subsequent insertion;
TVItemDropHilighted[Node] and **TVItemDropHilited [Node]** are property synonyms for a node that provide a special visual effect for a node. This is how elements in the tree are usually depicted that are the target for the object being dropped by the mouse (drag and drop). Unlike the TVDropHilited property, which specifies the only element in the tree intended for exactly this operation, this property simply changes the appearance of nodes, and the number of such nodes is not limited to a single one;
TVItemSelected[Node] - the "node is selected" property. Several nodes can be "selected" in a tree;

5.17.6 Node geometry and drag

TVItemRect[Node, textonly] - returns the rectangle occupied by the node in the client area of the window (all or only its text). If the node is partially visible in the window, the returned rectangle can go beyond the bounds of the client rectangle. If the node is not visible at all, a rectangle with all zero coordinates is returned;
TVItemAtPos(X, Y, where) - returns the handle of the node located in the client part of the object window at the point with coordinates (X, Y). In this case, the where variable returns what part of the given node is at this point: the main icon, the status icon, the text or part of the node to the right of the text or to the left of the displayed part of the element;
OnTVBeginDrag - an event that is triggered when the operation of starting dragging a tree node with the mouse is recorded;

5.17.7 Editing text

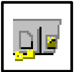
TVEditItem(Node) - programmatically starts editing the text of the node;
TVStopEdit(cancel) - programmatically terminates the editing of the node text. The cancel parameter determines whether the edit will be canceled or completed successfully;
TVEditing - checks if the window is in the text editing mode of the current node;

© 2024 Carl Peeraer

OnTVBeginEdit - an event that is triggered when editing the text of a node in the tree starts, for any reason (user or programmatically). Among other things, the handler can prohibit the ability to edit text for individual elements in the tree by returning false;
OnTVEndEdit - event of completion of editing the text of the node. In particular, the event handler can substitute its own text instead of the entered string;
Moreover:
SetUnicode - switches the object window to the mode of working with Unicode strings (nevertheless, the UNICODE_CTRLS conditional compilation symbol must be present in the project options);

In MCK, the tree is represented by the mirrored **TKOLTreeView** component.

5.18 Tool Bar



A visual of this type is usually used in an interface where, among the many commands available from the menu, there are a number of the most frequently used operations that can be conveniently performed by clicking on the icon associated with this action. For example, it is very easy for a user to remember that an open folder icon is associated with a file open operation.

Sometimes, on the contrary, if the number of commands is too small, the toolbar can completely replace the main menu. Usually, this is exactly the situation with secondary forms, on which their main menu is used extremely rarely (I do not remember a case when I had to use the main menu on additional forms in my projects).

A very convenient feature of the toolbar object is that it automatically provides floating text prompts for each button (of course, the application developer determines the text of the prompts). Thus, if not only the name of the action to be performed, but a combination of hot keys by which it can be performed is placed in the text of the hint, then the tool bar also turns into a tool for self-documenting the application, and into a tool for visual training of the user when he is just getting acquainted with the capabilities of the program ...

Most often, the toolbars are located at the top of the window, just below the menu. This is the most ergonomic location: if the user, choosing the action to be performed, changed his mind and wanted to execute a command that is in the menu (but which is not among the buttons on the ruler), then he does not need to move the mouse cursor far: the menu is very close.

But there is no special need to place the tool ruler on the upper part. Moreover, unlike the main menu, it is possible to place an arbitrary number of tool bars in the window (as long as there is space), arrange them not only horizontally, but also vertically, make them dynamically hidden, etc.

© 2024 Carl Peeraer

In fact, the ruler is a panel with buttons. But one window is responsible for the image and functioning of these buttons. (By the way, I forgot to say: if you need to have a large number of buttons, the ruler also saves window handles, which can be useful for improving the performance of the application and the entire operating environment). The programmer sets only the parameters of the buttons (text, images, features of functioning), and event handlers for mouse clicks on the buttons or on the ruler itself.

Object constructor:

`NewToolbar(Parent, align, options, bitmap, buttons [], imgindexes [])` 346 - allows directly in the object constructor to set, in addition to the parent and general options of the ruler, also the list of buttons, i.e. names and / or tooltips for them, and a set of icons in the form of a single bit image. The align property specifies for the toolbar whether it is automatically aligned to the top of the parent window by its own windowing means. This alignment does not cost anything for the size of the program. does not add generic code to the application to align child visuals with parent visuals (as it does when using the Align property of a TControl).

There are the following options for a ruler object:
tboTextRight - the text on the buttons is located to the right of the icons, and not below, as by default. Note that this option can affect the appearance of the ruler even if the text in the buttons is not used at all;
tboTextBottom - the text is located under the icons;
tboFlat - flat buttons (the border appears only when you hover over the buttons with the mouse cursor);
tboTransparent - property of "transparency" of the ruler (it is also provided by means of the window itself, and the general transparency is not involved);
tboWrapable - the buttons are automatically transferred to the bottom line when the right edge of the window is reached (should be used only for vertically arranged rulers, since KOL does not provide an automatic change in the height of the ruler window when such a transfer occurs);
tboNoDivider - prohibits drawing along the top edge of the ruler the dividing line, which the ruler window displays by default;
tbo3DBorder - adds a pseudo-three-dimensional border around the ruler (as a result, the ruler is deeply "depressed").

The buttons parameter is an array of strings that define for each button its type and its text - in one line. An empty string is used as a sign of the end of the array of buttons, therefore, even if the button does not have a caption, it must be given a text containing at least one space. Note: In order for such buttons to display correctly in Windows 9x, the tboTextRight option must be used. A separator button is specified by the string '-' (ie, the string for such a button contains a single minus sign). To create dockable buttons, the text is specified as usual, but it is preceded by the prefix character '-' or '+' (which one specifies whether the button will initially be depressed or depressed). In order to combine the fixed buttons into groups, after the prefix sign '+' or '-', there is a prefix exclamation mark, for example: '+! x', '-! y'. In a group, only one button can be pressed (this is provided by the window). It is also possible to define a "drop-down"

© 2024 Carl Peeraer

button, which is displayed with an additional "down triangle" sub-button on the right side of the button. The prefix character '^' is used to define the dropdown button.

In case the buttons are only for displaying icons, the text in the buttons can still be specified, in which case it can be used to define a set of tooltips. It is also possible to specify the list of tooltips additionally, in which case the tooltips may differ from the text assigned to the buttons themselves.

The bitmap parameter can set a bitmap of thumbnails for some buttons. If such an image is specified in the ruler constructor, then the width and height of the icons are taken equal to the height of this image. To use a different width of images, you should initially pass 0, then change the width by assigning a value to the TBBtnImgWidth property, and only then assign the image. In this way, you can also use several images, adding them sequentially (but the width of the icons must be the same).

Now is the time to remind about the existence of the LoadMappedBitmap functions and others that allow, when loading an image from resources, to ensure their adaptation to the system colors settings that take place at that moment. These functions can be used to turn a fixed light gray color into the clBtnFace color for displaying button faces, etc. - so that the ruler does not look like a foreign body on the desktop, if the used color scheme differs from the standard one.

Thumbnails are considered in the image following from left to right, and are numbered by indices starting from zero. Again, the imgindexes array tells the buttons the thumbnail indices from this bitmap. An important detail: when icons are mapped to buttons in a list of text strings, separator buttons are not counted (they still cannot be pictured).
Moreover, if the array of image indices is less than the number of buttons defined in the buttons array, then other buttons receive indices in ascending order. Ie, for example, it is enough to set an array of a single element [0], which assigns the first button in the array an icon with index 0, so that all other buttons get icons 1, 2, ...

When setting a bitmap image for icons, it is allowed to specify special (negative) values in place of its descriptor: as a result, the image for buttons is taken from the system standard set of images for toolbars. The choice of system images is not very rich, but it allows, at least for some of the actions of the standard type, to use standard icons, giving the appearance of the application a certain rigor, and making it easier for the user to remember new icons.

It is also possible to use not one bitmap, but a list of images. More precisely, up to three lists. One for the normal enabled state of the buttons, one for the disabled (disabled) state, and the third for the highlighted (hot) state. To do this, initially pass 0 as the bitmap parameter, and then assign the thumbnail lists additionally using the Perform method and passing TB_SETIMAGELIST, TB_SETDISABLEDIMAGELIST, TB_SETHOTIMAGELIST messages. Note: when designing in MCK for the toolbar mirror, there are design-time properties imageListNormal, imageListDisabled, imageListHot, which ensure the generation of the corresponding code.

Buttons on a toolbar can be identified by their numeric "identifiers" assigned to them when buttons are added, or by indexes. When programming in MCK, there is (and is enabled by

© 2024 Carl Peeraer

default) the ability to automatically generate symbolic constants that allow you to use identifiers assigned at the development stage to identify buttons. Because the buttons are assigned TB1 names by default. TB2, ..., then when multiple toolbars are used on a form, MCK generates code that specifies the values for such constants more than once. Such code, from the point of view of the compiler, is erroneous and cannot be compiled. To avoid such a situation, you should either rename the buttons (TBOpen, TBSave, ... - and then refer to them in this way),

Now I will list the methods, properties and events of the toolbar:

5.18.1 General properties, methods, events

OnClick - the event of a mouse click on the ruler, including any of its buttons. There is a synonym for this event, OnTBClick;
RightClick - this property can be interrogated in the OnClick event handler in order to distinguish between right-clicking and left-clicking;
CurIndex - can be used to clarify which button was pressed in the handler of the general event OnClick or OnDropDown - for the entire ruler. If the click happened "past" all the buttons on the ruler itself, then the CurIndex property returns -1. The use of a common event saves some code size, since there is no need to draw up a separate procedure for processing each button, and when initializing the ruler, it is enough to assign only one event handler. But it is also possible to assign separate click handlers for each button on the ruler;
TBCurItem - similar to **CurIndex**, but returns a numeric handle, not the index of the pressed button;
Count - returns the number of buttons, similar to **TBButtonCount** (property synonym);
OnDropDown - This event is triggered when the user clicks on a button created as a "dropdown" with a '^' prefix in the text. What exactly will drop out in this case, and from what position on the screen, is entirely determined by your code;
IsButton - returns true for the ruler object (although, of course, it is not a button, but the return value is used for internal purposes, to support mnemonics, i.e. keystrokes from the keyboard using the shortcut Alt + <letter>);
SupportMnemonics - calling this method provides a code for the ruler that provides automatic triggering of buttons by pressing Alt + <letter> mnemonic key combinations, where letter is a character from the button text, preceded by the '&' prefix. This method can be called for the entire form to provide mnemonics for all window objects in the form (menus, buttons). Or, this method can be called on the Applet object to provide this functionality for all forms in the application in a single call;
TBAutoSizeButtons- if set (TRUE, this value is used by default - unless there is a TBBUTTONS_DFLT_NOAUTOSIZE symbol in the project options), then the size of the buttons is adjusted to the size of the text and icons individually. If the value is FALSE, all buttons are the same size.

Note that if this parameter is different from the default, then you should change it before creating the buttons. For example, initially set empty arrays buttons and imgindexes in the parameters of the NewToolbar constructor 364*, then change this property for the created button bar, and only after that add buttons with the TBAddButtons functions.*

© 2024 Carl Peeraer

TBButtonsMinWidth - the minimum width of the buttons;
TBButtonsMaxWidth- the maximum width of the buttons. These two parameters must also be changed before buttons are added;
TBRows - allows you to set the number of lines that will be used to wrap buttons to the next line if the ruler is not wide enough.

5.18.2 Setting up the ruler

TBAddButtons(buttons, imgindexes)- adds buttons to the ruler in the same way as in the ruler constructor. It is possible to add buttons with several different calls to this method and the TBInsertButtons method;
TBInsertButtons(i, bitmap, imgindexes) - inserts the specified buttons at position i;
TBDeleteButton(btnID) - removes the button (by its numeric "identifier");
TBDeleteBtnByIdx(i) - removes the button by index;
Clear - removes all created buttons from the ruler;
TBSetTooltips(btnID, tooltips) - sets the pop-up text of tooltips for buttons starting with btnID;
TBBtnImgWidth - the width of the thumbnail for the button in the (first) image added in the TBAddBitmap method. To use this property, the number 0 should be passed as the bitmap parameter in the ruler constructor, and the image should be added after changing the value of this property using the **TBAddBitmap** method. This property should not be used if standard images are used for at least some of the buttons (in this case, the icons are always square, 16x16 or 32x32 pixels in size);
TBAddBitmap(bitmap)- adds a bitmap to the line of pictograms for toolbar buttons. On first addition, if the TBBtnImgWidth property has not changed, the height of the entire image is used as the width of each thumbnail. With subsequent additions, the width of the icons can no longer be changed, and if the parameters of the new images (height) differ, the height of the first bitmap added (by this method, or in the ruler constructor) is still used.
In addition, this method can be used to "add" system images to the rulers by using the reserved numeric values in the bitmap parameter (-1 - standard 16x16 small icons, -2 - standard 32x32 large icons, -5 and -6 - standard small and large "view" icons, -9 and -10 are standard small and large "history" icons).

The use of system images has a beneficial effect on the size of the application. pictures are taken from system resources, and are not stored in the executable module, spending 1 K byte (minimum) for each button.

TBAssignEvents(btnID, events) - assigns individual events to the ruler buttons, starting with the button identified by the numeric btnID descriptor;
TBResetImgIdx(btnID, i) - "resets" the indexes of icons for i buttons, starting with the given btnID;
TBItem2Index(btnID) - returns the button index by its numeric descriptor;
TBIndex2Item(i) - returns a handle to the button by its index;
TBConvertIdxArray2ID(idxVars) - "converts" indices to identifiers for the specified set of numeric variables. You should initially assign button indices to these variables, and after calling

© 2024 Carl Peeraer

TBConvertIdxArray2ID, all of these variables can be used as indices to refer to the buttons. This method allows you to combine the convenience of using symbolic names as button indices, when referring to them from the program code, with the ability to dynamically change the composition of the line. If buttons are inserted or deleted, then the correspondence between the symbolic names of the buttons and their descriptors obtained in this way is not violated.

5.18.3 Button properties

In subsequent properties, you can successfully use both IDs and indexes to identify buttons. The value of the parameter indicates that we are talking about an identifier and not about an index. If it is less than 100, then it is the index that is meant. Hence the conclusion: the number of buttons on the ruler should not exceed 100. However, this limitation does not seem excessive to me.
TBButtonEnabled[i] - button "available";
TBButtonVisible[i] - the button is visible (when you hide the button, the buttons located to the right of it are shifted to its place);
TBButtonChecked[i] - the "depressed" button (it makes sense to use only for fixed buttons, which were created with the prefix '-' or '+');
TBButtonMarked[i] - the button is highlighted (this property can also be set to true or false);
TBButtonPressed[i] - the button is "pressed";
TBButtonText[i] - the text of the label on the button;
TBButtonImage[i] - the index of the icon for the button;
TBButtonSeparator[i] - the button is "dividing" (in fact, a narrower dividing strip is depicted - flat or depressed);
TBButtonRect[i] - returns the rectangle occupied by the button on the ruler;
TBButtonWidth[i] - returns (and allows to change) the width of the button;
TBButtonAtPos(X, Y) - returns the handle of the button located on the ruler at the point with the specified coordinates (value -1 is returned if there are no buttons at this point);
TBBtnIdxAtPos(X, Y) - similar to the previous method, but returns the button index;
TBMoveButton(i, j) - moves the button with index i to position j;

5.18.4 Some features of working with the toolbar

In order to change the height of a horizontally located ruler, in OS Windows you have to change the height of the icons, and there is no other way to control the height of this visual element. Even if you don't use icons but want to increase the height of the ruler, create an image list with an image width of 1 pixel (ImgWidth = 1), and the desired height, and assign it as a set of icons for the ruler, without even adding to this picture list no picture. In the case of using MCK, it is enough to drop the TKOLImageList component onto the form, set ImgWidth = 1 to it in the Object Inspector, and for the TKOLToolbar mirror, assign this list of images as the main set of icons (ImageListNormal property).

Sometimes it becomes necessary to position the ruler vertically. In this case, you can do one of the following. Or, you must specify the number of lines in which the buttons should be placed (and this number must be the same as the number of buttons). Alternatively, you should set the tboWrapable option and make the ruler wide enough to wrap the buttons to the next line.

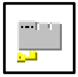
© 2024 Carl Peeraer

There is a known issue with incorrect rendering of the toolbar with the tboFlat option in Windows XP when themes are enabled: the background of the ruler turns black. To resolve this issue, use the design-time property FixFlatXP (it is enabled by default). When this option is enabled, when this OS version (or higher) is detected, the tboFlat option is not enabled.

In addition, I note that the toolbar window uses window messages TB_xxxx to control itself, which for some unknown reason appeared at the beginning of the user message range WM_USER + n. If the form itself receives such messages in the first place, intercepting them for its own purposes, it is quite possible that the toolbar will not respond to some requests and commands implemented through the same messages. (The same can apply to some other window controls in Windows, for example - list view or tree view). So my advice is not to use the first hundred of custom posts for your own purposes.

The **TKOLToolbar** component is the **MCK mirror** for this kind of control. It allows you to customize the ruler at the development stage, and not rack your brains over what calls and in what order to make the ruler looks exactly the way the developer wants it.

5.19 Tab Control



This visual control, encapsulated in the TControl type, also applies to list visual objects (its "items" are bookmarks, along with the corresponding pages). The main purpose of this object is to provide the presence of several "pages", or panels, with their own set of child visual objects located in each of them, and a set of bookmarks for switching between these pages. There is also a particular task: to ensure the presence of programmatically switchable pages, without giving the user the opportunity to independently navigate to any of these pages. This goal is also not difficult to achieve using this kind of window object (by hiding tabs).

Constructors:

`NewTabControl(Parent, tabs, options, imglist, imgidx1)` 346 - creates a multi-page object, immediately adding to it a number of bookmarks, specified by the composition of the lines in the tabs array, and assigning these lines to these bookmarks as the text of the bookmarks. The presence of the imgidx1 parameter allows you to combine the use of one list of images for some purposes: the indexes of icons to be displayed in bookmarks are assigned starting from the value of this parameter. Of course, image lists are optional, so it is okay to pass nil as the imglist parameter.

`function NewTabEmpty( Parent, options, imglist)` 346
Creates new empty tab control for using metods TC_Insert (to create Pages as Panel), or TC_InsertControl (if you want using your custom Pages).

© 2024 Carl Peeraer

The following options are defined for tabbed panels:
tcoButtons - bookmarks look like buttons;
tcoFixedWidth - fixed (the same for all) bookmark width;
tcoFocusTabs - draw a frame in a bookmark;
tcoIconLeft - display the icon in the tab on the left, and the text on the right;
tcoLabelLeft - display the icon on the right, and the text on the left;
tcoMultiline - bookmarks are placed on several lines;
tcoMultiselect - multiple selection of bookmarks;
tcoFitRows -
tcoScrollOpposite -
tcoBottom - bookmarks are located at the bottom;
tcoVertical - bookmarks are located on the left (if the tcoBottom option is present, on the right);
tcoFlat - "flat" bookmarks;
tcoHotTrack - "hot" highlighting of the bookmark under the mouse cursor;
tcoBorder - border around the entire window;
tcoOwnerDrawFixed - the OnDrawItem event handler is called to draw the contents of the bookmarks.

Properties, methods, events:

CurIndex - index of the current bookmark;
IndexOf(s) or **TC_IndexOf (s)** - returns the index of the page with the specified text in the bookmark;
SearchFor(s, i, partial) or **TC_SearchFor (s, i, partial)** - similar to **IndexOf**, but additionally allows you to specify the index after which to start the search, and set the way to compare the text of the element with the pattern (partial comparison of the first characters);
OnChange - this event is triggered when another bookmark becomes current (programmatically or as a result of user actions);
ImageListNormal - access to an object that provides images of icons for display in bookmarks;
SetUnicode(b) - allows you to switch the object window to the mode of working with Unicode strings (it is required to include the UNICODE_CTRLS conditional compilation symbol in the project options);

Special properties characteristic of this particular type of object:

Pages[i] or **TC_Pages [i]** - access to object panels. For example, to create a label on the panel with index 0, you should call:

```
NewLabel (Tabcontrol1.Pages [0], 'text');
```

TC_Insert(i, s, ii) - inserts one more bookmark (together with the panel);
TC_Delete(i) - deletes the bookmark with the specified index;
TC_Items[i] - access to the text of the i-th bookmark;
TC_Images[i] - managing the index of the icon in the bookmark;
TC_ItemRect[i] - the rectangle occupied by the bookmark in the window of the whole object;

© 2024 Carl Peeraer

TC_SetPadding(cx, cy) - sets the indent from the edge of the bookmark to the text in it;
TC_TabAtPos(X, Y) - returns the index of the bookmark located in the object window at the specified coordinates (or -1 if there are no bookmarks in this position);
TC_DisplayRect - the rectangle occupied by the client part of the current page in the object window (in fact, for all pages this rectangle is the same, because when another page becomes the current one, it is simply shown "in front" of all the others, obscuring them from view user). It is this rectangle that is convenient to use in order to "crop" the edges of the object along with the tabs at runtime, making them invisible and inaccessible to the user. For example, like this:

```
var Rgn: HRgn;
...
Rgn: = CreateRectRgnIndirect (Tabcontrol1.TC_DisplayRect);
SetWindowRgn (Tabcontrol1.Handle, Rgn, true);
DeleteObject (Rgn);
```

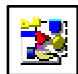
If at the same time. for example, place bookmarks at the bottom (**tcoBottom** option), then after trimming the bookmarks, the object looks almost the same as if it were a regular panel (without a convex or indented border). The remaining space at the bottom can be used to place (on the other parent) some buttons. When designing in MCK, such border cropping, of course, does not change the appearance of the object, and it is still possible to switch between bookmarks (by double-clicking on the bookmark).

The **Mirror Classes Kit** has a mirror component **TKOLTabControl** for the bookmarked pages object. Many beginners to work with MCK, having dropped it on the form, do not know what to do with it (how to add bookmarks, delete or move them). I think the information below will come in handy.

To set the initial number of bookmarks during development, or to increase this number, you need to change the value of the Count property - in the Object Inspector. For example, enter the number 3 to create initially three bookmarks. In order to select a certain bookmark as the current one during the configuration of the form (design time), double-click on it (just on the bookmark). In order to delete a bookmark, you need to make it current, select its panel, and press the <Delete> key on the keyboard. Finally, to change the order in which the tabs are displayed, use the **TabOrder** property of the panel. For the changes to take effect, you can, for example, make a double click on the object window, in the area free of both tabs and panels.

© 2024 Carl Peeraer

5.20 Frames (TKOLFrame)



Frames originally emerged as a way to dynamically create multiple sets of visual objects, grouped together, positioned in a certain way, and once customized. For example, a group of a pair of input elements (EditBox353), a checkbox (CheckBox352) and a button can be combined in one frame, after which the required number of such groups can be created on the parent scroll box to manage many such objects. In this case, all groups will look and function the same.

It makes sense to talk about this component only in the context of the **Mirror Classes Kit**. A frame during design time is a form on which other visual and non-visual components can reside. To organize a frame, unlike a form, put a **TKOLFrame** component on a Delphi form instead of a **TKOLForm** object.

But unlike a form, at runtime, a frame is a panel that can be created on an existing form or other parent visual that allows children (for example, a scroll box created by calling NewScrollBox or another panel). In this case, the function for creating a frame generated by the MCK components actually creates a panel, and all child elements of the frame, in accordance with what was specified at the design stage.

Accordingly, a frame should be created immediately as a child of the object on which this frame is supposed to be located. For example:

```
NewfrmMyFrame (MyFrame1, Panel1);
```

This call positions the frame to be created on Panel1 by assigning the pointer to the frame to be created in the variable MyFrame1. The type of the variable MyFrame1 must be **PfrmMyFrame**, and like the form in MCK, it is not the created panel itself, but provides a container object for this panel and its children. The frame panel itself becomes available through the Form field.

It looks, perhaps, a little confusing, but in fact, this is practically the only possible option. Indeed, after creating a frame, in order to provide some features of its functioning, the code will also need to refer to its elements. For example, if **frmMyFrame** contains **Panel1** and **EditBox1**, then you should use the compound names **MyFrame1.Panel1** and **MyFrame1.EditBox1** to refer to these objects in your code. In light of the above, it is easy to draw a parallel with the form and it becomes clear that the frame panel itself as an object of the PControl type should be available to the programmer exactly as **MyFrame1.Form**.

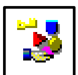
Note that you yourself create (declare, use) the MyFrame1 variable, MCK does not do this for you. The explanation is simple. It is entirely up to you where such a variable is located, whether it is a global, local, form field, or in general you store a list of such pointers in an object of the PList type or in object string pairs of the PStrListEx object. Or maybe you are not going to save a pointer to the frame at all, and after creating it and specifying some initial settings for it, it continues to live its own life (until the death of the parent visual object).

© 2024 Carl Peeraer

Just in case, make sure that the Delphi form on which the **TKOLFrame** is located instead of the **TKOLForm** is not included in the list of automatically generated forms. The function call is supposed to be `NewfrmMyFrame` you will do it yourself, in the place of the code where you need it.

If it is necessary to terminate the existence of a frame before the expiration of the parent's lifespan, then proceed in the same way as with the form: MyFrame1.Form.Free... And, of course, do not forget to reset the MyFrame1 variable for yourself, or in some other way ensure the impossibility of accessing a no longer existing object in the application code.

5.21 Data Module (TKOLDataModule)



Similar to the VCL data module (TDataModule), this MCK object is also created to organize a module containing only non-visual objects. The only difference from the usual form at the design stage is that the **TKOLDataModule** object is used instead of the **TKOLForm** object. The difference at runtime is that calling the function that constructs this module does not create a form, even invisible. Similarly to the MCK form, a "container" of objects is also created, and all objects specified at the development stage are built for it.

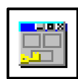
In contrast to a frame, a data module is most often used in a single copy (although this is not an axiom), and in this case it is just convenient to leave the corresponding Delphi form in the auto-created list, and refer to this object through the corresponding global variable.

In particular, it is allowed to change the order of automatic creation of forms so that the module with the data is created first. In itself, this is indifferent. It is entirely up to you to control access to objects that have not yet been created in your code.

However, do not try to use a module with data instead of a form, that is, using this object to create an application without forms will still fail. If you need an application without a form at all, then MCK in this case will not be needed at all (as well as a data module).

© 2024 Carl Peeraer

5.22 The Form



Finally, mention should be made of the form itself, which is also a visual object. In the case of designing without MCK, the form is created by calling

```
MyForm: = NewForm (Applet, 'form caption');
```

Always substitute the Applet variable as the parent parameter in the call. Even if the applet is not used, and the Applet variable is nil, the use of the Applet variable will still be correct, and if you decide that you still need to use a separate Applet object in the project, then you will not have to rewrite the code that constructs the forms. In this case, it will be enough to insert a call before creating the first form.

```
Applet: = NewApplet ('applet title');
```

Of course, when designing using MCK, you may not need to write out the code for constructing the form (if all forms are created automatically, and in a single copy). But even in the case of using MCK, nevertheless, the forms may need to be created dynamically. For example, if there are a lot of forms in the project, then to speed up the initial start of the application, it makes sense to postpone the creation of forms until the moment when they are really needed. Or in the case when the same form can be designed more than once to display several of its instances on the screen at a time (without going into the details of why this might be needed).

In this case, pay attention to the specifics of using forms in the case of using MCK. Namely, at the design stage, the MCK mirror of the form (TKOLForm object) generates the code, and in particular, for the form named MyForm, it creates the global function NewMyForm. The definition of this generated function is located in the front-end of the module itself, and its implementation code is in the file <unit_name> _1.inc. To create a form object, the entry will now be different than in the case of "pure" KOL:

```
NewMyForm (MyForm1, Applet);
```

As you can see, the variable to which the created form object is assigned has moved from the left side of the assignment operator to the place of the first parameter, and the "title" parameter has disappeared. The form now gets the title in the "constructor" code generated by MCK, based on the settings you made when designing the form.

When designing in MCK, you must select the TKOLForm component on the form to change the properties of the form, and then change something in the Object Inspector. This is mentioned in another section of this book, but it will not be superfluous to repeat it, since we have already talked about the form.

Now I will focus on how the TControl object type works at runtime when it performs the functions of a form. In the case when MCK is used, this is the variable MyForm.Form, in the case of "pure" KOL, this is the variable MyForm itself of the PControl type.

© 2024 Carl Peeraer

There are the following features: in KOL, the form cannot process the OnClick message (although the OnMouseDown and OnMouseUp events work correctly). In addition, the KeyPreview property is not available by simply assigning TRUE to this property. You must first add the KEY_PREVIEW symbol to the project options.

5.23 "Alien" Panel

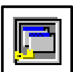
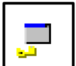
It is worth mentioning the relatively new feature in KOL to create a panel, for which an arbitrary window can be specified as a parent, including the window of someone else's application. For example, such a window could be the Windows taskbar. To create a "foreign" panel, use the constructor:

```
AP: = NewAlienPanel (ParentWnd, edgeStyle);
```

A “foreign” panel acts in the same way as a regular one, but its parent is specified not as a pointer to an object of the PControl type, but as a window handle. There are no other features. It should only be remembered that if the parent window is destroyed, the "foreign" panel will also be destroyed.

In MCK, there is no mirror for the foreign panel itself. you need to construct it with your own code. But to fill the "foreign" panel with content, you can use the frame mechanism described in two paragraphs above - if MCK is used for design.

5.24 MDI Interface

To conclude this section, let's take a look at creating MDI applications separately. A multi-document interface can be built either manually or using MCK. But in any case, the conditional compilation symbol should be added to the project options **USE_MDI** (This is done both to save code and to improve performance for all other cases - when an MDI interface is not required.)... With manual coding, the first step is to create a client window that is a child of the form. A common practice when designing an MDI interface is to use the entire form space except for the main menu, the toolbar (if present), and the status bar (also optional) for the client area. But it is not forbidden to place other visual elements, for example, on the sides of the client area. Client creation is done by calling the function

```
NewMDIClient (ParentForm, WindowMenuHandle)
```
345

Pay attention to the parameter WindowMenuHandle. This is a handle to a submenu where the system will automatically add the names of MDI child windows. If you specify 0 as this parameter, but the window names will not be added. If such a possibility is desired, then you should first create the main menu, and then pass the descriptor of the desired submenu as the second parameter. For example, like this:

© 2024 Carl Peeraer

```
var   MainMenu: PMenu;
   MainForm, MDI_Client: PControl;
MainMenu: = NewMenu (MainForm, 0, [
'New', '(', 'Create MDI Child', ')',
'Window', '(', 'Tile', 'Cascade', ')', ''],
    TOnMenuItem (MakeMethod (nil, @MenuItems)));
MDI_Client: = NewMDIClient (MainForm,
GetSubMenu (MainMenu, 1));
```

To create MDI child windows, use the function

NewMDIChild (ParentMDIClient, 'MDIChildName'); 345

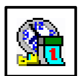
There can be several such windows, they can be created dynamically, which is quite common for an MDI interface. On such a child window, arbitrary visual elements can be located, as on a regular panel. When developing child windows, it should be taken into account that their sizes can change, in particular, the window can be maximized over the client window.

Visual development of an MDI interface using MCK is also possible. For which there are mirror classes **TKOLMDIClient** and **TKOLMDIChild**. It is likely that only the first of them will be useful, while child windows will still have to be created dynamically. To simplify creation, it is recommended to use TKOLFrame, on which the necessary visual objects are placed in development mode, and at runtime the resulting set of objects is simply cloned by calling the corresponding function.

It should be noted that some operations with MDI windows are performed in a slightly different way than with ordinary visual objects. For example, to programmatically maximize a certain client window, it is preferable to use sending a WM_MDIMAXIMIZE message to the client window, although it is possible to send a WM_SYSCOMMAND message with wParam = SC_MAXIMIZE to the child window itself.

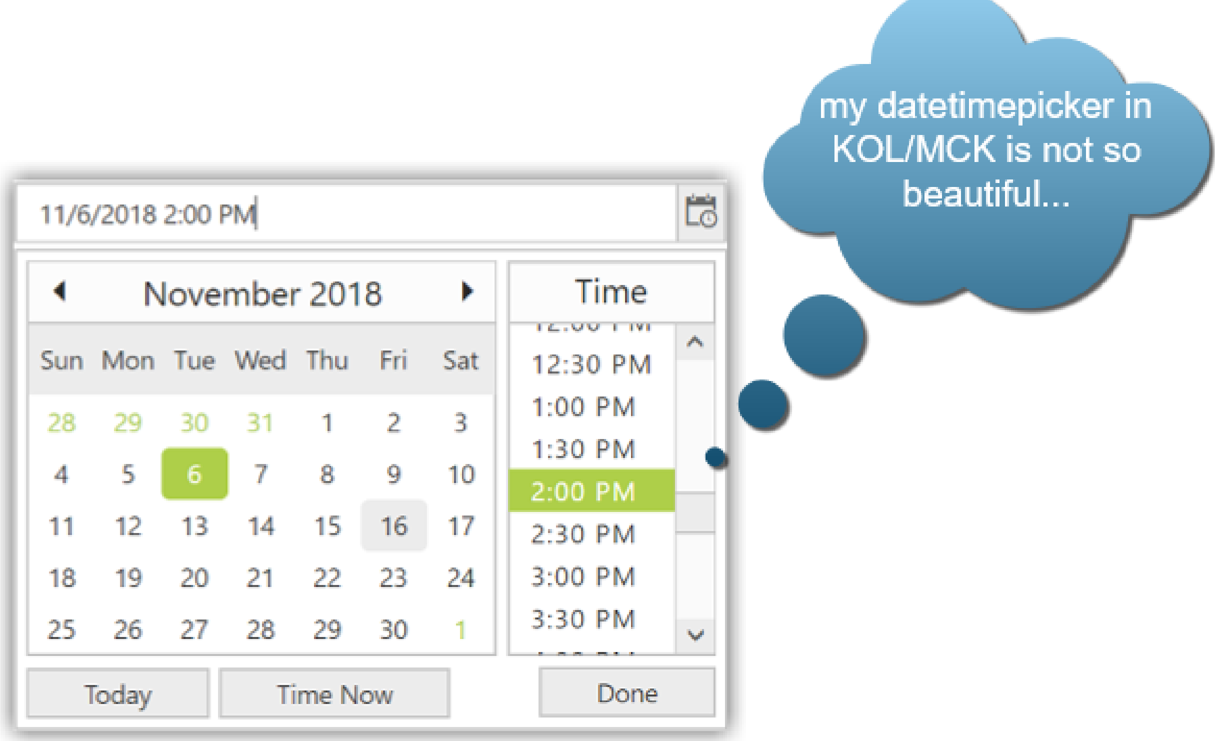
© 2024 Carl Peeraer

5.25 DateTime Picker



function **NewDateTimePicker(AParent: PControl; Options: TDateTimePickerOptions): PControl;** 346

Creates date and time picker common control.



5.26 Visual objects - Syntax

Following constructing functions for visual controls are available:

NewApplet 369	**function** NewApplet(**const** Caption: KOLString): PControl;
NewForm 367	**function** NewForm(AParent: PControl; **const** Caption: KOLString): PControl;
NewButton 350	**function** NewButton(AParent: PControl; **const** Caption: KOLString): PControl;
NewBitBtn 351	**function** NewBitBtn(AParent: PControl; **const** Caption: KOLString; Options: TBitBtnOptions; Layout: TGlyphLayout; GlyphBitmap: HBitmap; GlyphCount: Integer): PControl;

© 2024 Carl Peeraer

NewApplet 369	`function NewApplet( const Caption: KOLString ): PControl;`
NewLabel 346	`function NewLabel( AParent: PControl; const Caption: KOLString ): PControl;`
NewWordWrapLabel 346	`function NewWordWrapLabel( AParent: PControl; const Caption: KOLString ): PControl;`
NewLabelEffect 347	`function NewLabelEffect( AParent: PControl; const Caption: KOLString; ShadowDeep: Integer ): PControl;`
NewPaintbox 348	`function NewPaintbox( AParent: PControl ): PControl;`
NewImageShow 348	`function NewImageShow( AParent: PControl; AImgList: PImageList; ImgIdx: Integer ): PControl;`
NewScrollBar 349	`function NewScrollBar( AParent: PControl; BarSide: TScrollerBar ): PControl;`
NewScrollBox 350	`function NewScrollBox( AParent: PControl; EdgeStyle: TEdgeStyle; Bars: TScrollerBars ): PControl;`
NewScrollBoxEx 350	`function NewScrollBoxEx( AParent: PControl; EdgeStyle: TEdgeStyle ): PControl;`
NewGradientPanel 347	`function NewGradientPanel( AParent: PControl; Color1, Color2: TColor ): PControl;`
NewGradientPanelEx 347	`function NewGradientPanelEx( AParent: PControl; Color1, Color2: TColor; Style: TGradientStyle; Layout: TGradientLayout ): PControl;`
NewPanel 347	`function NewPanel( AParent: PControl; EdgeStyle: TEdgeStyle ): PControl;`
NewMDIClient 370	`function NewMDIClient( AParent: PControl; WindowMenu: THandle ): PControl;`
NewMDIChild 370	`function NewMDIChild( AParent: PControl; const ACaption: KOLString ): PControl;`
NewSplitter 348	`function NewSplitter( AParent: PControl; MinSizePrev, MinSizeNext: Integer ): PControl;`
NewSplitterEx 349	`function NewSplitterEx( AParent: PControl; MinSizePrev, MinSizeNext: Integer; EdgeStyle: TEdgeStyle ): PControl;`
NewGroupbox 348	`function NewGroupbox( AParent: PControl; const Caption: KOLString ): PControl;`
NewCheckbox 352	`function NewCheckbox( AParent: PControl; const Caption: KOLString ): PControl;`
NewCheckBox3State 353	`function NewCheckBox3State( AParent: PControl; const Caption: KOLString ): PControl;`
NewRadiobox 353	`function NewRadiobox( AParent: PControl; const Caption: KOLString ): PControl;`
NewEditbox 353	`function NewEditbox( AParent: PControl; Options: TEditOptions ): PControl;`
NewRichEdit 354	`function NewRichEdit( AParent: PControl; Options: TEditOptions ): PControl;`
NewRichEdit1 358	`function NewRichEdit1( AParent: PControl; Options: TEditOptions ): PControl;`

© 2024 Carl Peeraer

NewApplet 369	`function NewApplet( const Caption: KOLString ): PControl;`
NewListbox 358	`function NewListbox( AParent: PControl; Options: TListOptions ): PControl;`
NewCombobox 359	`function NewCombobox( AParent: PControl; Options: TComboOptions ): PControl;`
NewProgressbar 350	`function NewProgressbar( AParent: PControl ): PControl;`
NewProgressbarEx 350	`function NewProgressbarEx( AParent: PControl; Options: TProgressbarOptions ): PControl;`
NewListView 360	`function NewListView( AParent: PControl; Style: TListViewStyle; Options: TListViewOptions; ImageListSmall, ImageListNormal, ImageListState: PImageList ): PControl;`
NewTreeView 362	`function NewTreeView( AParent: PControl; Options: TTreeViewOptions; ImgListNormal, ImgListState: PImageList ): PControl;`
NewTabControl 366	`function NewTabControl( AParent: PControl; const Tabs: array of PKOLChar; Options: TTabControlOptions; ImgList: PImageList; ImgList1stIdx: Integer ): PControl;`
NewTabEmpty 366	`function NewTabEmpty( AParent: PControl; Options: TTabControlOptions; ImgList: PImageList ): PControl;`
NewToolbar 364	`function NewToolbar( AParent: PControl; Align: TControlAlign; Options: TToolbarOptions ; Bitmap: HBitmap; const Buttons: array of PKOLChar; const BtnImgIdxArray: array of Integer ): PControl;`
NewDateTimePicker 370	`function NewDateTimePicker( AParent: PControl; Options: TDateTimePickerOptions ): PControl;`

5.26.1 Function NewLabel

```
function NewLabel( AParent: PControl; const Caption: KOLString ): PControl;
```

Creates static text control (native Windows STATIC control). Use property Caption at run time to change label text. Also it is possible to adjust label Font , Brush or Color. Label can be Transparent . If You want to have rotated text label, call NewLabelEffect 347 instead and change its Font.FontOrientation.

Other references certain for a label:

Caption	`property Caption: KOLString;`
TextAlign	`property TextAlign: TTextAlign;`
VerticalAlign	`property VerticalAlign: TVerticalAlign;`

5.26.2 Function NewWordWrapLabel

```
function NewWordWrapLabel( AParent: PControl; const Caption: KOLString ): PControl;
```


© 2024 Carl Peeraer

Creates multiline static text control (native Windows STATIC control), which can wrap long text onto several lines. See also NewLabel346. See also:

Caption	`property Caption: KOLString;`

5.26.3 Function NewLabelEffect

```
function NewLabelEffect( AParent: PControl; const Caption: KOLString;
ShadowDeep: Integer ): PControl;
```

Creates 3D-label with capability to rotate its text Caption, which is controlled by changing Font.FontOrientation property. If You want to get flat effect label (e.g. to rotate it only), pass ShadowDeep= 0. Please note, that drawing procedure uses Canvas property, so using of **LabelEffect** leads to increase size of executable. See also:

Caption	`property Caption: KOLString;`
ShadowDeep	`property ShadowDeep: Integer;`

5.26.4 Function NewPanel

```
function NewPanel( AParent: PControl; EdgeStyle: TEdgeStyle ): PControl;
```

Creates panel, which can be parent for other controls (though, any control can be used as a parent for other ones, but panel is specially designed for such purpose).

5.26.5 Function NewGradientPanel

```
function NewGradientPanel( AParent: PControl; Color1, Color2: TColor ):
PControl;
```

Creates gradient-filled STATIC control. To adjust colors at the run time, change Color1 and Color2 properties (which initially are assigned from Color1, Color2 parameters), and call Invalidate method to repaint control.

5.26.6 Function NewGradientPanelEx

```
function NewGradientPanelEx( AParent: PControl; Color1, Color2: TColor;
Style: tGradientStyle; Layout: TGradientLayout ): PControl;
```

Creates gradient-filled STATIC control. To adjust colors at the run time, change Color1 and Color2 properties (which initially are assigned from Color1, Color2 parameters), and call Invalidate method to repaint control. Depending on style and first line/point layout, can looking different. Idea: Vladimir Stojiljkovic.

© 2024 Carl Peeraer

5.26.7 Function NewGroupBox

```
function NewGroupbox( AParent: PControl; const Caption: KOLString ):
PControl;
```

Creates group box control. Note, that to group radio items, group box is not necessary - any parent can play role of group for radio items. See also NewPanel347.

5.26.8 Function NewPaintBox

```
function NewPaintbox( AParent: PControl ): PControl;
```

Creates owner-drawn STATIC control. Set its OnPaint event to perform custom painting.

Canvas	`property Canvas: PCanvas;`

5.26.9 Function ImageShow

```
function NewImageShow( AParent: PControl; AImgList: PImageList; ImgIdx:
Integer ): PControl;
```

Creates an image show control, implemented as a **paintbox**348 which is used to draw an image from the **imagelist**174. At run-time, use property **CurIndex** to select another image from the **imagelist**174, and a **property ImageListNormal** to use another image list. When the control is created, its size becomes equal to dimensions of imagelist (if any).

5.26.10 Function NewSplitter

```
function NewSplitter( AParent: PControl; MinSizePrev, MinSizeNext: Integer ):
PControl;
```

Creates splitter control, which will separate previous one (i.e. last created one before splitter on the same parent) from created next, allowing to user to adjust size of separated controls by dragging the splitter in desired direction. Created splitter becomes vertical or horizontal depending on Align style of previous control on the same parent (if caLeft/caRight then vertical, if caTop/caBottom then horizontal).

Please note, what if previous control has no Align equal to caLeft/caRight or caTop/caBottom, splitter will not be able to function normally. If previous control does not exist, it is yet possible to use splitter as a resizeable panel (but set its initial Align value first - otherwise it is not set by default. Also, change Cursor property as You wish in that case, since it is not set too in case, when previous control does not exist).

Additional parameters determine, which minimal size (width or height - correspondently to split direction) is allowed for left (top) control and to rest of client area of parent, correspondingly. (It is possible later to set second control for checking its size with MinSizeNext value - using TControl.SecondControl property). If -1 passed, correspondent control size is not checked

© 2024 Carl Peeraer

during dragging of splitter. Usually 0 is more suitable value (with this value, it is guarantee, that splitter will be always available even if mouse was released far from the edge of form).

It is possible for user to press Escape any time while dragging splitter to abort all adjustments made starting from left mouse button push and begin of drag the splitter. But remember please, that such event is controlled using timer, and therefore correspondent keyboard events are received by currently focused control. Be sure, that pressing Escape will not affect to any control on form, which could be focused, otherwise filter keyboard messages (by yourself) to prevent undesired handling of Escape key by certain controls while splitting. (Use Dragging property to check if splitter is dragging by user with mouse).

See also: NewSplitterEx 349

OnSplit	**property** OnSplit: TOnSplit;
MinSizePrev	**property** MinSizePrev: Integer;
MinSizeNext	**property** MinSizeNext: Integer;
SecondControl	**property** SecondControl: PControl;
Dragging	**property** Dragging: Boolean;

```
function NewSplitterEx( AParent: PControl; MinSizePrev, MinSizeNext: Integer;
EdgeStyle: TEdgeStyle ): PControl;
```

Creates splitter control. Difference from NewSplitter 348 is what it is possible to determine if a splitter will be beveled or not. See also NewSplitter 348.

5.26.11 Function NewScrollBar

```
function NewScrollBar( AParent: PControl; BarSide: TScrollerBar ): PControl;
```

Creates simple scroll bar.

SBMin	property **SBMin**: Longint;
SBMax	property **SBMax**: Longint;
SBMinMax	property **SBMinMax**: TPoint;
SBPosition	property **SBPosition**: Integer;
SBPageSize	property **SBPageSize**: Integer;
OnSBScroll	property **OnSBScroll**: TOnSBScroll;

© 2024 Carl Peeraer

5.26.12 Function NewProgressBar

```
function NewProgressbar( AParent: PControl ): PControl;
```

Creates progress bar control. Following properties are special for progress bar:

Progress	`property Progress: Integer index(( PBM_SETPOS or $8000 ) shl 16 ) or PBM_GETPOS;`
MaxProgress	`property MaxProgress: Integer index(( PBM_SETRANGE32 or $8000 ) shl 16 ) or PBM_GETRANGE;`
ProgressColor	`property ProgressColor: TColor;`
ProgressBkColor	`property ProgressBkColor: TColor;`

```
function NewProgressbarEx( AParent: PControl; Options: TProgressbarOptions ):
PControl;
```

Can create progress bar with smooth style (progress is not segmented onto bricks) or/and vertical progress bar - using additional parameter. For list of properties, suitable for progress bars, see NewProgressbar350.

5.26.13 Function NewScrollBox

```
function NewScrollBox( AParent: PControl; EdgeStyle: TEdgeStyle; Bars:
TScrollerBars ): PControl;
```

Creates simple scrolling box, which can be used any way you wish, e.g. to scroll certain large image. To provide automatic scrolling of a set of child controls, use advanced scroll box, created with NewScrollBoxEx350.

```
function NewScrollBoxEx( AParent: PControl; EdgeStyle: TEdgeStyle ):
PControl;
```

Creates extended scrolling box control, which **automatically scrolls child controls** (if any).

5.26.14 Function NewButton

```
function NewButton( AParent: PControl; const Caption: KOLString ): PControl;
```

Creates button on given parent control or form. Please note, that in Windows, buttons can not change its **Font color** and to be Transparent .

Following methods, properies and events are (especially) useful with a button:

OnClick	`property OnClick: TOnEvent;`
SetButtonIcon	`function SetButtonIcon( aIcon: HIcon ): PControl;`
SetButtonBitmap	`function SetButtonBitmap( aBmp: HBitmap ): PControl;`

© 2024 Carl Peeraer

OnClick	`property OnClick: TOnEvent;`
LikeSpeedButton	`function LikeSpeedButton: PControl;`
Click	`procedure Click;`
Caption	`property Caption: KOLString;`
DefaultBtn	`property DefaultBtn: Boolean;`
CancelBtn	`property CancelBtn: Boolean;`
TextAlign	`property TextAlign: TTextAlign;`
VerticalAlign	`property VerticalAlign: TVerticalAlign;`

5.26.15 Function NewBitBtn

```
function NewBitBtn( AParent: PControl; const Caption: KOLString; Options:
TBitBtnOptions; Layout: TGlyphLayout; GlyphBitmap: HBitmap; GlyphCount:
Integer ): PControl;
```

Creates image button (actually implemented as owner-drawn). In **Options**, it is possible to determine, whether **bitmap or image list** used to contain one or more (up to 5) images, correspondent to certain BitBtn state.

For case of **imagelist** (option **bboImageList**), it is possible to use a number of glyphs from the image list, starting from image index given by **GlyphCount** parameter. Number of used glyphs is passed in that case in high word of **GlyphCount** parameter (if 0, one image is used therefore). For **bboImageList**, **BitBtn** can be **Transparent** (and in that case **bboNoBorder** style can be useful to draw custom buttons of non-rectangular shape).

For case of **bitmap BitBtn**, image is stretched down (if too big), but can not be **transparent**. It is not necessary for bitmap **BitBtn** to pass correct **GlyphCount** - it is calculated on base of bitmap size, if 0 is passed.

And, certainly, BitBtn can be without glyph image (text only). For that case, it is therefore is more flexible and power than usual Button (but requires more code). E.g., BitBtn can change its **Font**, **Color** , and to be totally **Transparent**. Moreover, BitBtn can be **Flat** , **bboFixed**, **SpeedButton** and have property **RepeatInterval**.

Note: if You use **bboFixed** Style, use **OnChange event** instead of **OnClick**, because **Checked** state is changed immediately however **OnClick** occure only when mouse or space key released (and can be not called at all if mouse button is released out of BitBtn bounds). Also, bboFixed defines only which glyph to show (the border if it is not turned off behaves as usual for a button, i.e. it becomes lowered and then raised again at any click).

Here You can find references to other properties, events and methods applicable to BitBtn:

© 2024 Carl Peeraer

OnBitBtnDraw	**property** OnBitBtnDraw: TOnBitBtnDraw;
OnTestMouseOver	**property** OnTestMouseOver: TOnTestMouseOver;
LikeSpeedButton	**function** LikeSpeedButton: PControl;
Caption	**property** Caption: KOLString;
BitBtnDrawMnemonic	**property** BitBtnDrawMnemonic: Boolean;
TextShiftX	**property** TextShiftX: Integer;
TextShiftY	**property** TextShiftY: Integer;
BitBtnImgIdx	**property** BitBtnImgIdx: Integer;
BitBtnImgList	**property** BitBtnImgList: THandle;
DefaultBtn	**property** DefaultBtn: Boolean;
CancelBtn	**property** CancelBtn: Boolean;
TextAlign	**property** TextAlign: TTextAlign;
MouseInControl	**property** MouseInControl: Boolean;
Flat	**property** Flat: Boolean;
RepeatInterval	**property** RepeatInterval: Integer;
ImageListNormal	**property** ImageListNormal: PImageList;
Checked	**property** Checked: Boolean;

5.26.16 Function NewCheckBox

```
function NewCheckbox( AParent: PControl; const Caption: KOLString ):
PControl;
```

Creates check box control. Special properties, methods, events:

OnClick	**property** OnClick: TOnEvent;
SetChecked	**function** SetChecked(**const** Value: Boolean): PControl;
Click	**procedure** Click;
Checked	**property** Checked: Boolean;
Check3	**property** Check3: TTriStateCheck;

© 2024 Carl Peeraer

5.26.17 Function NewCheckBox3State

```
function NewCheckBox3State( AParent: PControl; const Caption: KOLString ):
PControl;
```

Creates check box control with 3 states. Special properties, methods, events:

OnClick	`property OnClick: TOnEvent;`
SetChecked	`function SetChecked( const Value: Boolean ): PControl;`
Click	`procedure Click;`
Checked	`property Checked: Boolean;`
Check3	`property Check3: TTriStateCheck;`

5.26.18 Function NewRadiobox

```
function NewRadiobox( AParent: PControl; const Caption: KOLString ):
PControl;
```

Creates radio box control. Alternative radio items must have the same parent window (regardless of its kind, either groupbox (NewGroupbox 348), panel (NewPanel 347) or form itself). Following properties, methods and events are specially for radiobox controls:

OnClick	`property OnClick: TOnEvent;`
SetRadioChecked	`function SetRadioChecked: PControl;`
Click	`procedure Click;`
Checked	`property Checked: Boolean;`

5.26.19 Function NewEditBox

```
function NewEditbox( AParent: PControl; Options: TEditOptions ): PControl;
```

Creates edit box control. To create **multiline edit box**, similar to **TMemo in VCL**, apply **eoMultiline** in Options. Following properties, methods, events are special for edit controls:

OnChange	`property OnChange: TOnEvent;`
SelectAll	`procedure SelectAll;`
ReplaceSelection	`procedure ReplaceSelection( const Value: KOLString; aCanUndo: Boolean );`
DeleteLines	`procedure DeleteLines( FromLine, ToLine: Integer );`
Item2Pos	`function Item2Pos( ItemIdx: Integer ): DWORD;`
Pos2Item	`function Pos2Item( Pos: Integer ): DWORD;`
SavePosition	`function SavePosition: TEditPositions;`

© 2024 Carl Peeraer

OnChange	**property** OnChange: TOnEvent;
RestorePosition	**procedure** RestorePosition(**const** p: TEditPositions);
UpdatePosition	**procedure** UpdatePosition(**var** p: TEditPositions; FromPos, CountInsertDelChars, CountInsertDelLines: Integer);
EditTabChar	**function** EditTabChar: PControl;
CanUndo	**function** CanUndo: Boolean;
EmptyUndoBuffer	**procedure** EmptyUndoBuffer;
Undo	**function** Undo: Boolean;
Text	**property** Text: KOLString;
SelStart	**property** SelStart: Integer;
SelLength	**property** SelLength: Integer;
Selection	**property** Selection: KOLString;
Count	**property** Count: Integer;
Items	**property** Items[Idx: Integer]: KOLString;
ItemSelected	**property** ItemSelected[ItemIdx: Integer]: Boolean;
TextAlign	**property** TextAlign: TTextAlign;
Ed_Transparent	**property** Ed_Transparent: Boolean;

5.26.20 Function NewRichEdit

```
function NewRichEdit( AParent: PControl; Options: TEditOptions ): PControl;
```

Creates rich text edit control. A rich edit control is a window in which the user can enter and edit text. The text can be assigned character and paragraph formatting, and can include embedded OLE objects. Rich edit controls provide a programming interface for formatting text. However, an application must implement any user interface components necessary to make formatting operations available to the user.

Note: **eoPassword**, **eoMultiline** options have no effect for RichEdit control. Some operations are supersided with special versions of those, created especially for RichEdit, but in some cases it is necessary to use another properties and methods, specially designed for RichEdit (see methods and properties, which names are starting from RE_...).

Following properties, methods, events are special for edit controls:

OnSelChange	**property** OnSelChange: TOnEvent;
OnRE_InsOvrMode_Change	**property** OnRE_InsOvrMode_Change: TOnEvent;
OnProgress	**property** OnProgress: TOnEvent;

© 2024 Carl Peeraer

OnSelChange	**property** OnSelChange: TOnEvent;
OnRE_OverURL	**property** OnRE_OverURL: TOnEvent;
OnRE_URLClick	**property** OnRE_URLClick: TOnEvent;
BeginUpdate	**procedure** BeginUpdate;
SelectAll	**procedure** SelectAll;
ReplaceSelection	**procedure** ReplaceSelection(**const** Value: KOLString; aCanUndo: Boolean);
DeleteLines	**procedure** DeleteLines(FromLine, ToLine: Integer);
RE_TextSizePrecise	**function** RE_TextSizePrecise: Integer;
RE_FmtStandard	**function** RE_FmtStandard: PControl;
RE_CancelFmtStandard;	**procedure** RE_CancelFmtStandard;
RE_LoadFromStream	**function** RE_LoadFromStream(Stream: PStream; Length: Integer; Format: TRETextFormat; SelectionOnly: Boolean): Boolean;
RE_SaveToStream	**function** RE_SaveToStream(Stream: PStream; Format: TRETextFormat; SelectionOnly: Boolean): Boolean;
RE_LoadFromFile	**function** RE_LoadFromFile(**const** Filename: KOLString; Format: TRETextFormat; SelectionOnly: Boolean): Boolean;
RE_SaveToFile	**function** RE_SaveToFile(**const** Filename: KOLString; Format: TRETextFormat; SelectionOnly: Boolean): Boolean;
RE_HideSelection	**procedure** RE_HideSelection(aHide: Boolean);
RE_SearchText	**function** RE_SearchText(**const** Value: KOLString; MatchCase, WholeWord, ScanForward: Boolean; SearchFrom, SearchTo: Integer): Integer;
RE_WSearchText	**function** RE_WSearchText(**const** Value: KOLWideString; MatchCase, WholeWord, ScanForward: Boolean; SearchFrom, SearchTo: Integer): Integer;
RE_NoOLEDragDrop	**function** RE_NoOLEDragDrop: PControl;
CanUndo	**function** CanUndo: Boolean;
EmptyUndoBuffer	**procedure** EmptyUndoBuffer;
Undo	**function** Undo: Boolean;
FreeCharFormatRec	**procedure** FreeCharFormatRec;
SelStart	**property** SelStart: Integer;
SelLength	**property** SelLength: Integer;
Selection	**property** Selection: KOLString;
Count	**property** Count: Integer;
Items	**property** Items[Idx: Integer]: KOLString;

© 2024 Carl Peeraer

OnSelChange	**property** OnSelChange: TOnEvent;
MaxTextSize	**property** MaxTextSize: DWORD;
TextSize	**property** TextSize: Integer;
RE_TextSize	**property** RE_TextSize[Units: TRichTextSize]: Integer;
RE_CharFmtArea	**property** RE_CharFmtArea: TRichFmtArea;
RE_CharFormat	**property** RE_CharFormat: TCharFormat;
RE_Font	**property** RE_Font: PGraphicTool;
RE_FmtBold	**property** RE_FmtBold: Boolean;
RE_FmtItalic	**property** RE_FmtItalic: Boolean;
RE_FmtStrikeout	**property** RE_FmtStrikeout: Boolean;
RE_FmtUnderline	**property** RE_FmtUnderline: Boolean;
RE_FmtUnderlineStyle	**property** RE_FmtUnderlineStyle: TRichUnderline;
RE_FmtProtected	**property** RE_FmtProtected: Boolean;
RE_FmtProtectedValid	**property** RE_FmtProtectedValid: Boolean;
RE_FmtHidden	**property** RE_FmtHidden: Boolean;
RE_FmtHiddenValid	**property** RE_FmtHiddenValid: Boolean;
RE_FmtLink	**property** RE_FmtLink: Boolean;
RE_FmtFontSize	**property** RE_FmtFontSize: Integer **index**(12 **shl** 16) **or** CFM_SIZE;
RE_FmtFontSizeValid	**property** RE_FmtFontSizeValid: Boolean;
RE_FmtAutoBackColor	**property** RE_FmtAutoBackColor: Boolean;
RE_FmtFontColor	**property** RE_FmtFontColor: Integer **index**(20 **shl** 16) **or** CFM_COLOR;
RE_FmtFontColorValid	**property** RE_FmtFontColorValid: Boolean;
RE_FmtAutoColor	**property** RE_FmtAutoColor: Boolean;
RE_FmtBackColor	**property** RE_FmtBackColor: Integer **index**((64 + 32) **shl** 16) **or** CFM_BACKCOLOR;
RE_FmtFontOffset	**property** RE_FmtFontOffset: Integer **index**(16 **shl** 16) **or** CFM_OFFSET;
RE_FmtFontOffsetValid	**property** RE_FmtFontOffsetValid: Boolean;
RE_FmtFontCharset	**property** RE_FmtFontCharset: Integer **index**(25 **shl** 16) **or** CFM_CHARSET;
RE_FmtFontCharsetValid	**property** RE_FmtFontCharsetValid: Boolean;
RE_FmtFontName	**property** RE_FmtFontName: KOLString;
RE_FmtFontNameValid	**property** RE_FmtFontNameValid: Boolean;
RE_ParaFmt	**property** RE_ParaFmt: TParaFormat;
RE_TextAlign	**property** RE_TextAlign: TRichTextAlign;

© 2024 Carl Peeraer

OnSelChange	`property OnSelChange: TOnEvent;`
RE_TextAlignValid	`property RE_TextAlignValid: Boolean;`
RE_Numbering	`property RE_Numbering: Boolean;`
RE_NumStyle	`property RE_NumStyle: TRichNumbering;`
RE_NumStart	`property RE_NumStart: Integer;`
RE_NumBrackets	`property RE_NumBrackets: TRichNumBrackets;`
RE_NumTab	`property RE_NumTab: Integer;`
RE_NumberingValid	`property RE_NumberingValid: Boolean;`
RE_Level	`property RE_Level: Integer;`
RE_SpaceBefore	`property RE_SpaceBefore: Integer;`
RE_SpaceBeforeValid	`property RE_SpaceBeforeValid: Boolean;`
RE_SpaceAfter	`property RE_SpaceAfter: Integer;`
RE_SpaceAfterValid	`property RE_SpaceAfterValid: Boolean;`
RE_LineSpacing	`property RE_LineSpacing: Integer;`
RE_SpacingRule	`property RE_SpacingRule: Integer;`
RE_LineSpacingValid	`property RE_LineSpacingValid: Boolean;`
RE_Indent	`property RE_Indent: Integer index( 20 shl 16 ) or PFM_OFFSET;`
RE_IndentValid	`property RE_IndentValid: Boolean;`
RE_StartIndent	`property RE_StartIndent: Integer index( 12 shl 16 ) or PFM_STARTINDENT;`
RE_StartIndentValid	`property RE_StartIndentValid: Boolean;`
RE_RightIndent	`property RE_RightIndent: Integer index( 16 shl 16 ) or PFM_RIGHTINDENT;`
RE_RightIndentValid	`property RE_RightIndentValid: Boolean;`
RE_TabCount	`property RE_TabCount: Integer;`
RE_Tabs	`property RE_Tabs[ Idx: Integer ]: Integer;`
RE_TabsValid	`property RE_TabsValid: Boolean;`
RE_AutoKeyboard	`property RE_AutoKeyboard: Boolean;`
RE_AutoFont	`property RE_AutoFont: Boolean;`
RE_AutoFontSizeAdjust	`property RE_AutoFontSizeAdjust: Boolean;`
RE_DualFont	`property RE_DualFont: Boolean;`
RE_UIFonts	`property RE_UIFonts: Boolean;`
RE_IMECancelComplete	`property RE_IMECancelComplete: Boolean;`
RE_IMEAlwaysSendNotify	`property RE_IMEAlwaysSendNotify: Boolean;`
RE_OverwriteMode	`property RE_OverwriteMode: Boolean;`

© 2024 Carl Peeraer

OnSelChange	`property OnSelChange: TOnEvent;`
RE_DisableOverwriteChange	`property RE_DisableOverwriteChange: Boolean;`
RE_Text	`property RE_Text[ Format: TRETextFormat; SelectionOnly: Boolean ]: KOLString;`
RE_Error	`property RE_Error: Integer;`
RE_AutoURLDetect	`property RE_AutoURLDetect: Boolean;`
RE_URL	`property RE_URL: PKOLChar;`
RE_Transparent	`property RE_Transparent: Boolean;`
RE_Zoom	`property RE_Zoom: TSmallPoint;`

```
function NewRichEdit1( AParent: PControl; Options: TEditOptions ): PControl;
```

Like NewRichEdit354, but to work with older RichEdit control version 1.0 (window class 'RichEdit' forced to use instead of 'RichEdit20A', even if library RICHED20.DLL found and loaded successfully). One more difference - **OleInit** is not called, so the most of OLE capabilities of RichEdit could not working.

5.26.21 Function NewListbox

```
function NewListbox( AParent: PControl; Options: TListOptions ): PControl;
```

Creates list box control. Following properties, methods and events are special for Listbox:

OnMeasureItem	`property OnMeasureItem: TOnMeasureItem;`
OnChange	`property OnChange: TOnEvent;`
OnSelChange	`property OnSelChange: TOnEvent;`
OnDrawItem	`property OnDrawItem: TOnDrawItem;`
BeginUpdate	`procedure BeginUpdate;`
IndexOf	`function IndexOf( const S: KOLString ): Integer;`
SearchFor	`function SearchFor( const S: KOLString; StartAfter: Integer; Partial: Boolean ): Integer;`
AddDirList	`procedure AddDirList( const Filemask: KOLString; Attrs: DWORD );`
Add	`function Add( const S: KOLString ): Integer;`
Insert	`function Insert( Idx: Integer; const S: KOLString ): Integer;`
Delete	`procedure Delete( Idx: Integer );`
LBItemAtPos	`function LBItemAtPos( X, Y: Integer ): Integer;`
SelLength	`property SelLength: Integer;`
CurIndex	`property CurIndex: Integer;`

© 2024 Carl Peeraer

OnMeasureItem	**property** OnMeasureItem: TOnMeasureItem;
Count	**property** Count: Integer;
Items	**property** Items[Idx: Integer]: KOLString;
ItemSelected	**property** ItemSelected[ItemIdx: Integer]: Boolean;
ItemData	**property** ItemData[Idx: Integer]: DWORD;
LVItemHeight	**property** LVItemHeight: Integer;
LBTopIndex	**property** LBTopIndex: Integer;

5.26.22 Function NewCombobox

```
function NewCombobox( AParent: PControl; Options: TComboOptions ): PControl;
```

Creates new combo box control. Note, that it is not possible to align combobox caLeft or caRight: this can cause infinite recursion in the application.

Following properties, methods and events are special for Combobox:

OnDropDown	**property** OnDropDown: TOnEvent;
OnCloseUp	**property** OnCloseUp: TOnEvent;
OnMeasureItem	**property** OnMeasureItem: TOnMeasureItem;
OnChange	**property** OnChange: TOnEvent;
OnSelChange	**property** OnSelChange: TOnEvent;
OnDrawItem	**property** OnDrawItem: TOnDrawItem;
AddDirList	**procedure** AddDirList(**const** Filemask: KOLString; Attrs: DWORD);
Add	**function** Add(**const** S: KOLString): Integer;
Insert	**function** Insert(Idx: Integer; **const** S: KOLString): Integer;
Delete	**procedure** Delete(Idx: Integer);
CurIndex	**property** CurIndex: Integer;
Count	**property** Count: Integer;
Items	**property** Items[Idx: Integer]: KOLString;
ItemSelected	**property** ItemSelected[ItemIdx: Integer]: Boolean;
ItemData	**property** ItemData[Idx: Integer]: DWORD;
DroppedWidth	**property** DroppedWidth: Integer;
DroppedDown	**property** DroppedDown: Boolean;

© 2024 Carl Peeraer

5.26.23 Function NewListView

```
function NewListView( AParent: PControl; Style: TListViewStyle; Options: TListViewOptio
```

Creates list view control. It is very powerful control, which can partially compensate absence of grid controls (in lvsDetail view mode).

Properties, methods and events, special for list view control are:

OnMeasureItem	**property** OnMeasureItem: TOnMeasureItem;
OnEndEditLVItem	**property** OnEndEditLVItem: TOnEditLVItem;
OnLVDelete	**property** OnLVDelete: TOnDeleteLVItem;
OnDeleteLVItem	**property** OnDeleteLVItem: TOnDeleteLVItem;
OnDeleteAllLVItems	**property** OnDeleteAllLVItems: TOnEvent;
OnLVData	**property** OnLVData: TOnLVData;
OnCompareLVItems	**property** OnCompareLVItems: TOnCompareLVItems;
OnColumnClick	**property** OnColumnClick: TOnLVColumnClick;
OnLVStateChange	**property** OnLVStateChange: TOnLVStateChange;
OnDrawItem	**property** OnDrawItem: TOnDrawItem;
OnLVCustomDraw	**property** OnLVCustomDraw: TOnLVCustomDraw;
BeginUpdate	**procedure** BeginUpdate;
Delete	**procedure** Delete(Idx: Integer);
SetUnicode	**function** SetUnicode(Unicode: Boolean): PControl;
LVColAdd	**procedure** LVColAdd(**const** aText: KOLString; aalign: TTextAlign; aWidth: Integer);
LVColInsert	**procedure** LVColInsert(ColIdx: Integer; **const** aText: KOLString; aAlign: TTextAlign; aWidth: Integer);
LVColDelete	**procedure** LVColDelete(ColIdx: Integer);
LVNextItem	**function** LVNextItem(IdxPrev: Integer; Attrs: DWORD): Integer;
LVNextSelected	**function** LVNextSelected(IdxPrev: Integer): Integer;
LVAdd	**function** LVAdd(**const** aText: KOLString; ImgIdx: Integer; State: TListViewItemState; StateImgIdx, OverlayImgIdx: Integer; Data: DWORD): Integer;
LVItemAdd	**function** LVItemAdd(**const** aText: KOLString): Integer;
LVInsert	**function** LVInsert(Idx: Integer; **const** aText: KOLString; ImgIdx: Integer; State: TListViewItemState; StateImgIdx, OverlayImgIdx: Integer; Data: DWORD): Integer;

© 2024 Carl Peeraer

OnMeasureItem	**property** OnMeasureItem: TOnMeasureItem;
LVItemInsert	**function** LVItemInsert(Idx: Integer; **const** aText: KOLString): Integer;
LVDelete	**procedure** LVDelete(Idx: Integer);
LVSetItem	**procedure** LVSetItem(Idx, Col: Integer; **const** aText: KOLString; ImgIdx: Integer; State: TListViewItemState; StateImgIdx, OverlayImgIdx: Integer; Data: DWORD);
LVSelectAll	**procedure** LVSelectAll;
LVItemRect	**function** LVItemRect(Idx: Integer; Part: TGetLVItemPart): TRect;
LVSubItemRect	**function** LVSubItemRect(Idx, ColIdx: Integer): TRect;
LVItemAtPos	**function** LVItemAtPos(X, Y: Integer): Integer;
LVItemAtPosEx	**function** LVItemAtPosEx(X, Y: Integer; **var** Where: TWherePosLVItem): Integer;
LVMakeVisible	**procedure** LVMakeVisible(Item: Integer; PartiallyOK: Boolean);
LVEditItemLabel	**procedure** LVEditItemLabel(Idx: Integer);
LVSort	**procedure** LVSort;
LVSortData	**procedure** LVSortData;
LVSortColumn	**procedure** LVSortColumn(Idx: Integer);
SelLength	**property** SelLength: Integer;
Count	**property** Count: Integer;
ItemSelected	**property** ItemSelected[ItemIdx: Integer]: Boolean;
RightClick	**property** RightClick: Boolean;
ImageListSmall	**property** ImageListSmall: PImageList;
ImageListNormal	**property** ImageListNormal: PImageList;
ImageListState	**property** ImageListState: PImageList;
LVStyle	**property** LVStyle: TListViewStyle;
LVOptions	**property** LVOptions: TListViewOptions;
LVTextColor	**property** LVTextColor: TColor;
LVTextBkColor	**property** LVTextBkColor: TColor;
LVBkColor	**property** LVBkColor: TColor;
LVColCount	**property** LVColCount: Integer;
LVColWidth	**property** LVColWidth[Item: Integer]: Integer;
LVColText	**property** LVColText[Idx: Integer]: KOLString;
LVColAlign	**property** LVColAlign[Idx: Integer]: TTextAlign;

© 2024 Carl Peeraer

OnMeasureItem	**property** OnMeasureItem: TOnMeasureItem;
LVColImage	**property** LVColImage[Idx: Integer]: Integer;
LVColOrder	**property** LVColOrder[Idx: Integer]: Integer;
LVCount	**property** LVCount: Integer;
LVCurItem	**property** LVCurItem: Integer;
LVFocusItem	**property** LVFocusItem: Integer;
LVItemState	**property** LVItemState[Idx: Integer]: TListViewItemState;
LVItemStateImgIdx	**property** LVItemStateImgIdx[Idx: Integer]: Integer;
LVItemOverlayImgIdx	**property** LVItemOverlayImgIdx[Idx: Integer]: Integer;
LVItemData	**property** LVItemData[Idx: Integer]: DWORD;
LVSelCount	**property** LVSelCount: Integer;
LVItemImageIndex	**property** LVItemImageIndex[Idx: Integer]: Integer;
LVItems	**property** LVItems[Idx, Col: Integer]: KOLString;
LVItemPos	**property** LVItemPos[Idx: Integer]: TPoint;
LVTopItem	**property** LVTopItem: Integer;
LVPerPage	**property** LVPerPage: Integer;
LVItemHeight	**property** LVItemHeight: Integer;

5.26.24 Function NewTreeView

```
function NewTreeView( AParent: PControl; Options: TTreeViewOptions;
ImgListNormal, ImgListState: PImageList ): PControl;
```

Creates tree view control. See tree view methods and properties:

OnSelChange	**property** OnSelChange: TOnEvent;
OnTVBeginDrag	**property** OnTVBeginDrag: TOnTVBeginDrag;
OnTVBeginEdit	**property** OnTVBeginEdit: TOnTVBeginEdit;
OnTVEndEdit	**property** OnTVEndEdit: TOnTVEndEdit;
OnTVExpanding	**property** OnTVExpanding: TOnTVExpanding;
OnTVExpanded	**property** OnTVExpanded: TOnTVExpanded;
OnTVDelete	**property** OnTVDelete: TOnTVDelete;
OnTVSelChanging	**property** OnTVSelChanging: TOnTVSelChanging;
BeginUpdate	**procedure** BeginUpdate;
Delete	**procedure** Delete(Idx: Integer);

© 2024 Carl Peeraer

OnSelChange	**property** OnSelChange: TOnEvent;
SetUnicode	**function** SetUnicode(Unicode: Boolean): PControl;
TVInsert	**function** TVInsert(nParent, nAfter: THandle; **const** Txt: KOLString): THandle;
TVDelete	**procedure** TVDelete(Item: THandle);
TVItemPath	**function** TVItemPath(Item: THandle; Delimiter: KOLChar): KOLString;
TVItemAtPos	**function** TVItemAtPos(x, y: Integer; **var** Where: DWORD): THandle;
TVExpand	**procedure** TVExpand(Item: THandle; Flags: DWORD);
TVSort	**procedure** TVSort(N: THandle);
TVEditItem	**procedure** TVEditItem(Item: THandle);
TVStopEdit	**procedure** TVStopEdit(Cancel: Boolean);
Count	**property** Count: Integer;
ImageListNormal	**property** ImageListNormal: PImageList;
ImageListState	**property** ImageListState: PImageList;
TVSelected	**property** TVSelected: THandle;
TVDropHilighted	**property** TVDropHilighted: THandle;
TVFirstVisible	**property** TVFirstVisible: THandle;
TVIndent	**property** TVIndent: Integer;
TVVisibleCount	**property** TVVisibleCount: Integer;
TVRoot	**property** TVRoot: THandle;
TVItemChild	**property** TVItemChild[Item: THandle]: THandle;
TVItemHasChildren	**property** TVItemHasChildren[Item: THandle]: Boolean;
TVItemChildCount	**property** TVItemChildCount[Item: THandle]: Integer;
TVItemNext	**property** TVItemNext[Item: THandle]: THandle;
TVItemPrevious	**property** TVItemPrevious[Item: THandle]: THandle;
TVItemNextVisible	**property** TVItemNextVisible[Item: THandle]: THandle;
TVItemPreviousVisible	**property** TVItemPreviousVisible[Item: THandle]: THandle;
TVItemParent	**property** TVItemParent[Item: THandle]: THandle;
TVItemText	**property** TVItemText[Item: THandle]: KOLString;
TVItemRect	**property** TVItemRect[Item: THandle; TextOnly: Boolean]: TRect;
TVItemVisible	**property** TVItemVisible[Item: THandle]: Boolean;

© 2024 Carl Peeraer

OnSelChange	**property** OnSelChange: TOnEvent;
TVRightClickSelect	**property** TVRightClickSelect: Boolean;
TVEditing	**property** TVEditing: Boolean;
TVItemBold	**property** TVItemBold[Item: THandle]: Boolean;
TVItemCut	**property** TVItemCut[Item: THandle]: Boolean;
TVItemDropHighlighted	**property** TVItemDropHighlighted[Item: THandle]: Boolean;
TVItemExpanded	**property** TVItemExpanded[Item: THandle]: Boolean;
TVItemExpandedOnce	**property** TVItemExpandedOnce[Item: THandle]: Boolean;
TVItemSelected	**property** TVItemSelected[Item: THandle]: Boolean;
TVItemImage	**property** TVItemImage[Item: THandle]: Integer;
TVItemSelImg	**property** TVItemSelImg[Item: THandle]: Integer;
TVItemOverlay	**property** TVItemOverlay[Item: THandle]: Integer;
TVItemStateImg	**property** TVItemStateImg[Item: THandle]: Integer;
TVItemData	**property** TVItemData[Item: THandle]: Pointer;

5.26.25 Function NewToolbar

```
function NewToolbar( AParent: PControl; Align: TControlAlign; Options: TToolbarOptions; Bitmap: HBitmap; const Buttons: array of PKOLChar; const BtnImgIdxArray: array of Integer ): PControl;
```

Creates toolbar control. Bitmap (if present) must contain images for all buttons excluding separators (defined by string '-' in Buttons array) and system images, otherwise last buttons will no have images at all. Image width for every button is assumed to be equal to Bitmap height (if last of "squares" has insufficient width, it will not be used). To define fixed buttons, use characters '+' or '-' as a prefix for button string (even empty). To create groups of (radio-) buttons, use also '!' follow '+' or '-'. (These rules are similar used in menu creation). To define drop down button, use (as first) prefix '^'. (Do not forget to set OnTBDropDown event for this case). If You want to assign images to buttons not in the same order how these are placed in Bitmap (or You use system bitmap), define for every button (in BtnImgIdxArray array) indexes for every button (excluding separator buttons). Otherwise, it is possible to define index only for first button (e.g., [0]). It is also possible to change TBImages[ ] property for such purpose, or do the same in method TBSetBtnImgIdx).

Following properties, methods and event are specially designed to work with toolbar control:

OnDropDown	**property** OnDropDown: TOnEvent;
OnClick	**property** OnClick: TOnEvent;

© 2024 Carl Peeraer

OnDropDown	**property** OnDropDown: TOnEvent;
OnTBDropDown	**property** OnTBDropDown: TOnEvent;
OnTBClick	**property** OnTBClick: TOnEvent;
OnTBCustomDraw	**property** OnTBCustomDraw: TOnTBCustomDraw;
TBAddBitmap	**procedure** TBAddBitmap(Bitmap: HBitmap);
TBAddButtons	**function** TBAddButtons(**const** Buttons: **array of** PKOLChar; **const** BtnImgIdxArray: **array of** Integer): Integer;
TBInsertButtons	**function** TBInsertButtons(BeforeIdx: Integer; Buttons: **array of** PKOLChar; **const** BtnImgIdxArray: **array of** Integer): Integer;
TBDeleteButton	**procedure** TBDeleteButton(BtnID: Integer);
TBDeleteBtnByIdx	**procedure** TBDeleteBtnByIdx(Idx: Integer);
TBClear	**procedure** TBClear;
TBAssignEvents	**procedure** TBAssignEvents(BtnID: Integer; Events: **array of** TOnToolbarButtonClick);
TBResetImgIdx	**procedure** TBResetImgIdx(BtnID, BtnCount: Integer);
TBItem2Index	**function** TBItem2Index(BtnID: Integer): Integer;
TBIndex2Item	**function** TBIndex2Item(Idx: Integer): Integer;
TBConvertIdxArray2ID	**procedure** TBConvertIdxArray2ID(**const** IdxVars: **array of** PDWORD);
TBButtonSeparator	**function** TBButtonSeparator(BtnID: Integer): Boolean;
TBButtonAtPos	**function** TBButtonAtPos(X, Y: Integer): Integer;
TBBtnIdxAtPos	**function** TBBtnIdxAtPos(X, Y: Integer): Integer;
TBMoveBtn	**function** TBMoveBtn(FromIdx, ToIdx: Integer): Boolean;
TBSetTooltips	**procedure** TBSetTooltips(BtnID1st: Integer; **const** Tooltips: **array of** PKOLChar);
TBBtnTooltip	**function** TBBtnTooltip(BtnID: Integer): KOLString;
CurIndex	**property** CurIndex: Integer;
Count	**property** Count: Integer;
RightClick	**property** RightClick: Boolean;
TBCurItem	**property** TBCurItem: Integer;
TBButtonCount	**property** TBButtonCount: Integer;
TBBtnImgWidth	**property** TBBtnImgWidth: Integer;
TBButtonEnabled	**property** TBButtonEnabled[BtnID: Integer]: Boolean;
TBButtonVisible	**property** TBButtonVisible[BtnID: Integer]: Boolean;

© 2024 Carl Peeraer

OnDropDown	**property** OnDropDown: TOnEvent;
TBButtonChecked	**property** TBButtonChecked[BtnID: Integer]: Boolean;
TBButtonMarked	**property** TBButtonMarked[BtnID: Integer]: Boolean;
TBButtonPressed	**property** TBButtonPressed[BtnID: Integer]: Boolean;
TBButtonText	**property** TBButtonText[BtnID: Integer]: KOLString;
TBButtonImage	**property** TBButtonImage[BtnID: Integer]: Integer;
TBButtonRect	**property** TBButtonRect[BtnID: Integer]: TRect;
TBButtonWidth	**property** TBButtonWidth[BtnID: Integer]: Integer;
TBButtonLParam	**property** TBButtonLParam[**const** Idx: Integer]: DWORD;
TBButtonsMinWidth	**property** TBButtonsMinWidth: Integer;
TBButtonsMaxWidth	**property** TBButtonsMaxWidth: Integer;
TBRows	**property** TBRows: Integer;

5.26.26 Function NewTabControl

```
function NewTabControl( AParent: PControl; const Tabs: array of PKOLChar; Options: TTabControlOptions; ImgList: PImageList; ImgList1stIdx: Integer ): PControl;
```

Creates new tab control (like notebook).

```
function NewTabEmpty( AParent: PControl; Options: TTabControlOptions; ImgList: PImageList ): PControl;
```

Creates new empty tab control for using metods TC_Insert (to create Pages as Panel), or TC_InsertControl (if you want using your custom Pages).

To place child control on a certain page of TabControl, use property Pages[Idx], for example:
Label1 := **NewLabel** 346(TabControl1.Pages[0], 'Label1');

To determine number of pages at run time, use property **Count** ;
to determine which page is currently selected (or to change selection), use property **CurIndex** ;
to feedback to switch between tabs assign your handler to **OnSelChange** event;
Note, that by default, tab control is created with a border lowered to tab control's parent. To remove it, you can apply WS_EX_TRANSPARENT extended style (see **TControl.ExStyle** property), but painting of some child controls can be strange a bit in this case (no border drawing for edit controls was found, but not always...). You can also apply style WS_THICKFRAME (**TControl.Style** property) to make the border raised.

Other methods and properties, suitable for tab control, are:

© 2024 Carl Peeraer

OnChange	**property** OnChange: TOnEvent;
IndexOf	**function** IndexOf(**const** S: KOLString): Integer;
SearchFor	**function** SearchFor(**const** S: KOLString; StartAfter: Integer; **Partial**: Boolean): Integer;
SetUnicode	**function** SetUnicode(Unicode: Boolean): PControl;
TC_Insert	**function** TC_Insert(Idx: Integer; **const** TabText: KOLString; TabImgIdx: Integer): PControl;
TC_Delete	**procedure** TC_Delete(Idx: Integer);
TC_InsertControl	**procedure** TC_InsertControl(Idx: Integer; **const** TabText: KOLString; TabImgIdx: Integer; Page: PControl);
TC_Remove	**function** TC_Remove(Idx: Integer): PControl;
TC_SetPadding	**procedure** TC_SetPadding(cx, cy: Integer);
TC_TabAtPos	**function** TC_TabAtPos(x, y: Integer): Integer;
TC_DisplayRect	**function** TC_DisplayRect: TRect;
TC_IndexOf	**function** TC_IndexOf(**const** S: KOLString): Integer;
TC_SearchFor	**function** TC_SearchFor(**const** S: KOLString; StartAfter: Integer; **Partial**: Boolean): Integer;
ImageListNormal	**property** ImageListNormal: PImageList;
Pages	**property** Pages[Idx: Integer]: PControl;
TC_Pages	**property** TC_Pages[Idx: Integer]: PControl;
TC_Items	**property** TC_Items[Idx: Integer]: KOLString;
TC_Images	**property** TC_Images[Idx: Integer]: Integer;
TC_ItemRect	**property** TC_ItemRect[Idx: Integer]: TRect;

5.26.27 Function NewForm

```
function NewForm( AParent: PControl; const Caption: KOLString ): PControl;
```

Creates form window object and returns pointer to it. If You use only one form, and You are not going to do applet button on task bar invisible, it is not necessary to create also special applet button window - just pass your (main) form object to **Run** procedure. In that case, it is a good idea to assign pointer to your main form object to **Applet** 369 variable immediately following creating it - because some objects (e.g. TTimer) want to have **Applet** 369 assigned to something.

Following methods, properties and events are useful to work with forms (ones common for all visual objects, such as **Left**, **Top**, **Width**, **Height**, etc. are not listed here - look **TControl** for it):

OnMessage	**property** OnMessage: TOnMessage;
OnClose	**property** OnClose: TOnEventAccept;

© 2024 Carl Peeraer

OnMessage	**property** OnMessage: TOnMessage;
OnQueryEndSession	**property** OnQueryEndSession: TOnEventAccept;
OnMinimize	**property** OnMinimize: TOnEvent;
OnMaximize	**property** OnMaximize: TOnEvent;
OnRestore	**property** OnRestore: TOnEvent;
OnFormClick	**property** OnFormClick: TOnEvent;
ParentForm	**function** ParentForm: PControl;
FormParentForm	**function** FormParentForm: PControl;
CreateWindow	**function** CreateWindow: Boolean; **virtual**;
Close	**procedure** Close;
IconLoad	**procedure** IconLoad(Inst: Integer; ResName: PKOLChar);
IconLoadCursor	**procedure** IconLoadCursor(Inst: Integer; ResName: PKOLChar);
Show	**procedure** Show;
ShowModal	**function** ShowModal: Integer;
Hide	**procedure** Hide;
MinimizeNormalAnimated	**procedure** MinimizeNormalAnimated;
RestoreNormalMaximized	**procedure** RestoreNormalMaximized;
IsMainWindow	**function** IsMainWindow: Boolean;
GotoControl	**procedure** GotoControl(Key: DWORD);
RemoveStatus	**procedure** RemoveStatus;
StatusPanelCount	**function** StatusPanelCount: Integer;
CenterOnCurrentScreen	**function** CenterOnCurrentScreen: PControl;
Icon	**property** Icon: HIcon;
Caption	**property** Caption: KOLString;
ModalResult	**property** ModalResult: Integer;
Modal	**property** Modal: Boolean;
ModalForm	**property** ModalForm: PControl;
WindowState	**property** WindowState: TWindowState;
HasBorder	**property** HasBorder: Boolean;
HasCaption	**property** HasCaption: Boolean;
CanResize	**property** CanResize: Boolean;
StayOnTop	**property** StayOnTop: Boolean;
Border	**property** Border: ShortInt;

© 2024 Carl Peeraer

OnMessage	**property** OnMessage: TOnMessage;
Margin	**property** Margin: ShortInt;
AlphaBlend	**property** AlphaBlend: Byte;
StatusText	**property** StatusText[Idx: Integer]: KOLString;
SimpleStatusText	**property** SimpleStatusText: KOLString;
StatusPanelRightX	**property** StatusPanelRightX[Idx: Integer]: Integer;
StatusWindow	**property** StatusWindow: HWND;

5.26.28 Function NewApplet

function NewApplet(const Caption: KOLString): **PControl**;
Creates applet button window, which has to be parent of all other forms in your project (but this is *not must*). See also comments about **NewForm** 367.

Following methods, properties and events are useful to work with applet control:

Run	**procedure** Run(**var** AppletCtl: PControl);
OnMessage	**property** OnMessage: TOnMessage;
Close	**procedure** Close;
IconLoad	**procedure** IconLoad(Inst: Integer; ResName: PKOLChar);
IconLoadCursor	**procedure** IconLoadCursor(Inst: Integer; ResName: PKOLChar);
Show	**procedure** Show;
Hide	**procedure** Hide;
IsMainWindow	**function** IsMainWindow: Boolean;
ProcessMessage	**function** ProcessMessage: Boolean;
ProcessMessages	**procedure** ProcessMessages;
ProcessPendingMessages	**procedure** ProcessPendingMessages;
Icon	**property** Icon: HIcon;
Caption	**property** Caption: KOLString;
ModalForm	**property** ModalForm: PControl;

© 2024 Carl Peeraer

5.26.29 Function NewMDIClient

```
function NewMDIClient( AParent: PControl; WindowMenu: THandle ): PControl;
```

Creates MDI client window, which is a special type of child window, containing all MDI child windows, created calling NewMDIChild370 function. On a form, MDI client behaves like a panel, so it can be placed and sized (or aligned) like any other controls. To minimize flick during resizing main form having another aligned controls, place MDI client window on a panel and align it caClient in the panel.
Note: MDI client must be a single on the form.

5.26.30 Function NewMDIChild

```
function NewMDIChild( AParent: PControl; const ACaption: KOLString ):
PControl;
```

Creates MDI client window. AParent should be a MDI client window, created with NewMDIClient370 function.

5.26.31 Function NewDateTimePicker

```
function NewDateTimePicker( AParent: PControl; Options:
TDateTimePickerOptions ): PControl;
```

Creates date and time picker common control.

© 2024 Carl Peeraer

Graphic Visual Elements

This chapter will briefly discuss graphical, i.e. windowless counterparts of some window objects that do not have window descriptors.

6 Graphic Visual Elements

Graphic (Not Windowed) Visual Elements

This chapter will briefly discuss graphical, i.e. **windowless counterparts** of some window objects that do not have window descriptors. Surprising as it may seem, there is a much wider range of such objects in KOL than in VCL. In addition to the label and drawer, the KOL library provides graphical counterparts for buttons, check boxes and radio boxes, and even for a single-line edit box.

As you know, the main purpose of graphic controls is to reduce the load on the system in the case when the form contains a very large number of visual elements. For example, if you try to implement the game "Miner" for a field of 50x50 cells, depicting cells with ordinary buttons, then you should expect a sharp drop in the performance of even a very powerful computer when you run such an application. (Of course, nobody does that, this example is rather abstract).

Of course, it is always possible in each individual application to independently draw the necessary elements, which do not necessarily repeat the appearance of the simulated visual objects, the same buttons or checkboxes. Sometimes you can go the route of changing the interface to use lists or trees, if this is appropriate for the conditions of the problem being solved. And yet there are still cases when none of these approaches suits, and it is precisely a certain set of analogs of standard visual objects that is required.

This is what graphic visual objects are for. Unlike their windowed prototypes, graphical counterparts do not require a window handle (and do not have their own window at all). They draw themselves, without the participation of the operating system, in the procedure for drawing their window parent, which acts as an underlay.

In particular, this means that graphical visual objects provide a richer set of possibilities for modifying their appearance on the part of the developer than their window counterparts. For example, a regular button in Windows does not allow you to change its color or the font color of its label, and the same applies to checkboxes, which, in fact, are also buttons. [I don't know, honestly, why such a restriction is needed. This leads to the fact that the programmer people, trying to make their interface a little less gray and monotonous, instead of standard buttons start using self-colored buttons, and then it all starts to look worse when XP themes are turned on. In any case, this question should be asked by the developers of the Windows operating system].

For a graphic button, such a restriction on colors is not relevant. In addition, graphics controls in KOL have a number of additional event handlers that allow you to intervene in the drawing process at any stage of the redrawing process in order to provide the desired appearance. Namely, in addition to OnPaint, which allows you to replace the entire main drawing procedure, and OnEraseBkgnd, which replaces the background erasing procedure, there are also OnPrePaint and OnPostPaint. These handlers allow you to fix something before starting the main drawing (for example, change the font style, or draw some parts of the image and exclude them from the drawing area), or fix something (do overdrawing) after the main drawing is done.

© 2024 Carl Peeraer

When developing graphic visual objects for the KOL library, it is possible to correlate them with **XP themes**. Moreover, in order for these objects to look "like real", it is enough to include one or two more conditional compilation symbols (depending on how much you agree to increase the size of the application, or keep the size small, at the expense of some of the believability).
In the case of using MCK, in order for the window object to be replaced by a graphical one, it is enough to set the Windowed property to false. In this case, however, you should not forget to add the conditional compilation symbol **USE_GRAPHCTLS** to the project properties so that all the necessary declarations become available to the compiler (the addition of this symbol itself, even without the actual use of graphic controls, increases the size of the application by a hundred bytes).
For manual programming in KOL, you should create graphical interface elements specifically for this purpose with dedicated design functions. The addition of the **USE_GRAPHCTLS** symbol is, of course, mandatory in this case too.

- Graphic Label 373
- Graphic Canvas for Drawing 374
- Graphic Button 374
- Graphic Flags 375
- Graphic Input Field 375
- XP Themes 376

6.1 Graphic Label

The first (but not the last) graphic visual is the label, i.e. a field containing some text to be displayed on the form. Graphic labels constructors:

```
NewGraphLabel(Parent, Caption)
NewGraphWordWrapLabel(Parent, Caption)
```
374 374

The object created in this constructor is the same PControl as other visual objects. You can still change its position, size, set alignment. But the properties and methods that exploit the window handle turn out to be inapplicable. For example, many event handlers like **OnMessage cannot be used for it**. OnClick, **OnMouseXXXX** (as well as OnKeyXXXX and OnChar - for "focused" graphic controls, see below) remain guaranteed to be available. These events are specially simulated in the window parent's message handler so that the basic behavior of the graphics controls remains the same.

© 2024 Carl Peeraer

Syntax

```
function NewGraphLabel( AParent: PControl; const ACaption: AnsiString ):
PControl;
```
Creates graphic label, which does not require a window handle.

```
function NewWordWrapGraphLabel( AParent: PControl; const ACaption:
KOLString ): PControl;
```
Creates graphic label, which does not require a window handle.

6.2 Graphic Canvas for Drawing

Unlike its window counterpart (paintbox), a graphics box with pictures does not have its own window handle, and uses the parent window handle, like all other graphic visual elements. In particular, the graphic “artist's box” can no longer function as a regular panel and become a parent for other visual elements. Actually, there are no other differences, if we consider its basic functionality. The functionality of this element is all concentrated in the presence of the **OnPaint** event, and it works in much the same way as for the window twin.

6.3 Graphic Button

But there is no such analogue of the button in the VCL anymore. In KOL, a graphical button exactly repeats the appearance and functionality of a regular button (and even slightly poorer, considering that a window button in KOL can contain an icon instead of text). This is not an analogue of **TBitbtn** from the VCL (which also holds its own window handle), and not an analogue of **TSpeedButton** (since it can capture input focus), namely, an analogue of **TButton**, but without a window handle. In particular, this means that you can already do "Miner" on the graphic buttons without fear of the consequences associated with poor performance or even a crash of the operating system and applications running in parallel in the case of a very large number of such buttons.

As you know, the main functionality of a button is its ability to be "pressed" by a manipulator of the "mouse" type and to call the **OnClick** handler associated with this event, if any. In addition, the button must be able to be in focus (changing its appearance slightly, usually by adding a dotted border inside the button), and then such a visual button can be "pressed" from the keyboard. This is probably why the graphical analogue of the button was not created in the VCL, because there is a problem with transferring focus to an object that cannot have focus by definition, simply because this object does not have a window handle. In KOL, this problem is solved by simulating focus: the focus actually belongs to the parent window object, but this object "knows"

Graphic button constructor:
NewGraphButton(Parent, Caption) 375

© 2024 Carl Peeraer

Syntax

```
function NewGraphButton( AParent: PControl; const ACaption: KOLString ):
PControl;
```

Creates graphic button.

6.4 Graphic Flags

On Windows, check boxes (check boxes and radio boxes) are flavors of a button. This is not surprising since their basic functionality is about the same, namely the ability to be pressed with a mouse or keyboard. There are also graphical analogs for these objects in KOL. And in their graphical incarnation, it is these objects that turn out to be most useful for facilitating the descriptor "weight" of all kinds of configuration dialogs, in which there can be hundreds of flags.

Constructors:
NewGraphCheckBox(Parent, Caption) 375
NewGraphRadioBox(Parent, Caption) 375

To catch a click, these objects also use the **OnClick** event, and the "**checked**" state is read and written through the **Checked** properties and the **SetRadioChecked** method, just like for their window prototypes.

Syntax

```
function NewGraphCheckBox( AParent: PControl; const ACaption: KOLString ):
PControl;
```

Creates graphic checkbox.

```
function NewGraphRadioBox( AParent: PControl; const ACaption: KOLString ):
PControl;
```

Creates graphic radiobox.

6.5 Graphic Input Field

In fact, the one-line input field is the last visual object for which it would have made any sense to create a graphical counterpart. It makes no sense to move further, since the increase in the code will outweigh all the benefits of the absence of its own window descriptor.

But already when implementing a graphical input field, a simple trick was used to avoid duplicating the functionality of the input field itself in your code. Namely: when the focus is transferred to this element, a temporary (until the focus is lost), but "real" window object for line editing, which visually practically does not differ, is created for it, has the same borders, and allows you to edit the text as usual. The substitution process itself occurs in a completely transparent way for the user (and for the application), and is visually invisible.

© 2024 Carl Peeraer

The benefits of using a graphical input field are the same as those of other types of graphical elements. It should be noted that a form with fifty such fields, if they do not have window descriptors, works much faster, which is noticeable even on a very high-speed hardware configuration.

Constructor:
NewGraphEditBox(Parent, Options) 376

Syntax

```
function NewGraphEditbox( AParent: PControl; Options: TEditOptions ):
PControl;
```

Creates graphic edit box. To do editing, this box should be replaced with real edit box with a handle (actually, it is enough to place an edit box on the same Parent having the same BoundsRect).

6.6 XP Themes

XP Themes for Graphic Controls and more...

If the application is being developed with **XP themes** in mind, i.e. it is supposed to use a manifest that radically changes the appearance of the application when it is running in the operating system XP or Vista, then KOL provides the ability to draw graphical elements "to the theme". But by default, to make the code easier, this feature is disabled.

Initially, the conditional compilation symbol **GRAPHCTL_XPSTYLES** was introduced in order for graphical visuals to portray themselves in accordance with current **XP themes**. As a result, code was added to the final application that was responsible for rendering all the above graphic controls in accordance with the current theme. At the same time, if the user turned off the desktop themes, the drawing was performed with the same algorithm.

Later, the effect of the **GRAPHCTL_XP_STYLES** symbol was significantly extended, thanks to the MTsv DN (this is the developer's alias). Now, when you enable it in the project options, a rather considerable amount of code (about 10 KB) from the **visual_xp_styles.inc** file is connected to the application, which is responsible for the correct display of ordinary controls, including for the group box - with enabled themes, as well as when switching themes ...

But this may not be enough to completely match the appearance. The fact is that as soon as themes are enabled, the appearance of the controls of all applications that support the manifest begins to change dynamically when the mouse cursor moves over them. In order for this behavior to be reflected for graphic controls, you should add one more conditional compilation symbol: **GRAPHCTL_HOTTRACK**.

© 2024 Carl Peeraer

Non-Visual Objects

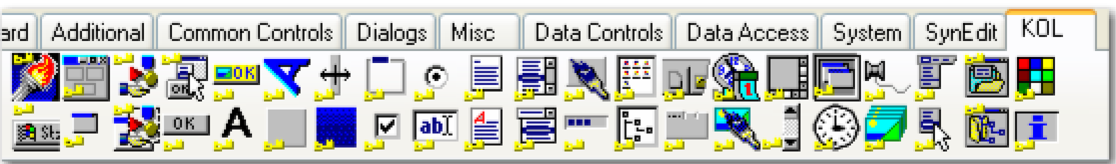
Menus, Dialogs for choosing a file or folder name, Clocks, Execution Threads

7 Non-Visual Objects

Non-Visual Objects KOL and MCK

This concludes the description of the main visual objects of KOL, which are window and pseudo-window objects (i.e. controls). Those "controls" that have already been described are already quite enough to implement a decent enough visual interface. But for a full-fledged application development, a certain set of auxiliary objects is still lacking for solving a number of common tasks, such as: menus, dialogs for choosing a file or folder name, clocks, execution threads.

All such objects are inherited directly from TObj, i.e. are simple objects. But, at the same time, for all of them there are mirror components in the Mirror Classes Kit, allowing them to be used in visual design. That is, they can be thrown onto the form and set up the desired properties and events. As a result, code is automatically generated that ensures the creation of the corresponding objects along with the form, and their destruction along with the destruction of the form object.

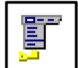
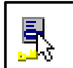


- Menu (TMenu) 379
  - Events for the entire menu or its child items 381
  - Events, methods, properties of an individual menu item as an object 382
  - Access to properties of subordinate menu items 383
  - Main menu 383
  - Pop-up menu 384
  - Accelerators 385
  - Menu at MCK 385
  - Menu - Syntax 386
- Tray Icon (TTrayIcon) 394
  - Tray Icon - Syntax 395
- File Selection Dialog (TopenSaveDialog) 397
  - File Selection Dialog - Syntax 399
- Directory Selection Dialog (TOpenDirDialog) 401
  - Directory Selection Dialog - Syntax 403
- Alternative Directory Selection Dialog (TOpenDirDialogEX) 404
  - Alternative Directory Selection Dialog - Syntax 407
- Color Selection Dialog (TColorDialog) 409
  - Color Selection Dialog - Syntax 410
- Clock (TTimer) 411
  - Multimedia Timer (TMMTimer) 413
  - Clock - Syntax 414
- Thread, or thread of commands (TThread) 416

© 2024 Carl Peeraer

  - Thread - Syntax 419
- Action and ActionList 424
  - Action and ActionList - Syntax 425

7.1 Menu (TMenu)

First on the list of non-visual objects, I decided to describe the menu. Although the "non-visibility" of this object is quite possible to question. The main menu is a completely visual element of the form, although it does not have its own window descriptor (of course, the menu has descriptors, but they are not window descriptors).

Moreover, the main menu is displayed on the form, including at the design stage, allowing you to create event handlers (and quickly jump into their code) when you select the appropriate item in the menu. Note: however, unlike VCL, to create new main menu items, or to move items, in MCK you should use the menu editor, which is invoked by double clicking on the mirrored component.

In the KOL library, the main menu, the popup menu, and the menu items are all implemented in one TMenu object, keeping the tradition of saving on a variety of objects. The menu builder uses a list-of-strings-based templating technique, similar to the one used for the toolbar. Historically, the menu was developed earlier than the line of buttons, and this method of economical construction of multi-element objects was used for the first time precisely to build a menu tree. Digging even deeper, this object first appeared in XCL, the predecessor of KOL, and little has changed since then.

When you create the first menu and assign it to a form, this menu object is automatically made the main menu for the form (and if it has at least one item, it is displayed at the top of the form). All subsequent menu objects added to the form become popups and are not displayed until either the Popup or PopupEx method is called programmatically or automatically (SetAutoPopupMenu) for them. In particular, if the form should not have a main menu, but it has one or more pop-up menus, then the first step is to add a dummy main menu that does not have displayed elements.

 © 2024 Carl Peeraer

Constructors:

`NewMenu(Parent, dummy, template, onmenu)` 387 - constructs a menu based on the specified template, adding it to the form specified by the Parent parameter;
`NewMenuEx(Parent, dummy, template, onitems_array)` 387 - similar to the previous function, but allows you to assign your own event handlers for all or part of the menu items at once.
The dummy parameter was retained for compatibility with the first versions of KOL when it was used and named maxcmdreserve (and then firstcmd). Since the menu for each item began to create its own instance of the TMenu object, the need for this parameter has disappeared.

The rules for constructing a template template should be discussed in more detail. Template is an array of strings (of type PChar) that define for menu items:

- their appearance (text, mnemonics, accelerators);
- mode of operation (normal, separator, switchable, groupable radio switch, initial state of the switch).

To specify all these features, prefix characters in strings are used:

'&'	The character before the letter or number that becomes the mnemonic of the menu item. The mnemonic is displayed with an underline (in newer versions of the OS, by default, the underline is shown only when the Alt key is pressed), and allows you to invoke a menu from the keyboard. Let me remind you that in order for the KOL application to be able to use mnemonics in the menu without first activating the menu itself, you must provide a call to the SupportMnemonics method;
'+'	The menu item is "marked" with a special checkmark. For radio-toggled menu items, a circle is used instead of a checkmark. If such a prefix is specified before the text of an element, then the element becomes automatically switchable. In this case, when a user selects a menu item, it automatically changes its state from "checked" to "not checked". If the prefix '+' (or '-', see below) is followed by a '!', Then the menu item is radio switchable. The system combines several consecutive radio switches into one group automatically (there should not be other types of menu items between them);
'-'	The menu item is "unchecked", but otherwise everything said about the '+' prefix is also true for '-';
'-'	If the text of an element consists of a single minus, then the element is a separator (and is shown in the menu as a narrow line between groups of regular menu elements). The separator is always not selectable, and therefore it doesn't make sense to assign an OnMenu event handler to it;
'('	Starts a submenu subordinate to the previous menu item;
')'	Ends the submenu, returning to the previous nesting level.

In the NewMenuEx constructor, as well as in the AssignEvents method, when events from an array parameter are assigned to menu items, separators and template elements '(' and ')' are not taken into account (skipped).

Properties, methods and events of the TMenu object:

Handle - menu descriptor. The menu itself and any of its elements (including separators) have such a descriptor. This descriptor is a number (of the hMenu type) known to the system, and allows you to call API functions to perform some kind of action on the menu at a low level. For example, a menu, along with all its submenus, can be passed as a parameter to the TrackPopupMenu and TrackPopupMenuEx functions to "pop up" the menu with any additional display styles;
SubMenu - descriptor of the subordinate menu. In fact, equivalent to Handle;
MenuID - internal numerical "identifier" of the menu item, assigned by the KOL code. Because the number of available identifiers cannot exceed 65535, and these identifiers cannot be reused, then you should not constantly create and delete menu items too often during long-term work. In particular, you should not make them "hidden", because hiding and showing menu items is implemented exactly as destruction and creation of new items (in Windows there is no way to hide menu items in a different way). Instead, it is recommended to use the ability to make menu items unavailable or available (enabled) as needed;
Insert(i, s, event, options) - adds a new menu item (creating another TMenu object corresponding to the newly created menu item, and returning a PMenu pointer to this object;
Parent - parent menu (if available);
TopParent - top-level parent menu;
Owner - an object of type TControl to which this menu belongs (must be a form);
Items[i]- subordinate menu items (including nested menu items, including separators). The i parameter can be the absolute index of the nested element with a value between 0 and 4096, or a numeric ID descriptor. A value of -1 returns the menu item itself (itself);
Count - the number of subordinate menu items, including recursively nested items;
IndexOf(s) - returns the index of the subordinate menu item (including nested items of any level), searching for it by the text s. A value of -1 is returned for the menu item itself, and -2 if no such menu item is found;
InsertSubMenu(submenu, i) - allows you to add a previously prepared menu as an element of this menu, along with all its subordinate elements;
RemoveSubmenu(i) - detaches a subordinate menu added, for example, by the InsertSubMenu method;
AddItem(s, event, options) - adds a menu item to the end of the list;
InsertItem(i, s, event, options) - inserts a menu item at the specified position;
AssignBitmaps(i, bitmaps) - allows you to assign several bitmaps at once to menu items, starting with a given one;

7.1.1 Events for the entire menu or its child items

OnMenuItem - an event that is triggered for a menu object when an element is selected in it. When a menu item is selected, such an event (if assigned) is triggered for the menu item itself, and for each parent menu. This approach allows, if desired, to save on the creation of separate event handlers for each menu item, and to concentrate all the processing of clicks on the menu in one procedure, assigning it as a handler for clicks on the parent menu of the highest level;

© 2024 Carl Peeraer

ByAccel - this property can be interrogated in the menu handler to determine whether the menu item was "clicked" by the coordinate device (mouse or its substitute), or selected using a shortcut key. Note: accelerator is an accelerator, accelerators should not be confused with mnemonics, they are different mechanisms;
IsSeparator - returns true if item is a separator;
OnUncheckRadioItem - this event allows you to assign an additional event handler "the current element of the radio group is no longer current". The aforementioned OnMenuItem event fires only for the radio group item in the menu that has been selected;
AssignEvents(i, events) - allows you to assign event handlers for several menu items, starting with i;

7.1.2 Events, methods, properties of an individual menu item as an object

The events, methods and properties listed below relate primarily to each individual menu item without affecting the entire menu tree or subordinate items of this menu item. To access such properties from code, you must have a pointer to the object corresponding to the menu item. For example, such a pointer can be obtained for the main menu using the **Items [i]** property. If the composition of the menu changes dynamically, the best way is immediately after the initial creation of the menu and before performing any modifications to the menu (i.e. when its index is precisely known for each menu item) to copy the pointers of those menu items to which the program code contains calls to their variables (like **PMenu**).

OnMeasureItem - an event that is called for a menu item with the OwnerDrawFixed option to set the size of the menu (in the lower word of the result, the handler must return the height, in the upper word - the width of the menu item);
OnDrawItem - an event for drawing a menu item by a handler assigned by the programmer. The menu must have the **OwnerDraw** property equal to true;
OwnerDraw - the **OnDrawItem** event handler is called for displaying menu items;
Caption - Menu item caption text (including '&' indicating mnemonic characters, and keyboard accelerator representation string, usually following tabulation character).
MenuBreak - the type of separation of this menu item from subsequent ones (for automatic transfer of menu items to the next line or column);
RadioGroup - radio group index. Several consecutive menu items with the same RadioGroup property value form a single group of toggle items, in which only one item can be "checked";
IsCheckItem - the menu item is automatically tagged. By choosing such an item in the menu, the user automatically changes his states "marked" - "not marked" to the opposite (before the **OnMenuItem** event is triggered);
Checked - the menu item corresponding to the object is "marked";
Enabled - the object menu item is allowed (if not, then the menu item becomes pale and unavailable for selection by the user, i.e. the OnMenuItem event will never occur for it while it is in this state);
DefaultItem - the menu item is the "default" item, i.e. it is visually highlighted (in bold) and is triggered by pressing the <Enter> key when the parent menu is displayed on the screen along with its children;
Highlight - the menu item is highlighted;

© 2024 Carl Peeraer

Visible - property of the object corresponding to the menu item. The object makes the given menu item "visible" by destroying the menu item when this property is set to false and re-creating it when the property is set to true again;
Data - a pointer that allows you to associate some additional data with a menu item (including, it can be any 32-bit number);
Bitmap - the hBitmap bitmap used to display the icon to the left of the text in the menu (in the same place where the system displays a "bird" or a marking circle for "marked" menu items);
BitmapItem - the bitmap hBitmap, rendered in place of the menu text, if assigned. There are a number of reserved system constants that can optionally be used as a value for this property. For example, HBMMENU_CALLBACK - allows you to organize the substitution of the required image by an additional request from the system, as well as: HBMMENU_MBAR_CLOSE, HBMMENU_MBAR_MINIMIZE, etc .;
Accelerator - "accelerator", or a keyboard shortcut that can be used to invoke a menu item. The accelerator is created in code by calling the **MakeAccelerator** function;
HelpContext - a number that is used in the embedded help system of the application to identify the article in the help, which is activated when requesting contextual help for the menu item (F1);

7.1.3 Access to properties of subordinate menu items

Access to properties of subordinate menu items (by index or numeric identifier)

All the properties of this group are just another (equivalent) way to access the properties of individual menu items.

GetMenuItemHandle(i) and **ItemHandle [i]** - returns the numeric identifier of the menu item (see MenuID);
ItemChecked[i]- the menu item "marked". Do not use this property to "check" a radio group item to toggle menu items, use the **RadioCheck** property for that;
ItemEnabled[i] - the menu item is available;
ItemVisible[i]- the menu item is visible. See the note on the Visible property - everything said for it is the same for this property, because it is actually another way to refer to this property;
RadioCheck(i) - makes the menu item included in the group of radio-switchable menu items "marked", while removing the marking from all other items in this group;
ItemBitmap[i] - a bitmap to display to the left of the text in a menu item;
ItemHelpContext[i] - the context of the help system;
ItemAccelerator[i] - accelerator (keyboard shortcut) for quick access to a menu item;
ItemSubMenu[i] - descriptor of the parent menu item;

7.1.4 Main menu

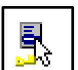
RedrawFormMenuBar - for the main menu, this call provides an update of the image of the main menu bar after making any modifications in it. If such a call is not made, then the menu itself is not updated.

Sometimes the main menu uses a special visual effect of aligning one or more of the last top-level items of the main menu to the right. To achieve this effect, just run the following code:

© 2024 Carl Peeraer

```
i: = MainMenu1.ItemHandle [mmAbout];
ModifyMenu (MainMenu1.Handle, i,
MF_BYCOMMAND or mf_Help,
i,
PChar (MainMenu1.ItemText [mmAbout]));
```

7.1.5 Pop-up menu



As already noted, pop-up menus in KOL are no different from the main one (except for the order in which they are created). But in order to display them, they must be ordered to "pop up", or by using the **SetAutoPopupMenu** method of any visual object on the form, an additional handler to the window that will execute such an order when the user performs certain actions (pressing a special button on the keyboard or the right mouse button) ...

Popup(X, Y) - makes a popup menu appear at the specified coordinates on the screen;
PopupEx(X, Y) - similar to the previous method, but behaves in a special way if the window of the parent form is invisible on the screen at that moment. Namely, it makes it visible (Visible = true), but takes it out of the screen for a while, effectively leaving it invisible to the user. The point of this "forgery" is to ensure that the pop-up menu is automatically hidden correctly when it "loses" focus, more precisely, when any other window is in focus. It is advisable to use this method when organizing the pop-up menu on the "tray icon" (see TTrayIcon), otherwise the pop-up menu does not "guess" in any way that it is time to hide if the user clicks past the menu. (Usually, the user does this if he decided that he does not want to select any of the items of such a menu, and wants to do something else, and here is the menu,

Flags - allows you to define a set of flags that will be used in the **Popup** and **PopupEx** methods as a parameter for the TrackPopupMenuEx API function. With the help of these flags it is possible to change such parameters as alignment and placement on the screen, permission to click on menu items with the right mouse button, animation method;
OnPopup - this event is triggered immediately before displaying a pop-up menu, or before expanding child menu items (when the mouse is hovering or moving the cursor to a menu item that has child items). In the handler of this event, it is allowed, inter alia, to change the composition, availability or some other states of individual subordinate items, depending on any external conditions. Among other things, it is possible to set the value of the NotPopup property to true, thereby preventing the "popup";
NotPopup - setting this property to true prevents the popup menu from popping up, or expanding the list of subordinate menu items;
CurCtl - a pointer to a window object of the **PControl** type, which initiated the automatic popup of the menu (since its handler of the corresponding window message, attached to it by calling **SetAutoPopupMenu**, triggered).

© 2024 Carl Peeraer

7.1.6 Accelerators

To conclude our discussion of menus in KOL, we need to supplement information on accelerators (keyboard shortcuts) that are used to invoke menu items from the keyboard. These keyboard shortcuts are automatically displayed to the right of the text in the menu when the Accelerator or **ItemAccelerator [i]** property is used to assign them. In fact, their text is simply added to the text of the menu through the tabulator character (and the system already provides the alignment of such additional columns after the tabulation character # 9). Note: if you want to place a non-standard text designation of the accelerator to the right of the menu text in this position, or list several keyboard shortcuts that cause the same action, you just need to change the menu text yourself using this menu feature.

Accelerator, i.e. a keyboard shortcut can be generated by calling the global function **MakeAccelerator (Virt, Key)**, where Virt is a combination of the FSHIFT, FCONTROL, FALT, FVIRTKEY, FNOINVERT flags, and Key is a character or virtual key code.

You can get some "standard" text corresponding to the accelerator using the **GetAcceleratorText (acc)** function. In fact, this function forms the text itself using the GetKeyNameText API function. If the resulting text does not suit you, you can use your own.

7.1.7 Menu at MCK

For convenient visual design of menus in MCK projects, the mirror components **TKOLMainMenu** and **TKOLPopupMenu** have been developed. With the help of the editor of these components (called by double clicking on the component on the form), it is possible to add, remove, move menu items. When you select items in the displayed menu tree, this provides the ability to edit the properties of individual menu items in the Object Inspector.

I will pay special attention to the constants that MCK forms by default for menu items. This, in fact, is a very convenient tool for accessing menu items (and sometimes you still have to access them, for example, to check their Checked state). In my opinion, it is obvious that the line of code

```
if MainMenu1.ItemChecked [mmOptionOne] then ...
```

is significantly more informative (and will be correct even after any changes in the menu design!) than

```
if MainMenu1.ItemChecked [12] then ...
```

(Or do you think it is not?).

However, if the form uses several different menu components, and the menu items in them remain named N1, N2, ..., i.e. Since the names that were assigned initially have been preserved, then when trying to compile the code, a problem will arise due to the repeated definition of the same constants.

For menu mirrors, MCK has properties **generateConstants** (by default, true, that is, generate) and **generateSeparatorConstants** (by default, false). The second property was introduced to prevent MCK menu mirrors from generating such constants for separators in the menu. Usually

© 2024 Carl Peeraer

there is no need to name separators, although there is a situation when you need to refer to the properties of a separator item (for example, its Visible property).

You can also disable the generation of constants for other items, but if you need to access the properties of menu items in dynamics, then it's better to just rename them.

7.1.8 Menu - Syntax

```
type TMenuitemInfo = packed record
    cbSize: UINT;
    fMask: UINT;
    fType: UINT;             { used if MIIM_TYPE}
    fState: UINT;            { used if MIIM_STATE}
    wID: UINT;               { used if MIIM_ID}
    hSubMenu: HMENU;         { used if MIIM_SUBMENU}
    hbmpChecked: HBITMAP;    { used if MIIM_CHECKMARKS}
    hbmpUnchecked: HBITMAP;  { used if MIIM_CHECKMARKS}
    dwItemData: DWORD;       { used if MIIM_DATA}
    dwTypeData: PKOLChar;    { used if MIIM_TYPE}
    cch: UINT;               { used if MIIM_TYPE}
    hbmpItem: HBITMAP;       { used if MIIM_BITMAP }
  end;
```

```
const
  TPM_HORPOSANIMATION = $0400;
  TPM_HORNEGANIMATION = $0800;
  TPM_VERPOSANIMATION = $1000;
  TPM_VERNEGANIMATION = $2000;
  TPM_NOANIMATION     = $4000;
```

```
type PMenu = ^TMenu;
```

```
type TOnMenuItem = procedure( Sender: PMenu; Item: Integer ) of object;
```

Event type to define OnMenuItem event.

```
type TMenuAccelerator = packed Record
    fVirt: Byte;
    Key: Word;
    NotUsed: Byte;
end;
```

Field	Description
fVirt: Byte;	or-combination of FSHIFT, FCONTROL, FALT, FVIRTKEY, FNOINVERT
Key: Word;	character or virtual key code (FVIRTKEY flag is present above)
NotUsed: Byte;	not used

Menu accelerator record. Use MakeAccelerator 390 function to combine desired attributes into a record, describing the accelerator.

```
type TMenuOption =( moDefault, moDisabled, moChecked, moCheckMark, moRadioMark,
moSeparator, moBitmap, moSubMenu, moBreak, moBarBreak );
```

Options to add menu items dynamically.

```
type TMenuOptions = set of TMenuOption;
```

TMenuOption 386

Set of options for menu item to use it in TMenu.AddItem method.

© 2024 Carl Peeraer

`type TMenuBreak =( mbrNone, mbrBreak, mbrBarBreak );`
Possible menu item break types.

`function MenuStructSize: Integer;`
Returns 44 under Windows95, and 48 (=sizeof(TMenuItemInfo) under all other Windows versions.

Constructors:

`function NewMenu( AParent: PControl; MaxCmdReserve: DWORD; const Template: array of PKOLChar; aOnMenuItem: TOnMenuItem` 386 `): PMenu;`
Menu constructor. First created menu becomes main menu of form (if AParent is a form). All other menus becomes popup (can be activated using Popup method). To provide dynamic replacing of main menu, create all popup menus as children of any other control, not form itself. When Menu is created, pass FirstCmd integer value to set it as ID of first menu item (all other ID's obtained by incrementing this value), and Template, which is an array of PChar (usually array of string constants), containing list of menu item identifiers and/or formatting characters.
FirstCmd value is assigned to first menu item created as its ID, all follow menu items are assigned to ID's obtained from FirstCmd incrementing it by 1. It is desirable to provide not intersected ranges of ID's for different menus in the applet.

Following formatting characters can be used in menu template strings:
- & (in identifier) - to underline next character and use it as a shortcut character when possible;
- + (in front of identifier) - to make item checked. If also ! is used before & than radioitem is defined;
- - (in front of identifier) - item not checked;
- - (separate) - separator (between two items);
- ((separate) - start of submenu;
-) (separate) - end of submenu;

To get access to menu items, use constants 0, 1, etc. It is a good idea to create special enumerated type to index correspondent menu items using Ord( ) operator. Note in that case, that it is necessary only to define constants correspondent to identifiers (positions, correspondent to separators or submenu brackets are not identified by numbers).

`function NewMenuEx( AParent: PControl; FirstCmd: Integer; const Template: array of`
Creates menu, assigning its own event handler for every (enough) menu item.

Properties, Methods and Events

`property Handle : HMenu;`
`Handle of Windows menu object.`

© 2024 Carl Peeraer

`property MenuId: Integer;`
Id of the menu item object. If menu item has subitems, it has also submenu Handle. Top parent menu object itself has no Id. Id-s areassigned automatically starting from 4096. Do not (re)create menu items instantly, because such values are not reused, and maximum possible Id value must not exceed 65535.

`property Parent: PMenu;` 386
Parent menu item (or parent menu).

`property TopParent: PMenu;` 386
Top parent menu, owning all nested subitems.

`property Owner: PControl;`
Parent control or form.

`property Caption: KOLString;`
Menu item caption text (including '&' indicating mnemonic characters, and keyboard accelerator representation string, usually following tabulation character).

`property Items[ Id: HMenu ]: PMenu;`
Returns menu item object by its index or by menu id. Since menu id values are starting from 4096, values from 0 to 4095 are interpreted as absolute index of menu item. Be careful accessing menu items or submenus by index, if you dynamically insert or delete items or submenus. In this version, separators are enumerating too, like all other items. Use index -1 to access object itself. The first item of a menu (or the first subitem of submenu item) has index 0. Children are enumerating before all siblings. The maximum available index is (Count - 1), when accessing menu items by index.

`property Count: Integer;`
Count of items together with all its nested subitems.

`function IndexOf( Item: PMenu ): Integer;` 386
Returns index of an item. This index can be used to access menu item. Value -2 is returned, if the Item is not a child for menu or menu item, and has no parents, which are children for it, etc. Menu object itself always has index -1.

`property OnMenuItem : TOnMenuItem;`
Is called when menu item is clicked. Absolute index of menu item clicked is passed as the second parameter. TopParent always is passed as a Sender parameter.

`property ByAccel: Boolean;`
True, when OnMenuItem 388 is called not by mouse, but by accelerator key. Check this flag for entire menu (TopParent), not for item itself.
(Note, that Sender in OnMenuItem 388 always is TopParent menu object).

© 2024 Carl Peeraer

`property` **IsSeparator:** `Boolean;`
TRUE, if a separator menu item.

`property` **MenuBreak:** `TMenuBreak` 387`;`
Menu item break type.

`property` **OnUncheckRadioItem :** `TOnMenuItem` 386`;`
Is called when radio item becomes unchecked in menu in result of checking another radio item of the same radio group.

`property` **RadioGroup:** `Integer;`
Radio group index. Several neighbor items with the same radio group index form radio group. Only single item from the same group can be checked at a time.

`property` **IsCheckItem:** `Boolean;`
If menu item is defined as check item, it is checked automatically when clicked.

`procedure` **RadioCheckItem;**
Call this method to check radio item. (Calling this method for an item, which is not belonging to a radio group, just sets its Checked state to TRUE).

`property` **Checked:** `Boolean;`
Checked state of the item.

`property` **Enabled:** `Boolean;`
Enabled state of the item. When assigned, Grayed state also is set to arbitrary value (i.e., when Enabled is set to true, Grayed is set to FALSE.)

`property` **DefaultItem:** `Boolean;`
Set this property to TRUE to make menu item default. Default item is drawn with bold. If you change **DefaultItem** at run-time and want to provide changing its visual state, recreate the item first resetting Visible property, then set it again.

`property` **Highlight:** `Boolean;`
Highlight state of the item.

`property` **Visible:** `Boolean;`
Visibility of menu item.

`property` **Data:** `Pointer;`

© 2024 Carl Peeraer

Data pointer, associated with the menu item.

```
property Bitmap: HBitmap;
```
Bitmap used for unchecked state of the menu item.

```
property BitmapChecked: HBitmap;
```
Bitmap used for checked state of the menu item.

```
property BitmapItem: HBitmap;
```
Bitmap used for item itself. In addition, following special values are possible: HBMMENU_CALLBACK, HBMMENU_MBAR_CLOSE, HBMMENU_MBAR_CLOSE_D, HBMMENU_MBAR_MINIMIZE, HBMMENU_MBAR_MINIMIZE_D, HBMMENU_MBAR_RESTORE, HBMMENU_POPUP_CLOSE, HBMMENU_POPUP_MAXIMIZE, HBMMENU_POPUP_MINIMIZE, HBMMENU_POPUP_RESTORE, HBMMENU_SYSTEM.

```
property Accelerator: TMenuAccelerator 386;
```
Accelerator for menu item.

```
property HelpContext: Integer;
```
Help context for entire menu (help context can not be assigned to individual menu items).

```
procedure AssignEvents( StartIdx: Integer; const Events: array of TOnMenuItem 386 );
```
It is possible to assign its own event handler to every menu item using this call. This procedure also is called automatically in a constructor NewMenuEx 387.

```
function MakeAccelerator( fVirt: Byte; Key: Word ): TMenuAccelerator 386;
```
Creates accelerator item to assign it to TMenu.ItemAccelerator[ ] property easy.

```
function GetAcceleratorText( const Accelerator: TMenuAccelerator 386 ):
KOLString;
```
Returns text representation of accelerator.

```
function Insert(InsertBefore: Integer; ACaption: PKOLChar; Event: TOnMenuItem 386;
Options: TMenuOptions 386): PMenu 386;
```
Inserts new menu item before item, given by Id (>=4096) or index value InsertBefore. Pointer to an object created is returned.

```
property SubMenu: HMenu read FHandle; // write SetSubMenu;
```
Submenu associated with the menu item. The same as Handle. It was possible in earlier versions to change this value, replacing (removing, assigning) entire popup menu as a submenu for menu item.

© 2024 Carl Peeraer

But in modern version of TMenu, this is not possible. Instead, entire menu object should be added or removed using **InsertSubmenu** 391 or **RemoveSubmenu** 391 methods.

procedure **InsertSubMenu**(SubMenuToInsert: PMenu 386; InsertBefore: Integer);
Inserts existing menu item (together with its subitems if any present) into given position. See also RemoveSubMenu 391.

function **RemoveSubMenu**(ItemToRemove: Integer): PMenu;
Removes menu item from the menu, returning TMenu object, representing it, if submenu item, having its own children, detached. If an individual menu item is removed, nil is returned.

function **AddItem**(ACaption: PKOLChar; Event: TOnMenuItem 386; Options: TMenuOptions 386): Integer;
Adds menu item dynamically. Returns ID of the added item.

function **InsertItem**(InsertBefore: Integer; ACaption: PKOLChar; Event: TOnMenuItem 386; Options: TMenuOptions 386): Integer;
Inserts menu item before an item with ID, given by InsertBefore parameter.

function **InsertItemEx**(InsertBefore: Integer; ACaption: PKOLChar; Event: TOnMenuItem 386; Options: TMenuOptions 386; ByPosition: Boolean): Integer;
Inserts menu item by command or by position, dependant on ByPosition parameter

procedure **AssignBitmaps**(StartIdx: Integer; Bitmaps: array of HBitmap);
Can be used to assign bitmaps to several menu items during one call.

function **GetMenuItemHandle**(Idx : Integer): DWORD;
Returns Id of menu item with given index.

property **ItemHandle**[Idx: Integer]: DWORD;
Returns handle for item given by index.

property **ItemChecked**[Idx : Integer] : Boolean;
True, if correspondent menu item is checked.

procedure **RadioCheck**(Idx : Integer);
Call this method to check radio item. For radio items, do not use assignment to ItemChecked or Checked properties.

property **ItemBitmap**[Idx: Integer]: HBitmap read GetItemBitmap write SetItemBitmap;
This property allows to assign bitmap to menu item (for unchecked state only - for checked menu items default checkmark bitmap is used).

© 2024 Carl Peeraer

`property ItemText[ Idx: Integer ]: KOLString;`
This property allows to get / modify menu item text at run time.

`property ItemEnabled[ Idx: Integer ]: Boolean;`
Controls enabling / disabling menu items. Disabled menu items are displayed (grayed) but inaccessible to click.

`property ItemVisible[ Idx: Integer ]: Boolean;`
This property allows to simulate visibility of menu items (implementing it by removing or inserting again if needed. For items of submenu, which is made invisible, True is returned. If such item made Visible, entire submenu with all its parent menu items becomes visible. To release menu properly it is necessary to make before all its items visible again.
This does not matter, if menu is released at the end of execution, but can be sensible if owner form is destroyed and re-created at run time dynamically.

`property ItemHelpContext[ Idx: Integer ]: Integer;`
The context of the help system

`property ItemAccelerator[ Idx: Integer ]: TMenuAccelerator 386;`
Allows to get / change accelerator key codes assigned to menu items. Has no effect unless SupportMnemonics 266 called for a form.

`property ItemSubmenu[ Idx: Integer ]: HMenu;` // write SetItemSubmenu;
Retrieves submenu item dynamically. See also SubMenu 390 property.

`procedure RedrawFormMenuBar;`
for the main menu, this call provides an update of the image of the main menu bar after making any modifications in it. If such a call is not made, then the menu itself is not updated.

`property OnMeasureItem: TOnMeasureItem 210;`
This event is called for owner-drawn menu items. Event handler should return menu item height in lower word of a result and item width (for menu) in high word of result. If either for height or for width returned value is 0, a default one is used.

`property OnDrawItem: TOnDrawItem;`
This event is called for owner-drawn menu items.

`property OwnerDraw: Boolean;`
Set this property to true for some items to make it owner-draw.

© 2024 Carl Peeraer

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

Only for popup menu - to popup it at the given position on screen.
Return: If you specify TPM_RETURNCMD in the uFlags parameter, the return value is the menu-item identifier of the item that the user selected.
If the user cancels the menu without making a selection, or if an error occurs, then the return value is zero.
If you do not specify TPM_RETURNCMD in the uFlags parameter, the return value is nonzero if the function succeeds and zero if it fails.

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

This version of popup command is very useful, when popup menu is activated when its parent window is not visible (e.g., for a kind of applications, which always are invisible, and can be activated only using tray icon).
PopupEx method provides correct tracking of menu disappearing when mouse is clicked anywhere else on screen, fixing strange menu behavior in some Windows versions (NT).
Actually, when PopupEx used, parent form is shown but below of visible screen, and when menu is disappearing, previous state of the form (visibility and position) are restored. If such solution is not satisfying You, You can do something else (e.g., use region clipping, etc.)

```
property OnPopup: TOnEvent;
```

This event occurs before the popup menu is shown.

```
property NotPopup: Boolean;
```

Set this property to true to prevent popup of popup menu, e.g. in OnPopup 393 event handler.

```
property Flags: DWORD;
```

Pop-up flags, which are used to call TrackPopupMenuEx, when Popup or PopupEx method is called. Can be a combination of following values:

```
TPM_CENTERALIGN or TPM_LEFTALIGN or TPM_RIGHTALIGN
TPM_BOTTOMALIGN or TPM_TOPALIGN or TPM_VCENTERALIGN
TPM_NONOTIFY or TPM_RETURNCMD
TPM_LEFTBUTTON or TPM_RIGHTBUTTON
TPM_HORNEGANIMATION or TPM_HORPOSANIMATION or TPM_NOANIMATION or
TPM_VERNEGANIMATION or TPM_VERPOSANIMATION
TPM_HORIZONTAL or TPM_VERTICAL.
```

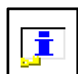
By default, a combination TPM_LEFTALIGN or TPM_LEFTBUTTON is used.

```
property CurCtl: PControl;
```

By Alexander Pravdin. This property is assigned to a control which were initiated a pop-up, for popup menu.
See also conditional compilation symbol: **USE_MENU_CURCTL** 42

© 2024 Carl Peeraer

7.2 Tray Icon (TTrayIcon)



Perhaps this is not the most necessary object for most applications. But I put it directly after the menu, tk. it appears to be almost as visual as the main menu - at least when used and activated. This is an object that encapsulates calls to API functions that display a certain icon in a specially designed area of the system taskbar.

Such an icon (usually referred to as a "tray icon") allows the user to organize a visual connection with the application, which is temporarily hidden from his eyes, i.e. does not even take up the buttons on the taskbar. The application can permanently show its tray icon (or even several icons), or hide them as needed. There is also the ability to change the icon image on the fly, providing animation to show the activity or readiness of a background process.

All the specified functionality, and some additional features, is contained in the TTrayIcon object type. Its **constructor**:

`NewTrayIcon(Parent, icon)`. Here Parent is a pointer to the window object (form), and icon is a handle to an hIcon icon. Initially, the object is created in an inactive state (the icon is not displayed in the system tray).

Object properties, methods and events:

Icon - descriptor of the hIcon type icon;
Active - state of activity;
Tooltip - a tooltip that appears when the mouse cursor stops over the icon. A maximum of 63 characters of this string are displayed (this is the system limit);
AutoRecreate - if you set this value to true, the icon will be automatically restored in the system tray if, for any reason, the Explorer.exe program is restarted. It is Windows Explorer that provides the system bar and other elements of the desktop. usually the user's "shell" of the system. Unfortunately, this program can also crash. Not all (even reputable) applications provide automatic self-recovery of the tray icon after such an incident, and as a result, if their windows are hidden, it is not so easy to return them to the screen;
NoAutoDeactivate - by default, this property is false, i.e. automatic deactivation of the icon is provided when the application is closed;
Wnd - handle to the window used to receive mouse messages. Initially, this is the window of the Parent object specified when the object was created. It is also possible to attach a message handler to a foreign window (not the window of the KOL.TControl object) using the **AttachProc2Wnd** method;
AttachProc2Wnd - attaches a handler to an arbitrary **Wnd** window of the application. Such a window can process messages for several different **TTrayIcon** objects without restriction;
DetachProc2Wnd - detaches the handler from the **Wnd** window;

© 2024 Carl Peeraer

OnMouse - a mouse event that occurs when the mouse cursor moves over the icon of an object in the system area, and when the mouse is clicked on it. The handler receives, besides the sender, only the message type (WM_LBUTTONDOWN, WM_RBUTTONDOWN, WM_MOUSEMOVE, WM_LBUTTONDBLCLK, etc.). Other parameters, such as the coordinates of the mouse cursor on the screen, must be obtained by the application itself;

In **MCK**, the **TTrayIcon** object corresponds to the mirror component **TKOLTrayIcon**.

7.2.1 Tray Icon - Syntax

```
TTrayIcon( unit KOL.pas ) ← TObj 92 ← _TObj 92
TTrayIcon = object( TObj 92 )
```
Object to place (and change) a single icon onto taskbar tray.

```
type TTrayIcon 395 = object( TObj 92 )
```
Object to place (and change) a single icon onto taskbar tray.

```
type TOnTrayIconMouse = procedure( Sender: PObj; Message: Word ) of object;
```
Event type to be called when Applet 369 receives a message from an icon, added to the taskbar tray.

Constructor

```
function NewTrayIcon( Wnd: PControl 203; Icon: HIcon ): PTrayIcon;
```
Constructor of TTrayIcon 395 object. Pass main form or applet as Wnd parameter.

TTrayIcon properties

```
property Icon: HIcon;
```
Icon to be shown on taskbar tray. If not set, value of Active 395 property has no effect. It is also possible to assign a value to Icon property after assigning True to Active 395 to install icon first time or to replace icon with another one (e.g. to get animation effect).
Previously allocated icon (if any) is not deleted using DeleteObject. This is normal for icons, loaded from resource (e.g., by LoadIcon API call). But if icon was created (e.g.) by CreateIconIndirect, your code is responsible for destroying of it).

```
property Active: Boolean;
```
Set it to True to show assigned Icon 395 on taskbar tray. Default is False. Has no effect if Icon 395 property is not assigned. TrayIcon is deactivated automatically when Applet 369 is finishing (but only if Applet 369 window is used as a "parent" for tray icon object).

© 2024 Carl Peeraer

property **Tooltip**: KOLString;
Tooltip string, showing automatically when mouse is moving over installed icon. Though "huge string" type is used, only first 63 characters are considered. Also note, that only in most recent versions of Windows multiline tooltips are supported.

property **AutoRecreate**: Boolean;
If set to TRUE, auto-recreating of tray icon is proveded in case, when Explorer is restarted for some (unpredictable) reasons. Otherwise, your tray icon is disappeared forever, and if this is the single way to communicate with your application, the user no more can achieve it.

property **NoAutoDeactivate**: Boolean;
If set to true, tray icon is not removed from tray automatically on WM_CLOSE message receive by owner control. Set Active395 := FALSE in your code for such case before accepting closing the form.

property **Wnd**: HWnd;
A window to use as a base window for tray icon messages. Overrides parent Control handle is assigned. Note, that if Wnd property used, message handling is not done automatically, and you should do this in your code, or at least for one tray icon object, call AttachProc2Wnd396.

TTrayIcon methods

destructor **Destroy**; virtual;
Destructor. Use **Free** method instead (as usual).

procedure **ForceActive**(SleepTime, Timeout: DWORD);
Sets Active395 := TRUE unil it becomes TRUE or Timeout exceeds, sleeping for SleepTime milliseconds between attempts. E.g.: `Trayicon1.ForceActive(100, 5000);`

procedure **AttachProc2Wnd**;
Call this method for a tray icon object in case if Wnd396 used rather then control. It is enough to call this method once for each Wnd396 used, even if several other tray icons are also based on the same Wnd396. See also DetachProc2Wnd396 method.

procedure **DetachProc2Wnd**;
Call this method to detach window procedure attached via AttachProc2Wnd396. Do it once for a Wnd396, used as a base to handle tray icon messages. Caution! If you do not call this method before destroying Wnd396, the application will not functioning normally.

© 2024 Carl Peeraer

TTrayIcon events

`property OnMouse: TOnTrayIconMouse`395`;`
Is called then mouse message is taking place concerning installed icon. Only type of message can be obtained (e.g. WM_MOUSEMOVE, WM_LBUTTONDOWN etc.)

7.3 File Selection Dialog (TopenSaveDialog)

Dialogues are also non-visual objects, although working with them leads to the appearance of some system windows on the screen. But these are system windows, they cannot be configured as a form, and they appear only for a while, to perform certain actions.

The file selection dialog allows the user, using a standard interface, to select a file name to create and write to it some information provided by the application, or the name of an existing file to perform any actions with it. The file selection dialog can be a file open or save dialog. Only an existing file can be specified in the open dialog. In the save dialog, it is also possible to specify an existing file, and it is usually accepted that the system in this case asks the user an additional question about whether he really wants to write new information into it (most likely, by screwing up the previous contents of this file). Although, if you specify the "silent" mode in the options, this question will not be asked.

The **constructor** of this control object with such a dialog:

`NewOpenSaveDialog(s, dir, options)` - creates a dialog with the header s (if the string is empty, then the system header is used), with the initial dir directory and options from the following set:

OSCreatePrompt - asking the user for confirmation to create a file, if it does not already exist;
OSExtensionDifferent - contains true after the end of the dialog, if the extension of the selected file differs from the default;
OSFileMustExist - the file must exist;
OSHideReadonly - hide the "Read only" switch in the dialog;
OSNoChangedir - the user can select the file name only in the specified directory;
OSNoReferenceLinks - for shortcuts (.lnk) return the path to the shortcut file itself, and not to the file associated with it;
OSAllowMultiSelect - allows multiple choice (multiple filenames are returned);
OSNoNetworkButton - does not allow selection in network folders;
OSNoReadonlyReturn - do not return files with the "read-only" attribute, including from devices and directories to which writing is not allowed;
OSOverwritePrompt - issue a request for confirmation of overwriting for existing files;
OSPathMustExist - the directory must exist;

© 2024 Carl Peeraer

OSReadonly - the "read-only" switch is initially on, when the dialog is finished this option shows the last value of the switch;
OSNoValidate - do not check the presence of such a file and the correctness of the name (when manually typing the file name in the dialog box);
OSTemplate - the dialog extension template from resources is used (see the Template property);
OSHook - enables an additional custom dialog message handler (see the HookProc property).

To simplify the creation procedure, so as not to write out all the necessary properties every time, you can pass the global constant **DefOpenSaveDlgOptions** as a parameter. Subsequently, the options can be changed to the standard ones that are most suitable for the open or save dialog by changing the value of the **OpenDialog** property.

So, the properties, fields, method and event of the object to control the file selection dialog:

Execute - a function for invoking a dialogue. Returns true if the dialog ended with a successful file selection;
Filename - a string that at the output contains the name of the selected file (if the dialog ended successfully, i.e. the Execute method returned true). At the entrance, i.e. before calling the Execute method, this property can be used to assign a default filename to return (this name will be shown initially in the filename input field). When enabling multiple file selection in a dialog, be sure to clear this property by assigning an empty string before calling Execute. For the case when multiple files are selected, the output line is split into parts separated by # 13, the first part contains the path to the directory, and all the rest contain only the file names;
InitialDir - the source directory, the list of which is opened when calling Execute. After calling the dialog, in case of its successful completion, this property contains the path to the directory in which the file was selected;
Filter - a string containing pairs <filter definition> | <filter patterns> separated by '|' (i.e., if there are several filters, then it schematically looks like this: <OF1> | <WF1> | <OF2> | <WF2> | ...). If there are several templates in one filter, then they are separated by the ';' symbol. An example of a typical filter: 'Documents | * .doc; *. Txt | All files | *. *';
FilterIndex - index of the current filter (both before the call to Execute and as a result of its successful execution);
DefExtension - the default extension string (written without a leading period, i.e., for example, 'txt', not '.txt'. Apparently, this property has no other purpose than the ability to check for differences in the extension of the selected file from the default extension, just not;
Title - the title of the dialogue. If this line is empty, the system displays its title;
WndOwner - a window for processing messages, and for transferring to the system as the "owner" of the dialogue (at the end of the dialogue, this window will be activated, as after the usual exit from any modal dialogue);
OpenDialog - when assigning a value (true or false) to the Options property, a set of the most appropriate options is assigned, respectively, for the dialog for opening and saving a file;
Options - dialogue options;
OpenReadOnly - TRUE after Execute401, if Read Only check box was checked by the user. Options400 are not affected anyway.

© 2024 Carl Peeraer

TemplateName - the name of the resource from which the system loads and configures the extension for the dialog. It is required to add the **OpenSaveDialog_Extended** symbol to the list of symbols of the conditional compilation of the project, and the **osTemplate** value in the option; **HookProc** - an event for processing messages from the dialogue. It also requires the symbol for conditional compilation OpenSaveDialog_Extended, and the option values osHook; **NoPlaceBar** - prohibits showing the "placements" ruler on the left side of the new style dialog.	An additional window template for a dialog in a resource can be created using Borland Workshop or MS Visual C ++. Thus, for example, a set of additional checkboxes, buttons or windows can be created to display the contents of the selected file. You will have to work with windows in the HookProc handler at a low level by calling API functions. But sometimes the ability to add your own controls to the standard opening dialog can be very useful or just necessary.

In the **MCK package**, this object is represented by a non-visual mirror component **TOpenSaveDialog**.

7.3.1 File Selection Dialog - Syntax

```
TOpenSaveDialog( unit KOL.pas ) ← TObj 92 ← _TObj 92
TOpenSaveDialog = object( TObj 92 )
```

Object to show standard Open/Save dialog. Initially provided for XCL by Carlo Kok.

```
type TOpenSaveDialog = object( TObj 92 )
```

Object to show standard Open/Save dialog. Initially provided for XCL by Carlo Kok.

```
type TOpenSaveOption = ( OSCreatePrompt, OSExtensionDiffent, OSFileMustExist,
OSHideReadonly, OSNoChangedir, OSNoReferenceLinks, OSAllowMultiSelect,
OSNoNetworkButton, OSNoReadonlyReturn, OSOverwritePrompt, OSPathMustExist,
OSReadonly, OSNoValidate, OSTemplate, OSHook);
```

```
type TOpenSaveOptions = set of TOpenSaveOption;
```

Options available for TOpenSaveDialog 399.

Constructor

```
function NewOpenSaveDialog( const Title, StrtDir: KOLString; Options:
TOpenSaveOptions 399 ): POpenSaveDialog;
```

Creates object, which can be used (several times) to open file(s) selecting dialog.

© 2024 Carl Peeraer

TOpenSaveDialog properties

`property` **`Filename:`** `KOLString;`
Filename is separated by #13 when multiselect is true and the first file, is the path of the files selected.

```
C:\Projects
Test1.Dpr
Test2.Dpr
```

If only one file is selected, it is provided as (e.g.) C:\Projects\Test1.dpr
For case when OSAllowMultiselect 399 option used, after each call initial value for a Filename containing several files prevents system from opening the dialog. To fix this, assign another initial value to Filename property in your code, when you use multiselect.

`property` **`InitialDir:`** `KOLString;`
Initial directory path. If not set, current directory (usually directory when program is started) is used.

`property` **`Filter:`** `KOLString;`
A list of pairs of filter names and filter masks, separated with '|'. If a mask contains more than one mask, it should be separated with ';'. E.g.:
'All files|*.*|Text files|*.txt;*.1st;*.diz'

`property` **`FilterIndex:`** `Integer;`
Index of default filter mask (0 by default, which means "first").

`property` **`OpenDialog:`** `Boolean;`
True, if "Open" dialog. False, if "Save" dialog. True is default.

`property` **`Title:`** `KOLString;`
Title for dialog.

`property` **`Options:`** `TOpenSaveOptions 399;`
Options.

`property` **`DefExtension:`** `KOLString;`
Default extention. Set it to desired extension without leading period, e.g. 'txt', but not '.txt'.

`property` **`WndOwner:`** `THandle;`
Owner window handle. If not assigned, Applet.Handle 369 is used (whenever possible). Assign it, if your application has stay-on-top forms, and a separate Applet 369 object is used.

© 2024 Carl Peeraer

```
property OpenReadOnly: Boolean;
```

TRUE after Execute 401, if Read Only check box was checked by the user. Options 400 are not affected anyway.
Properties, inherited from TObj 92

```
Property TemplateName: KOLString;
```

Do not forget to add OpenSaveDialog_Extended 41 to project options conditionals!

```
Property HookProc: Pointer;
```

```
Property NoPlaceBar: Boolean;
```

TRUE, if place bar is disabled in the new style dialogs (if the symbol OpenSaveDialog_Extended 41 is not added in project options, place bar is always enabled in Windows 2000 and higher).

TOpenSaveDialog methods

```
destructor Destroy; virtual;
```

destructor

```
Function Execute: Boolean;
```

Call it after creating to perform selecting of file by user.

7.4 Directory Selection Dialog (TOpenDirDialog)

The directory selection dialog is one of the cases when in KOL it was decided to create an object type, while on the VCL it is proposed to directly call an API function, entering many parameters each time. I decided to "wrap" the call of this function into an object also because, just like for the file selection dialog, there are a number of properties that can "inherit" their state from one call of the dialog to another (namely: start the directory that is made current when the dialog is called on the screen). It is completely incomprehensible why the user should always start choosing a directory from the "My Computer" or "Desktop" folder. It can be especially offensive if very often you have to select the same directory while interacting with the application. However, if you like,

 © 2024 Carl Peeraer

Constructor:
NewOpenDirDialog(s, options) - creates a dialog for choosing a directory with the header s (if the line is empty, then the standard system header is used), and with a set of options:

odBrowseForComputer - dialog for choosing a computer, not a directory;
odBrowseForPrinter - the dialog is used to select a printer;
odDontGoBelowDomain - do not include network folders below the domain level (if someone does not know what exactly this phrase means, then most likely you simply do not need this option);
odOnlyFileSystemAncestors - the dialog allows you to select only system file objects. / Similar to the previous one, if this phrase tells you little, there is nothing terrible about it. At the moment, I myself do not know exactly what this means. Whenever I need to find out, I'll check the Help. /
odOnlySystemDirs - the dialog allows you to select only system folders;
odStatusText - the presence of the status bar in the dialog is ensured;
odBrowseIncludeFiles - the dialog also displays the contents of folders (list of files);
odEditBox - the presence of a field for entering the name of the folder or the entire path is provided;
odNewDialogStyle - a new style of dialogue. Compared to the old style, the new one, for example, provides the ability to resize the dialog.

Properties, method and event of the dialog:

Execute - a method for invoking a dialog. Similar to the **TOpenSaveDialog** 397 dialog, returns true if the dialog succeeded by selecting a directory;
Title - the title of the dialog box. If the string is empty, the system uses the default header;
Options - dialogue options;
Path - path to the directory selected by the user at the exit from the dialog;
InitialPath - the initial path from which the directory selection starts when showing the dialog;
CenterOnScreen - center the dialog box on the screen when displaying;
OnSelChanged - an event that is triggered when the user selects a different directory while the dialog is running. The handler can prohibit the selection of some directories, making the selection button unavailable;
DialogWnd - the window of the dialog itself (available during the execution of the dialog itself, can be used to enumerate its children in the custom event handler **OnSelChanged**);
WndOwner - the window responsible for the transmission of messages. The same window is used as the "parent" for the dialog, i.e. is automatically activated upon completion.

In the package of mirror components **MCK**, the **TKOLOpenDirDialog** component takes on the role of the image of this object.
But, in addition, it has a design-time property **AltDialog**, setting which to true means that the form will use the alternative **TOpenDirDialogEx** 404 **dialog** (see below).

© 2024 Carl Peeraer

7.4.1 Directory Selection Dialog - Syntax

TOpenDirDialog(unit KOL.pas) 🡐 TObj 92 🡐 _TObj 92
TOpenDirDialog = object(TObj 92)
Dialog for open directories, uses SHBrowseForFolder.

type **TOpenDirOption** =(odBrowseForComputer, odBrowseForPrinter, odDontGoBelowDomain, odOnlyFileSystemAncestors, odOnlySystemDirs, odStatusText, odBrowseIncludeFiles, odEditBox, odNewDialogStyle);
Flags available for TOpenDirDialog 403 object.

type **TOpenDirOptions** = set of TOpenDirOption 403;
Set of all flags used to control ZOpenDirDialog class.

type **TOnODSelChange** = procedure(Sender: POpenDirDialog; NewSelDir: PKOL_Char; var EnableOK: Integer; var StatusText: KOL_String) of object;
Event type to be called when user select another directory in OpenDirDialog. Set EnableOK to -1 to disable OK button, or to +1 to enable it. It is also possible to set new StatusText string.

Constructor

function **NewOpenDirDialog**(const Title: KOLString; Options: TOpenDirOptions 403): POpenDirDialog;
Creates object, which can be used (several times) to open directory selecting dialog (using SHBrowseForFolder API call).

TOpenDirDialog properties

property **Title**: KOLString;
Title for a dialog.

property **Options**: TOpenDirOptions 403;
Option flags.

property **Path**: KOLString;
Resulting (selected by user) path.

property **InitialPath**: KOLString;
Set this property to a path of directory to be selected initially in a dialog.

© 2024 Carl Peeraer

`property CenterOnScreen: Boolean;`
Set it to True to center dialog on screen.

`property WndOwner: HWnd;`
Owner window. If you want to provide your dialog visible over stay-on-top form, fire it as a child of the form, assigning the handle of form window to this property first.

`property DialogWnd: HWnd;`
Handle to the open directory dialog itself, become available on the first call of callback procedure (i.e. on the first call to OnSelChanged 404).

TOpenDirDialog methods

`destructor Destroy; virtual;`
Destructor

`function Execute: Boolean;`
Call it to select directory by user. Returns True, if operation was not cancelled by user.

TOpenDirDialog events

`property OnSelChanged: TOnODSelChange;` 403
This event is called every time, when user selects another directory. It is possible to enable/disable OK button in dialog and/or change dialog status text in response to event.

7.5 Alternative Directory Selection Dialog (TOpenDirDialogEX)

Sometimes the slowness of opening the standard dialog for choosing a directory becomes annoying. But even that is not the main reason why I finally (most recently) made my own dialogue for this purpose. The main reason is the incorrect display of the directory tree (simply, the "glitchiness" of the system dialogue).

For example, a dialog starts to open, and a certain directory is selected in the tree by default. And the parent folder for this directory only shows this subordinate folder, and flatly refuses to show its other children. In addition, I do not really understand the intention of the Microsoft programmers, who designed this dialog in such a way that, by default, the focus is not on the folder tree, but on the OK button. As if this dialog is needed only for the user to confirm the choice of the directory that the program offers him.

© 2024 Carl Peeraer

Further, the ability to create directories for selection in the directory selection process is potentially a very good feature. But how absurdly it is implemented! After creating the directory and renaming it, the line "New Folder" remains in the input field. In order to still select the newly created directory, you have to do additional manipulations. It would have been better then this opportunity had not existed at all. [Ed. 2010: This item appears to have been fixed in Windows 7 - have you really read the book about KOL?]

And again, speed. Why recalculate the entire tree of folders from disk every time, if the dialog can be hidden after the first call, and quickly shown again on the screen after repeated calls? After all, it saves a lot of time and nerves.

Disadvantages of my alternate dialog:
everything is done by its own code, i.e. the size of the application is larger (by how much it depends on whether the same objects are used elsewhere in the program code);
there is no way to create a new directory during the selection process;
the alternative dialog is not intended and cannot be used to search for a computer, printer, or network folders if they are not connected as virtual drives with their own device letters.

Everything else I would attribute to the benefits. For example, the size and position of the dialog box can be easily controlled from the application. You can add your own elements to the form (including ensuring the creation of a new folder, if required). And most importantly, the speed of reopening is almost instantaneous (and for the first time there are fewer "brakes", forgive me this vernacular expression).

Note; Starting with version 3.00, this useful property of fast folder tree building has been "compounded" by the transition to directory scanning by UNICODE versions of API functions that perform file enumeration. Of course, these versions of the functions are only used on their respective NT-based operating systems.

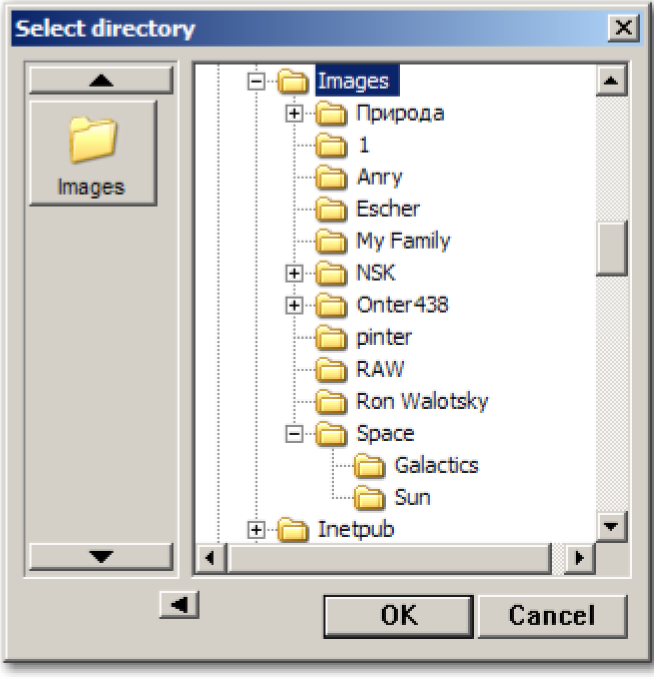
The **TOpenDirDialogEx** object type is implemented in a separate module **KOLDirDlgEx.pas**. Its **constructor**:

NewOpenDirDialogEx - does not require parameters.

 © 2024 Carl Peeraer

Methods and properties:

Execute - a method for displaying the dialog form on the screen in the modal window mode, until the user closes the dialog, or until a directory is selected from the folder tree;
InitialPath - the directory from which the dialogue starts;
Path - folder selected by the user (in case of successful completion of the dialogue);
Form - pointer to the **PControl** window object, which is a dialog form. You can customize this shape (color, size, and other parameters), attach any handlers to it, add your own or change existing elements - before calling the dialog, at your discretion. For the composition of the form, see the **CreateDialogForm** method in the implementation part, where it is generated dynamically;
Title - the title of the dialogue form;
OKCaption - the title of the OK button. By default, the string is 'OK';
CancelCaption - the title of the Cancel button. By default 'Cancel';
FilterAttrs - a set of file attributes for a directory filter. This field allows you to determine whether to give the ability to select system and hidden folders - if necessary (enabling the corresponding attribute value excludes directories with this attribute from viewing);
FilterRecycled - setting this property to true excludes the folder for "deleted" files from the list of displayed files, regardless of its name on this computer (Recycled Bin, Recycle Bin, etc.).



In addition, a link bar can be added to the extended folder selection dialog. To do this, add the **conditional compilation symbol DIRDLGEX_LINKSPANEL** and set the **LinksPanelOn property to TRUE**. But that's not enough if only youdo not intend to force the user to fill the left panel of the dialog with links again in each session of the application. You should use the **CollectLinks function** at the end of the work to get a list of links selected by the user in the left panel, after which you can save it in a way convenient for you - in the registry, ini-file, or something else, so that it can be downloaded and used on subsequent launches of the program. To programmatically add links to the left pane, use calls to the **AddLinks method**.
In addition, in the process of work, properties with self-explanatory names are available:
Links [] - list of links;
LinksCount - the number of links;
LinkPresent[] - checks for a link with the specified path to the folder,
as well as methods:
RemoveLink (lnk) - removes the link to the folder with the specified path;
ClearLinks - clears the list of links.

 © 2024 Carl Peeraer

For this object, I decided not to make a separate mirror in MCK, but to use the existing mirror component **TKOLOpendirDialog** 401, adding only the design-time property **AltDialog** 402. This property allows you to instantly "turn" a standard dialogue into an alternative one, and vice versa.

But the composition of the properties used to customize the dialog at the design stage of the form does not change. For the alternative dialog, the MCK generates a code that takes into account only those properties that coincide with the properties of the standard dialog (in this case, properties that are not "inherent" to the alternative dialog are ignored). Changing other settings of the alternative dialog, specific only to it, must be done by your code, during the execution of the application.

7.5.1 Alternative Directory Selection Dialog - Syntax

```
TOpenDirDialogEx( unit KOL.pas ) ← TObj 92 ← _TObj 92
TOpenDirDialogEx = object( TObj 92 )

Type POpenDirDialogEx = ^TOpenDirDialogEx;
```

Constructor

```
function NewOpenDirDialogEx: POpenDirDialogEx;
```
Creates object, which can be used (several times) to open alternative directory selecting dialog

TOpenDirDialogEx properties

```
Property OKCaption: KOLString;
```
The title of the OK button. By default, the string is 'OK'.

```
Property CancelCaption: KOLString;
```
The title of the Cancel button. By default 'Cancel'.

```
Property FilterAttrs: DWORD;
```
A set of file attributes for a directory filter. This field allows you to determine whether to give the ability to select system and hidden folders - if necessary (enabling the corresponding attribute value excludes directories with this attribute from viewing).

```
Property FilterRecycled: Boolean;
```
Setting this property to true excludes the folder for "deleted" files from the list of displayed files, regardless of its name on this computer (Recycled Bin, Recycle Bin, etc.).

```
Property Title: String;
```
The title of the dialogue form.

© 2024 Carl Peeraer

`Property **Form**: PControl;`
DialogForm object. Though it is possible to do anything since it is in public section, do this only if you understand possible consequences.
E.g., use it to change DialogForm bounding rectangle on screen or to add your own controls, event handlers and so on.

`property **InitialPath**: KOLString;`
Set this property to a path of directory to be selected initially in a dialog.

`property **Path**: KOLString;`
Resulting (selected by user) path.

`property **FastScan**: Boolean;`

`property **Links**[ idx: Integer ]: KOLString;`
List of links

`function **CollectLinks**: PStrList;`

`property **LinksPanelOn**: Boolean;`

`property **LinksCount**: Integer;`
The number of links

`function **LinkPresent**( const s: KOLString ): Boolean;`
Checks for a link with the specified path to the folder.

`function GetLinksPanelOn: Boolean;`

TOpenDirDialog methods

`procedure **DoubleClick**( Sender: PControl; var M: TMouseEventData );`

`procedure **CreateDialogForm**;`
Method in the implementation part, where it is generated dynamically.

`Destructor **Destroy**; virtual;`
Destructor

`function **Execute**: Boolean;`
Call it to select directory by user. Returns True, if operation was not cancelled by user.

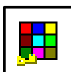
`procedure **AddLinks**( SL: PStrList );`

© 2024 Carl Peeraer

```
procedure RemoveLink( const s: KOLString );
```
Removes the link to the folder with the specified path.

```
procedure ClearLinks;
```
Clears the list of links.

7.6 Color Selection Dialog (TColorDialog)



There is no friend for taste and color. (Russian folk proverb)

When working with graphics, when customizing the interface, etc., you often have to choose the color of the drawing tool, interface element, etc. This work is performed by this object, referring to the standard system color selection dialog.

Constructor:
NewColorDialog(fillopen) - creates an object for invoking the color dialog, returning a pointer of the PColorDialog type. The fullopen parameter specifies whether the dialog will immediately open "completely", with an additional field for choosing an arbitrary True Color (16 million colors), or only in a reduced form.

Object's only method:
Execute - calls a dialog to the screen, and in case of a successful color selection, signals this by returning the value true. The selection result should be read from the Color field of the object, after returning from the Execute method.

To customize the dialog, before invoking the dialog, you can change the following fields:
OwnerWindow - the window that "owns" the dialog (it becomes active immediately after the end of the dialog, it is also used to determine the place on the screen for placing the dialog at the moment of its opening);
CustomColors[1..16] - additional 16 colors, which are placed in additional squares at the bottom of the dialog. By default, all these squares are white, and in the dialog itself, the user can add his own colors to them using the shape extension, where you can select an arbitrary RGB color;
ColorCustomOption - additional operating mode (open completely, open in abbreviated form, do not allow to open completely);
Color - the color chosen by the user as a result of the successful completion of the dialog, such as TColor.

Mirror in **MCK**: **TKOLColorDialog**.

© 2024 Carl Peeraer

7.6.1 Color Selection Dialog - Syntax

TColorDialog(unit KOL.pas) 🡐 TObj 92 🡐 _TObj 92
TColorDialog = object(TObj 92)
Color choosing dialog.

Constructor

function **NewColorDialog**(FullOpen: TColorCustomOption): PColorDialog;
Creates color choosing dialog object.

TColorDialog methods

function **Execute**: Boolean;
Call this method to open a dialog and wait its result.

TColorDialog fields

OwnerWindow: HWnd;
Owner window (can be 0).

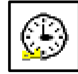
CustomColors: array[1 . . 16] of TColor;
Array of stored custom colors.

ColorCustomOption: TColorCustomOption;
Options (how to open a dialog).

Color: TColor;
Returned color (if the result of Execute 410 is True).

© 2024 Carl Peeraer

7.7 Clock (TTimer)



Now is the time to talk about time counting. As with the VCL, KOL has a TTimer object for this. It, when activated, creates a system object that regularly invokes the designated timer handler. In fact, such a timer is bound to one of the windows: by default, to a window specially created for all timers of the main thread, or, if the **TIMER_APPLETWND** symbol is specified in the application, then to the applet window (or the main form if the applet is not used).

Why am I writing all this. First, it is clear from what has been said that there are no guarantees that such a watch will "tick" with the highest accuracy. The time lag between invocations of the event handler assigned to the timer may not be exactly the same. And this essentially depends, among other things, on the speed of the system, on the degree of its workload with various tasks, on the requested response period. In particular, such a timer is unlikely to be triggered more often than once every 50 milliseconds, i.e. more often than 20 times per second (1 millisecond = 0.001 seconds, i.e. 1000 milliseconds are included in a second, if anyone has forgotten).

Second, a message from the system is first queued and then processed. If your own task is busy with any long calculations (or waiting), it will not be able to handle this event from a simple timer until control returns to the message loop, or one of the methods like **ProcessMessages** is called.

Third, it is obvious that in order for a timer that works through window messages to work, it needs at least one window. In case the application has no windows at all (for example, if you are creating a console application), a special window (TimerOwnerWnd) is created for timers by default. But the window handle can be saved by specifying the **TIMER_APPLETWND** conditional compilation symbol in the project options. In this case, the applet window will be used (which may sometimes be the same as the main form window). The timer message handler is "attached" to this window. But if there is no such window, then the **TTimer** object will not be used.

Note that if, in the case of a multithreaded application, the first timer is "started" (by setting its Enabled property to TRUE) not in the context of the main thread, and the conditional compilation symbol **TIMER_APPLETWND** is not defined in the project, then the special TimerOwnerWnd window that "owns" the timer will be created in the context of the current one, i.e. not the main thread. As a result, if there is no message loop running on that thread, your timers will never fire. I was somehow “lucky” to get this very rare combination of conditions, after which I had to puzzle for a long time what was wrong. In my case, the problem was solved by adding the TIMER_APPLETWND symbol, but in principle, it can be solved by immediately starting (and stopping, if not really needed) some trial timer in the main thread, for example, in the **OnFormCreate** handler.

© 2024 Carl Peeraer

I would like to note that in order to ensure minimal code, the _NewWindowed call is used to create such a window, and for this reason the window is not a clean window for receiving only messages. It becomes the so-called topmost window, and can receive broadcast messages. If there is a separate Applet object in the application and the **OnMessage** handler is installed in it, this handler will receive, among other things, all messages intended for this invisible window. This means, in particular, that system broadcasts will be intercepted one time more than you have forms in the application. Conclusion: parse the handle field of the incoming message to find out which window it is intended for, if required.

However, for all its drawbacks, an important advantage of such an imprecise timer is its relative safety. Its timer handler is called on the same thread (thread) of commands in which the code of other event handlers is running. That is, in the case of a single-threaded application, event handlers never intersect at all, because each of them, including the timer handler, can be considered executing "continuously" within the task. Of course, the system can interrupt it and switch to another task, but then it will still return control to this particular code when it returns control to your application.

Clock constructor:

NewTimer(i)- creates a **TTimer** object with an interval of i milliseconds, returning a pointer of the PTimer type. The timer is initially created inactive. To run it, you need to set its Enabled property to true.

Timer properties, methods and event:

Handle - descriptor of the hTimer system object, i.e. a number that allows the system to identify this object in low-level API requests. This descriptor contains the value 0 if the timer is currently inactive (the system object is only created when the timer starts);
Enabled - timer activity. This property can be used to start or restart the timer (to restart the clock, you must first stop the clock, that is, set the Enabled property to false);
Interval - timer interval. When this value is changed, when the object is active, the timer is "reset", i.e. the system object is recreated, and the countdown to the next triggering starts over;
OnTimer - timer event. In the event handler, you are allowed to change any properties of the timer object, including the interval, or the active state. For example, if you want the timer to fire once instead of regularly, you would add code to the handler to set the Enabled property to false.

The **TTimer object MCK** has a mirrored **TKOLTimer** component. But it allows you to generate code for more than just a simple clock object. When the multimedia timer object was developed, I decided to use the same mirror component to generate its code, especially since the TTimer and TMMTimer objects are very similar (see below). As a result, the TKOLTimer mirror has been enriched with a number of properties that cannot be used when generating code for a regular timer, and are simply ignored, namely: **Periodic**, **Resolution**.

© 2024 Carl Peeraer

7.7.1 Multimedia Timer (TMMTimer)

This object is a more accurate instrument for counting time intervals than a simple timer. It uses the so-called "multimedia" timer, which does not require a window handle for its operation, and instead of sending messages, it directly calls the custom handler. Moreover, the call always takes place in someone else's (system) command stream. Those. not only is it not guaranteed that the event will only fire when the process is waiting for messages, but on the contrary: it will almost certainly interrupt the current operation in order to execute the specified handler.

Hierarchically, TMMTimer does not derive from TObj, but inherits from **TTimer**. The SetEnabled method of the TObj object is virtual, therefore, in principle, you can pass the TMMTimer object as a parameter of some procedures instead of TTimer.

Multimedia Timer Constructor:

NewMMTimer(i)- returns a pointer of the **PMMTimer** type. As an interval, you can specify values less than 50 milliseconds (including the minimum possible value of 1 millisecond). But the accuracy of this timer, although higher, still cannot be limitless. The Windows operating system is not a real-time system. Even the observance of the declared accuracy for this kind of timer (10 milliseconds by default) is not guaranteed if the system suddenly thought that it had more important things to do. To improve accuracy, you can specify a higher resolution (i.e. lower the Resolution property), raise the priority of your task, or stop using Windows (just kidding).

Properties, methods, event for TMMTimer are the same as those of its ancestor in the TTimer hierarchy. Two more properties are added:

Periodic - the timer is periodic (by default, this property contains true immediately after the object is created, i.e. the timer is created periodic). The non-periodic timer differs in that when triggered, it automatically goes into an inactive state, i.e. it is "disposable";
Resolution - the accuracy of the multimedia timer. A value of 0 (which is used by default) means absolute accuracy, but leads to a complete degradation of system performance. Those. the system can no longer do anything, in this case it only counts the time, and the task manager, if it can work, will only show that the processor is 100% loaded. A value of 10 is generally an acceptable value and it is recommended not to use lower values whenever possible.

In the general **MCK mirror** for simple and multimedia timers, **TKOLTimer**, specifically for setting up a multimedia timer, there are Periodic and Resolution properties. The design-time **property Multimedia** should be used to switch the timer from normal to multimedia and back again.

And please don't forget that the media timer handler is called on its own command flow, i.e. it is necessary to ensure the protection of resources and parts of the code that are not "re-entrant" (reentrant is an old designation, which even managed to firmly enter the technical language before the advocates of the purity of Russian speech realized themselves). In particular, it is undesirable to work with window objects, except by sending messages to them by en-queuing messages (TControl.Postmsg method, or PostMessage API function). See also the next chapter (TThread object) for more details on securing code sections.

© 2024 Carl Peeraer

7.7.2 Clock - Syntax

TTimer(unit KOL.pas) ← TObj 92 ← _TObj 92
TTimer = object(TObj 92)
Easy timer encapsulation object. It uses separate topmost window, common for all timers in the application, to handle WM_TIMER message. This allows using timers in non-windowed application (but anyway it should contain message handling loop for a thread).
Note: in UNIX, there are no special windows created, certainly.

TMMTimer(unit KOL.pas) ← TTimer ← TObj 92 ← _TObj 92
TMMTimer = object(TTimer)
Multimedia timer encapsulation object. Does not require Applet 369 or special window to handle it. System creates a thread for each high resolution timer, so using many such objects can degrade total PC performance.

type **TTimer** = object(TObj 92)
Easy timer encapsulation object. It uses separate topmost window, common for all timers in the application, to handle WM_TIMER message. This allows using timers in non-windowed application (but anyway it should contain message handling loop for a thread).
Note: in UNIX, there are no special windows created, certainly.

type **TMMTimer** = object(TTimer 414)
Multimedia timer encapsulation object. Does not require Applet 369 or special window to handle it. System creates a thread for each high resolution timer, so using many such objects can degrade total PC performance.

Constructors:

function **NewTimer**(Interval: Integer): PTimer;
Constructs initially disabled timer with interval 1000 (1 second).

function **NewMMTimer**(Interval: Integer): PMMTimer;
Creates multimedia timer object. Initially, it has Resolution = 0, Periodic = TRUE and Enabled = FALSE. Do not forget also to assign your event handler to OnTimer to do something on timer shot.

TTimer properties

property **Handle**: Integer;
Windows timer object handle.

© 2024 Carl Peeraer

`property` **`Enabled:`** `Boolean;`
True, is timer is on. Initially, always False.

`property` **`Interval:`** `Integer;`
Interval in milliseconds (1000 is default and means 1 second). Note: in UNIX, if an Interval can be set to a value large then 30 minutes, add a conditional definition SUPPORT_LONG_TIMER to the project options.

TMMTimer properties

`property` **`Resolution:`** `Integer;`
Minimum timer resolution. The less the more accuracy (0 is exactly Interval 415 milliseconds between timer shots). It is recommended to set this property greater to prevent entire system from reducing overhead. If you change this value, reset and then set Enabled 415 again to apply changes.

`property` **`Periodic:`** `Boolean;`
TRUE, if timer is periodic (default). Otherwise, timer is one-shot (set it Enabled 415 every time in such case for each shot). If you change this property, reset and set Enabled 415 property again to get effect.

TTimer methods

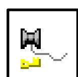
`destructor` **`Destroy;`** `virtual;`
Destructor.

TTimer events

`property` **`OnTimer:`** `TOnEvent;`
Event, which is called when time interval is over.

© 2024 Carl Peeraer

7.8 Thread, or thread of commands (TThread)



Just like the VCL, to organize an independent thread of commands (or thread of commands, thread is translated as "thread"), the KOL library has an object that is named the same - TThread. But, unlike VCL, in KOL you do not need to create a descendant of the TThread object type to organize your own thread. It is enough to assign an OnExecute event handler to it, and in it implement the code that will be executed when the thread is started. To start the thread, call the Resume method (newbies who are not familiar with this circumstance try to call the Execute method, but this does not lead to the desired results, since in this case the launch occurs in the same thread from which Execute was called).

There are several different constructors for organizing command flow:

NewThread - creates an object in the Suspended state. After such an object is created, it is possible to assign an event handler to it on the OnExecute event, and start the thread (Resume);
NewThreadEx(onexec) - creates an object, assigns the handler specified in the parameter to the OnExecute event, and starts it for execution (unless the onexec parameter is nil). Those. as a result of this construction of the thread, it starts working immediately. If the application is running on a machine with one processor, most likely, some part of the handler code (that is, the thread code) will be executed before control returns to the thread that created the object, at the point following the call to the NewThreadEx constructor;
NewThreadAutoFree(onexec) - creates and starts a thread object similarly to the previous constructor, but additionally provides automatic destruction of the object upon completion of the thread.
An important detail: the thread object in KOL does not allow itself to be started more than once. In any case, if the stream was terminated, i.e. there was a return from the OnExecute handler, the object is no longer needed. The difference in the third construction method is that the object itself is comfortable with calling the Free method when the handler completes.

Methods, properties and events of the TThread object:

Execute - this method is for internal purposes only (although it is declared in the public section). In principle, since this method is declared virtual, it is possible to inherit the TThread object, similar to how it is done in the VCL. But I prefer to use the more economical method of assigning a handler to the OnExecute event. This method is also more convenient, as it allows you to use the "mirror" component of the MCK, and generate code for the stream "visually". (Can you guess why the VCL does not have a design-time component to represent command streams, and the code has to be written by hand?);
Resume - puts the suspended (Suspended) thread in the "command execution" mode. If the thread is already running, this method does nothing. To start a thread, you should use this method, not Execute, otherwise the thread statements will be executed directly in the same thread from which Execute was called;

© 2024 Carl Peeraer

Suspend - suspends the execution of the thread. This method can also be executed in the suspended thread itself. In this case, the call to the Suspend method will return only when the Resume method is executed for it from another thread;
Suspended - verifies that the thread is "suspended";
Terminate - stops the thread using the TerminateThread API function. Unlike the VCL, this is not a normal shutdown method, but rather an emergency shutdown. In order to terminate the execution of a thread more safely, you should somehow notify your thread that it would be time for it to terminate (for example, set a flag in some global variable or in a field of an object known to the thread - and at least use its Tag property common to all descendants of TObj 92, including the TThread object). After that, you should wait for some time for your flow to take note of this information and end itself. Of course, the code for such interaction is entirely on the shoulders of the programmer, that is, on you;
Terminated - checks that the stream has terminated (not necessarily abnormally, using the Terminate method, maybe in the usual way - by returning from the OnExecute handler);
WaitFor - waiting for the completion of this thread, the return from this method occurs only when the thread has actually finished;
Handle - the system handle to the TThread thread object. It can be useful for any API calls, for example, it can be passed to the WaitForMultipleObjects and WaitForMultipleObjectsEx procedures;
ThreadID - one more descriptor of the command stream as a system object of the kernel level (kernel);
PriorityClass - thread priority class. I recommend especially not to overuse priority classes. Windows' way of serving tasks and threads is imperfect. The slightest increase in the priority of one of the threads often leads to the fact that only this thread is executed, and the rest are forced to stand idle while it has something to do (that is, until it stops itself, or does not request a long asynchronous I / O operation). Classes differ:

- IDLE_PRIORITY_CLASS - lowest, i.e. the thread only works when the system has nothing else to do;
- NORMAL_PRIORITY_CLASS - normal priority class;
- HIGH_PRIORITY_CLASS - increased, i.e. all normal priority threads will run extremely slowly if at least one higher priority thread is running;
- REALTIME_PRIORITY_CLASS - in real time: in general, everything stops, only this thread works, even the system in this mode will not be able to service the mouse if your thread is busy (hmm, is there a "reset" button on your computer?);

ThreadPriority - priority of a thread within its own priority class. Unlike the previous property, this is a softer (and less fatal) way of managing thread priorities. However, the picture is exactly the opposite: quite often, very little can be changed by controlling the priority using this property, if you do not use extreme values.
The following priorities are distinguished:

- THREAD_PRIORITY_IDLE - lowest;
- THREAD_PRIORITY_LOWEST - minimal, allowing the thread to run even when the system is not idle;
- THREAD_PRIORITY_BELOW_NORMAL - reduced compared to normal;
- THREAD_PRIORITY_NORMAL - normal;
- THREAD_PRIORITY_ABOVE_NORMAL - higher than normal,

© 2024 Carl Peeraer

- THREAD_PRIORITY_HIGHEST - increased;
- THREAD_PRIORITY_TIME_CRITICAL — time-critical (this priority is good for a thread that processes events from a timer created in the same thread);

PriorityBoost - Enables or disables for all threads in the system at once (that is, in general in the entire system, according to the Win32 API help), a temporary increase in priority for threads leaving the waiting state for a system event. Usually, this behavior is the default behavior for the system, i.e. temporary increase in priority is allowed;
Data - an arbitrary pointer (or 32-bit number) associated with the stream object;
AutoFree - Set this property to true to provide automatic destroying of thread object when its executing is finished.
OnExecute - "event", which is used to assign code to be executed in a thread, without inheriting other object types from TThread. The termination of this handler means the end of the thread, its Terminated property is set to true. It is no longer possible to run the stream again. If one and the same thread is supposed to be used for multiple execution of tasks of the same type, then it is necessary in the handler to organize an "eternal" cycle of waiting for orders to complete these tasks. Of course, it is necessary to provide for an exit from this "eternal" cycle when some event occurs, otherwise the thread will have to terminate abnormally when the application is about to close;
OnSuspend - an event that is triggered immediately before the suspension of this thread. This event fires in the context of the thread that called the Suspend. In particular, if a thread suspends itself, then the handler is called in its context. When this event is fired, the thread is not yet suspended .;
OnResume - an event that is triggered after the thread resumes. The handler for this event is also called in the context of the command stream that called the Resume method. This means that, firstly, this handler, in principle, will never be called in the context of the resumed thread itself (since the thread cannot resume itself). And, secondly, by the time the handler for this event is triggered, the resumed thread by the time (or in the process) of calling the handler for this event, it is quite possible that it has already been stopped again (or even completed, and even destroyed as an object!);
Synchronize(method) - calls the specified method "synchronously" on the main thread. The Synchronize method should be used within the executing thread itself in order to provide safer execution of sections of code that you want to execute in the context of the main thread. By "main" thread is meant the command thread in which the Applet window object is created (and all forms should be created in the same thread, to avoid confusion). This method provides synchronization by sending (SendMessage) a message to the main window (applet), and as a result, the system "freezes" the sending thread until it returns from the method passed for execution in the main thread. This freezing does not make the thread "Suspended" in the usual sense; it cannot be "renewed" at this moment
SynchronizeEx(method, param)- similar to the previous method, with the only difference that the method must have (the only one, not counting Self) parameter, in the general case - a pointer. What kind of pointer it is is up to the programmer. I added such a method when I felt the need to not just synchronize some non-parameterizable action, and pass it some set of input data to work with. One parameter is, of course, not enough, but, in principle, you can pass anything through a pointer, you just need to declare your data structure and create a pointer type to it.

© 2024 Carl Peeraer

Like many other property and event-driven objects, there is a mirrored **TKOLThread** component for the **TThread object in the MCK**. It is enough to put it on the form, assign an event handler to the OnExecute event, and set up its other properties at the design stage to start building a multi-threaded application.

However, you should understand what command streams are and how to use them correctly so you don't run into big trouble. First of all, you need to clearly understand that the execution of code executed in a thread can be interrupted at any time by the system, at the boundary of any machine instruction (not even a Pascal operator), and control can be transferred to any other thread in the system or in your own application. And the main stream in this is not at all different from other streams.

This means that data (variables, descriptors of system objects) that are shared across multiple threads of execution must be protected. The most unpleasant thing that can happen is the destruction of objects (or freeing the memory of structures) in one thread, while they are being handled in another thread.

My recommendations:

- Protect areas of code responsible for the creation and destruction of shared data using mutexes, semaphores, and critical sections;
- Protect shared objects from premature destruction (using the RefCount property and the RefInc and RefDec methods);
- Synchronize execution of actions that change the state of window objects with the main thread (methods Synchronize and SynchronizeEx);
- Finally, avoid using too many threads in your application. If it is possible to solve the same problem without creating command flows, do it this way.

7.8.1 Thread - Syntax

TThread(unit KOL.pas) ← TObj 92 ← _TObj 92
TThread = object(TObj 92)

type **Thread** = ^TThread;

type **TOnThreadExecute** = function(Sender: PThread): Integer of object;
Event to be called when Execute method is called for TThread 419

Constructors:

function **NewThread**: PThread;

Creates thread object (always suspended 420). After creating, set event OnExecute 421 and perform Resume 420 operation.

```
function NewThreadEx( const Proc: TOnThreadExecute ): PThread; stdcall;
```
Creates thread object, assigns Proc to its OnExecute 421 event and runs it.

```
function NewThreadAutoFree( const Proc: TOnThreadExecute ): PThread;
```
Creates thread object similar to NewThreadEx 420, but freeing automatically when executing of such thread finished. Be sure that a thread is resumed at least to provide its object keeper freeing.

Methods and properties:

```
function Execute: integer; virtual;
```
Executes thread. Do not call this method from another thread! (Even do not call this method at all!) Instead, use Resume 420.
Note also that in contrast to VCL, it is not necessary to create your own descendant object from TThread 419 and override Execute method. In KOL, it is sufficient to create an instance of TThread 419 object (see NewThread 419, NewThreadEx 420, NewThreadAutoFree 420 functions) and assign OnExecute 421 event handler for it.

```
procedure Resume;
```
Continues executing. It is necessary to make call for every nested Suspend 420.

```
procedure Suspend;
```
Suspends thread until it will be resumed 420. Can be called from another thread or from the thread itself.

```
procedure Terminate;
```
Terminates thread.

```
function WaitFor: Integer;
```
Waits (infinitively) until thread will be finished.

```
function WaitForTime( T: DWORD ): Integer;
```
Waits (T milliseconds) until thread will be finished.

```
property Handle: THandle;
```
Thread handle. It is created immediately when object is created (using NewThread 419).

© 2024 Carl Peeraer

```
property Suspended: Boolean;
```
True, if suspended.

```
property Terminated: Boolean;
```
True, if terminated.

```
property ThreadId: DWORD;
```
Thread id.

```
property PriorityClass: Integer;
```
Thread priority class. One of following values: HIGH_PRIORITY_CLASS, IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, REALTIME_PRIORITY_CLASS.

```
property ThreadPriority: Integer;
```
Thread priority value. One of following values: THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_BELOW_NORMAL, THREAD_PRIORITY_HIGHEST, THREAD_PRIORITY_IDLE, THREAD_PRIORITY_LOWEST, THREAD_PRIORITY_NORMAL, THREAD_PRIORITY_TIME_CRITICAL.

```
property Data : Pointer;
```
Custom data pointer. Use it for Your own purpose.

```
property AutoFree: Boolean;
```
Set this property to true to provide automatic destroying of thread object when its executing is finished.

```
property PriorityBoost: Boolean;
```
By default, priority boost is enabled for all threads.

```
procedure Synchronize( Method: TThreadMethod );
```
Call it to execute given method in main thread context. Applet variable must exist for that time.

```
procedure SynchronizeEx( Method: TThreadMethodEx; Param: Pointer );
```
Call it to execute given method in main thread context, with a given parameter. Applet variable must exist for that time. Param must not be nil.

Events:

```
property OnExecute: TOnThreadExecute;
```
Is called, when Execute is starting.

```
property OnSuspend: TObjectMethod;
```


© 2024 Carl Peeraer

Is called, when Suspend is performed.

```
property OnResume: TOnEvent;
```
Is called, when resumed.

Pseudo Streams
Variables:

```
MainThread: PThread;
```

```
PseudoThreadStackSize: DWORD = 1024 * 1024;
```

```
CreatingMainThread: Boolean;
```

Methods:

```
function WaitForSingleObject( hHandle: THandle; dwMilliseconds: DWORD ): DWORD;
stdcall;
```

```
function WaitForMultipleObjects( nCount: DWORD; lpHandles: PHandle; fWaitAll: BOOL;
dwMilliseconds: DWORD ): DWORD; stdcall;
```

7.9 Pseudo Streams

If you chase two hares, you won't catch a single one.
(Russian folk proverb)

Perhaps the biggest drawback of multithreading is the additional barriers to successfully debugging subtle bugs. The operating system does the switching of threads, and it does it according to its own understanding, without asking us when to execute which thread and when to pause. These decisions of the system depend on external factors (work with other applications, the network, the state of the swap file on the disk, etc., etc. - unless the weather on Mars is in this list). The situation is aggravated in the case of multi-core processors, where multiple threads can actually run completely in parallel. Therefore, with multiple executions of the same application with the same data, even if you are trying to reproduce the entire sequence of pressed buttons, there is no certainty. that you can accurately reproduce the desired event. This is especially unpleasant if the desired event is a repetition of an error that occurred under the same conditions in your program.

© 2024 Carl Peeraer

The error that can be reproduced is easy to fix. It is often enough to stop the program and go into single-step debugging mode shortly before the situation in which the crash occurred. The bad thing about elusive mistakes is that they are almost impossible to fix as long as they remain elusive. In the case of a multi-threaded application, many errors that are easy to catch in a normal single-threaded case turn into elusive ones.

Sometimes in such cases it helps to write code like this when multithreading is optional. For example, with the conditional compilation symbol, you specify whether to build your application for multi-threaded or single-threaded work. And then the code uses conditional branching, driven by conditional compilation symbols. And depending on whether your conditional compilation symbol is defined or not, the threads are run or not, and in the case of a single thread, all actions in the application are executed sequentially.

Unfortunately, this path is not only difficult from the start (since instead of developing and debugging one application, you actually have to create two different applications and debug them separately), but it is not always useful for debugging purposes. The error that occurs with enviable regularity in a multithreaded case, when multiple threads are disconnected, suddenly disappears.

For KOL, I came up with a replacement for streams with **pseudo-streams**, in which the application does not basically change its behavior, but in fact becomes single-threaded. To turn streams into pseudo-streams, just add the **PSEUDO_THREADS** conditional compilation symbol to the project options and build the application. For each pseudo-thread, except for the main thread (represented by the global variable MainThread), a block of memory is allocated to store the stack. The block size is 1 MB by default, but can be changed by setting the **PseudoThreadStackSize** variable to the desired value.

Pseudo-Threads, like regular streams, can be started (**Resume**), suspended (**Suspend**), and switched. The only difference is that pseudo-thread switching is not managed by the operating system, which now considers the entire application to be single-threaded, but by the main pseudo-thread. Switches now occur automatically in just a few places: in the **Applet.ProcessMessage** method, in the **Sleep** procedure, and in the **WaitForMultipleObjects** and **WaitForSingleObject** functions. Of course, to extend the functionality of these three API functions, if the **PSEUDO_THREADS** symbol is defined, the KOL module declares its own versions of these functions that can call the **MainThread.NextThread** method when the current pseudo-thread has nothing else to do.

Thus, without changing the application code, the multi-threaded application becomes single-threaded. The streams are preserved, but in a somewhat truncated form. For debugging purposes, this model can be extremely useful, as pseudo-streams continue to "emulate" (mostly) the behavior of streams. Although, without observing some rules, such a model may not work. Namely:

- Critical sections should not be used to control exclusive access to shared resources: the thread is now one, and no control will actually be performed. It will be much more useful to use semaphores for the same purpose, for example: they will work successfully for both real threads and pseudo-threads;

© 2024 Carl Peeraer

- You should not use the multimedia timer to organize pseudo-stream switching. If you try to call the MainThread.SwithToThread or NextThread method directly from the multimedia timer event handler, the application will simply break, since the call will actually be made from a really separate thread created by the system for each active multimedia timer. If this action is performed by sending a message (SendMessage), this message will still be processed only in the message handler, i.e. only when the main pseudo-thread receives control, so there is no special sense in such a switch;
- You should not take advantage of the fact that when working with window objects (when creating them) outside the main thread, such windows usually do not appear on the screen, remaining invisible. Or, if you create message handlers in additional threads in order to work with its own window objects in each thread, then in the case of pseudo-threads, when there is only one real thread, this model will most likely not work (only the loop last started).
- And, conversely, moving on to pseudo-threads, remember that in the case of normal threads with window objects, work usually only happens in the main thread. And if in the process of working with pseudo-streams, you start to change the code and directly work with window methods and messages, then this can further prevent the return from pseudo-streams to regular streams.

In fact, the transition to pseudo-streams is not a sufficient condition to ensure that all events occurring in the application are accurately repeated (for example, for debugging purposes). In addition to switching streams by the system, timers, both regular and multimedia, as well as messages from the mouse and keyboard, are still elements of randomness. But at a certain stage of execution, the probability of a repetition of events increases significantly, which means that the chances of localizing the source of the error increase. And in principle, it becomes possible to log all events affecting the operation of the application, and then, on subsequent launches, reproduce them one-to-one. But you will have to do it with your own code.

7.10 Action and ActionList

Use action objects, in conjunction with action lists, to centralize the response to user commands (actions).
Use AddControl, AddMenuItem, AddToolbarButton methods to link controls to an action.
See also **TActionList**.

TActionList maintains a list of actions used with components and controls, such as menu items and buttons.
Action lists are used, in conjunction with actions, to centralize the response to user commands (actions).
Write an OnUpdateActions handler to update actions state.
Created using function **NewActionList**.
See also **TAction**.

© 2024 Carl Peeraer

TAction and TActionList Constructors:

function **NewAction**
function **NewActionList**: Action list constructor. AOwner - owner form.

Properties and Methods:

Caption: Text caption.
Hint: Hint (tooltip). Currently used for toolbar buttons only.
Checked: Checked state.
Enabled: Enabled state.
Visible: Visible state.
HelpContext: Help context.
Accelerator: Accelerator for menu items.
Actions: Access to actions in the list.
Count: Number of actions in the list.
Destroy
LinkControl: Add a link to a TControl or descendant control.
LinkMenuItem: Add a link to a menu item.
LinkToolbarButton: Add a link to a toolbar button.
Execute: Executes a OnExecute event handler.
Add: Add a new action to the list. Returns pointer to action object.
Delete: Delete action by index from list.
Clear: Clear all actions in the list.

Events:

OnExecute: This event is executed when user clicks on a linked object or Execute method was called.
OnUpdateActions: Event handler to update actions state. This event is called each time when application goes in the idle state (no messages in the queue).

7.10.1 Action and ActionList - Syntax

TAction = object(TObj 92)
Use action objects, in conjunction with action lists, to centralize the response to user commands (actions).
Use AddControl, AddMenuItem, AddToolbarButton methods to link controls to an action.
See also TActionList 425.

TActionList = object(TObj 92)
TActionList maintains a list of actions used with components and controls, such as menu items and buttons.

© 2024 Carl Peeraer

Action lists are used, in conjunction with actions, to centralize the response to user commands (actions).
Write an OnUpdateActions 428 handler to update actions state.
Created using function NewActionList 426.
See also TAction 425.

```
type PControlRec = ^TControlRec;

type TOnUpdateCtrlEvent = procedure(Sender: PControlRec) of object;

type TCtrlKind = (ckControl, ckMenu, ckToolbar);

type TControlRec = record
    Ctrl: PObj;
    CtrlKind: TCtrlKind 426;
    ItemID: integer;
    UpdateProc: TOnUpdateCtrlEvent 426;
end;

PAction = ^TAction;

PActionList = ^TActionList;
```

Constructors:

```
function NewAction(const ACaption, AHint: KOLString; AOnExecute: TOnEvent):
PAction;
```

```
function NewActionList(AOwner: PControl): PActionList;
```
Action list constructor. AOwner - owner form.

TAction and TActionList properties

```
property Caption: KOLString;
```
Text caption.

```
property Hint: KOLString;
```
Hint (tooltip). Currently used for toolbar buttons only.

```
property Checked: boolean;
```
Checked state.

```
property Enabled: boolean;
```


© 2024 Carl Peeraer

Enabled state.

`property` **`Visible`**`: boolean;`
Visible state.

`property` **`HelpContext`**`: integer;`
Help context.

`property` **`Accelerator`**`: TMenuAccelerator 386;`
Accelerator for menu items.

`property` **`Actions`**`[Idx: integer]: PAction 426;`
Access to actions in the list.

`property` **`Count`**`: integer;`
Number of actions in the list..

TAction and TActionList methods

`destructor` **`Destroy`**`; virtual;`

`procedure` **`LinkControl`**`(Ctrl: PControl);`
Add a link to a TControl or descendant control.

`procedure` **`LinkMenuItem`**`(Menu: PMenu; MenuItemIdx: integer);`
Add a link to a menu item.

`procedure` **`LinkToolbarButton`**`(Toolbar: PControl; ButtonIdx: integer);`
Add a link to a toolbar button.

`procedure` **`Execute`**`;`
Executes a OnExecute 428 event handler.

`function` **`Add`**`(const ACaption, AHint: KOLString; OnExecute 428: TOnEvent): PAction 426;`
Add a new action to the list. Returns pointer to action object.

`procedure` **`Delete`**`(Idx: integer);`
Delete action by index from list.

© 2024 Carl Peeraer

```
procedure Clear;
```
Clear all actions in the list.

TAction and TActionList events

```
property OnExecute: TOnEvent;
```
This event is executed when user clicks on a linked object or Execute method was called.

```
property OnUpdateActions: TOnEvent;
```
Event handler to update actions state. This event is called each time when application goes in the idle state (no messages in the queue).

© 2024 Carl Peeraer

KOL Extensions

In this chapter, I will try to provide an overview of KOL extensions. Many of them can be found in the archives on the main KOL WEB site (http://f0460945.xsph.ru/), others on some other KOL sites.

8 KOL Extensions

In addition to the **KOL.pas** file itself, the **KOLadd.pas** file also belongs to the main delivery set, into which minor objects are taken out. As I mentioned above in the text, the main reason for moving some part of the code from **KOL.pas** into an additional module is the need to save lines in the main file of the **KOL.pas** library. The fact is that when the number of lines reaches 65536, the Delphi debugger refuses to work normally. Apparently, this is due to the fact that double-byte unsigned numbers are used to store line numbers in debug information. If not for this circumstance, I would gladly leave all this code in one module, making it easier for myself to maintain and saving 56 bytes in the resulting application.

The content of the **KOLadd.pas** file is mainly described above. When talking about the objects that are defined in this file, I have always mentioned this circumstance. This chapter is not about them.

Literally from the very publication of KOL, there were many programmers on the World Wide Web who contributed to its development. Not only by fixing errors and reporting any inaccuracies noticed, but also by creating additional objects, visual controls, adapting existing VCL components, translating code from C ++. I have also done sometimes similar work, extending the capabilities of the library as needed, and sometimes for the purpose of demonstrating how such extensions should be performed. The KOL library is actually the result of the collective work of many people. As a result, KOL in its capabilities not only approached, but in some areas surpassed the VCL library in its capabilities.

Often there is more than one implementation of the same functionality for KOL, done by different authors at different times, and often independently of each other. Most often, because the existing implementation for some reason did not suit the new author, and the new code is either larger or smaller, it may slightly differ in the set of implemented features and in the quality of execution. Some projects were never brought to a state of completion, and were abandoned by the authors (but this does not happen so often). As a result, the developer has the opportunity to choose the extension implementation that suits him best.

In this chapter, I will try to provide an overview of such extensions. Many of them can be found in the archives on the main KOL WEB site (http://f0460945.xsph.ru/), others on some other KOL sites.

- Exception Handling 432
  - Exception Handling - Syntax 434
- Floating Point Math 436
- Complex Numbers 436
- Dialogues 437
  - Font selection 437
  - Find and replace dialog 437
  - System dialogue "About the program" 437
- Printing and Preparing Reports 437
  - Dialogs for choosing a printer and printing settings. 438

 © 2024 Carl Peeraer

  - Printing reports 438
- Working with Databases 439
  - KOLEDB 439
  - KOLODBC 440
  - KOLIB 441
  - KOLSQLite 441
  - Working with DBF files and other databases 441
- Graphics Extensions 441
  - Metafiles WMF, EMF 442
  - Metafiles - Syntax 442
  - JPEG images 443
  - GIF Images, GIFShow, AniShow 444
  - KOLGraphic Library 446
  - Using GDI + (KOLGdiPlus) 446
  - Other image formats 447
  - Additional utilities for working with graphics 447
  - Open GL: KOLOGL12 and OpenGLContext modules 447
- Sound and Video
  - KOLMediaPlayer 447
  - KOLMediaPlayer - Syntax 449
  - PlaySoundXXXX 459
  - KOLMP3 460
  - Other means for working with sound 460
- Working with Archives
  - TCabFile 460
  - TCabFile - Syntax 460
  - KOLZLib 462
  - KOL_UnZip 462
  - KOLZip 462
  - DIUCL 462
  - KOLmdvLZH 463
- Cryptography
  - TwoFish 463
  - KOLMD5 463
  - KOLAES 463
  - KOLCryptoLib 463
- ActiveX 463
  - Active Script 464
- OLE and DDE
  - KOL DDE 464
  - Drag-n-Drop 464
- NET 464
  - Sockets and protocols 465
  - Working with ports 465
  - CGI 466
- System Utilities 466

- NT services 466
- Control Panel Applet (CPL) 467
- Writing your own driver 467
- NT Privilege Management 467

- Other Useful Extensions
  - Working with shortcuts, registering file extensions 467
  - Sharing memory between applications 467
  - Saving and restoring form properties 467
  - Additional buttons on the title bar 468
  - Macroassembly in memory (PC Asm) 468
  - Collapse Virtual Machine 469
  - FormCompact Property 470
- Additional Visual Objects 470
  - Progress bar 470
  - Track bar (marked ruler) 471
  - Header (tables) 471
  - Font selection 471
  - Color selection 471
  - Disk selection 471
  - Entering the path to a directory 472
  - Selecting a file name filter 472
  - List of files and directories 472
  - IP Input 472
  - Calendar and date and / or time selection 472
  - Double List 472
  - Two-position button (up-down) 473
  - Button, non-rectangular 473
  - Extended panel 473
  - Label with image 473
  - Separator 473
  - Table 473
  - Syntax highlighting 473
  - GRush Controls 474
  - Other additional visual elements 476
  - Tooltips 477
- XP Themes 477
- Extensions of MCK itself
  - Improved font customization 479
  - Alternative component icons 479

8.1 Exception Handling

As a side note, we are not interested in the try finally end block in this chapter. The finally processing block does not require the use of any additional code, classes, and all that is needed to use it is in the System.pas system module (included in the way used by each module in a way

© 2024 Carl Peeraer

transparent to the programmer). There are no restrictions in KOL to use it. It will be about blocks of the try except end type, and in particular about the ability to recognize what kind of exception occurred by analyzing the code of the raised exception.

In the Delphi VCL, exception handling requires working with classes because all exception objects inherit from the Exception class, which is a direct inheritor of the TObject base class. Classes are not used in the KOL library. The very fact of adding classes does not increase the application by much (about 2.5K), but the exceptions are described in the SysUtils module, the connection of which already adds more than 20K to the weight of the finished program. Therefore, in order to enable KOL programs to handle exceptions without inflating their size too much, I added the **err.pas** module at the time.

This module contains a KOL-specific definition of the Exception class. Moreover, unlike the Delphi standard, you do not need to use inheritance to use it. The rule here is "do not inherit unless strictly necessary." In accordance with the same rule, KOL organizes graphic tools for drawing a canvas (brush, pencil, font - in one object type), basic visual objects (TControl). As already mentioned, creating a new inheritor will add at least one more virtual method table (vmt) to the application's weight, 4 bytes for each virtual method that exists in the class and all its ancestors in the inheritance hierarchy.

That is, in order to raise its own exceptions in the KOL application, there is no need to inherit its class from Exception. It is enough to write:

```
raise Exception.Create (e_Custom, 'some error message here');
```

This option is useful if you just need to distinguish your software exceptions from standard system exceptions, which include, for example, division by zero (e_ZeroDivide).

If your exceptions also need to be classified by type, then the following exception constructor will do:

```
raise Exception.CreateCustom (my_code, 'some error message');
```

When constructing an object of type Exception, the number my_code is assigned to the **ErrorCode** 434 property, and then can be parsed in the exception handling block as usual. For example:

```
try
 ... code that can lead to an exception
except on E: Exception do
 if E.ErrorCode = my_code then…
end;
```

Of course, when there are many such codes, it may be more convenient to use the multiple choice construct case (which, by the way, is impossible to distinguish between the exception classes in the VCL, inherited for each of its kind of exceptions).

The **err** module is in the ***https://www.artwerp.be/kol/kol-mck-master_3.23.zip*** archive, along with the **kolmath** and **Cplxmath** modules discussed below.

© 2024 Carl Peeraer

8.1.1 Exception Handling - Syntax

Exception(unit err.pas) 🡐 TObject
Exception = class(TObject)

Exception class. In KOL, there is a single exception class is used. Instead of inheriting new exception classes from this ancestor, an instance of the same Exception class should be used. The difference is only in Code 434 property, which contains a kind of exception.

```
type TError =( e_Abort, e_Heap, e_OutOfMem, e_InOut, e_External, e_Int, e_DivBy0,
e_Range, e_IntOverflow, e_Math, e_Math_InvalidArgument, e_InvalidOp, e_ZeroDivide,
e_Overflow, e_Underflow, e_InvalidPointer, e_InvalidCast, e_Convert,
e_AccessViolation, e_Privilege, e_StackOverflow, e_CtrlC, e_Variant, e_PropReadOnly,
e_PropWriteOnly, e_Assertion, e_Abstract, e_IntfCast, e_InvalidContainer,
e_InvalidInsert, e_Package, e_Win32, e_SafeCall, e_License, e_Custom, e_Com, e_Ole,
e_Registry );
Main error codes. These are to determine which exception occure. You can use
e_Custom code for your own exceptions.
```

type **Exception** = class(TObject)

Exception class. In KOL, there is a single exception class is used. Instead of inheriting new exception classes from this ancestor, an instance of the same Exception class should be used. The difference is only in Code property, which contains a kind of exception.

Exception properties

property **Message**: KOLString;

Text string, containing descriptive message about the exception.

property **Code**: TError 434;

Main exception code. This property can be used to determine, which exception occur.

property **ErrorCode**: DWORD;

This code is to detailize error. For Code 434 = e_InOut, ErrorCode contains more detail description of input/output error. For e_Custom, You can assign it to any value You want.

property **ExceptionRecord**: PExceptionRecord;

This property is only for e_External exception.

property **Data**: Pointer;

Custom defined pointer. Use it in your custom exceptions.

© 2024 Carl Peeraer

Exception methods

`constructor` **`Create`**`( ACode:` [TError](434)`; const Msg: string );`
Use this constructor to raise exception, which does not require of argument formatting.

`constructor` **`CreateFmt`**`( ACode:` [TError](434)`; const Msg: string; const Args: array of const );`
Use this constructor to raise an exception with formatted [Message](434) string. Take into attention, that Format procedure defined in KOL, uses API wvsprintf function, which can understand a restricted set of format specifications.

`constructor` **`CreateCustom`**`( AError: DWORD; const Msg: String );`
Use this constructor to create e_Custom exception and to assign AError to its [ErrorCode](434) property.

`constructor` **`CreateCustomFmt`**`( AError: DWORD; const Msg: String; const Args: array of const );`
Use this constructor to create e_Custom exception with formatted message string and to assign AError to its [ErrorCode](434) property.

`constructor` **`CreateResFmt`**`( ACode:` [TError](434)`; Ident: Integer; const Args: array of const );`

`destructor` **`Destroy`**`; override;`
destructor

Exception events

`property` **`OnDestroy`**`: TDestroyException;`
This event is to allow to do something when custom Exception is released.

© 2024 Carl Peeraer

Exception tasks

With err unit, it is possible to use all capabilities of Delphi exception handling almost in the same way as usual. The difference only in that the single exception class should be used. To determine which exception occure, use property Code 434. So, code to handle exception can be written like follow:

```
try
...
except on E: Exception do
  case E.Code of
  e_DivBy0: HandleDivideByZero;
  e_Overflow: HandleOverflow;
  ...
  end;
end;
```

To raise an error, create an instance of Exception class object, but pass a Code 434 to its constructor:

```
var E: Exception;
...
E := Exception.Create( e_Custom, 'My custom exception' );
E.ErrorCode := MY_MAGIC_CODE_FOR_CUSTOM_EXCEPTION;
raise E;
```

8.2 Floating Point Math

Exactly for the same reason as in the previous paragraph, namely, out of the desire to abandon the connection of the SysUtils module, the **kolmath.pas** module was created for KOL. In many ways, this module repeats the contents of the **math.pas** module, but provides all its capabilities without unnecessarily burdening * the application code. In addition to the general set of functions from the standard math.pas module, a number of useful constants have also been added to **kolmath** (for example, MinSingle, MaxSingle, MinDouble, MaxDouble, MinExtended, ... MaxComp) and functions (EAbs, EMin, EMax, ESign, iMin, iMax , iSign, IsPowerOf2, Low0, Low1, count_1_bits_in_byte, count_1_bits_in_dword). The IntPower function is present in the KOL.pas module itself, so it is commented out in kolmath. For a more detailed study of the contents of the kolmath module, I suggest that you familiarize yourself with its source code.

The kolmath module, like the previous one, is located in the ***https://www.artwerp.be/kol/kol-mck-master_3.23.zip*** archive.

8.3 Complex Numbers

Once upon a time I received such a request from one of the KOL users: to help in writing a set of functions for more convenient work with complex mathematics. If possible, I granted the request, and as a result, the **Cplxmath** module appeared. In fact, this is an add-on over the

© 2024 Carl Peeraer

komath module, with a description of mathematical operations on complex numbers, represented by a pair of floating point numbers. The Complex type is described as

```
Complex = record Re, Im: double end;
```

Of course, writing code that converts mathematical operations into function calls makes the code significantly less readable than using operators, so using the free pascal compiler (in which the class of complex numbers is already implemented through operator redefinition) to work with complex arithmetic will be more productive.

The **Cplxmath** module is in the***https://www.artwerp.be/kol/kol-mck-master_3.23.zip*** archive, along with the **err** and **kolmath** modules.

8.4 Dialogues

Initially, only the most essential dialogs were included in KOL. As needed, the developers added those that they needed for their work. You can download them and use them if needed.

8.4.1 Font selection

The dialog for choosing the **TMHFontDialog** font was written by Dmitry Zharov (nickname Gandalf). See the **MHFontDialog.zip** archive in the Dialogues section of the main site. In addition, the choice of the font name can be done through the combo box. This can be done using a simple combo list filled with API calls to list the fonts installed on the system. Or, in order not to reinvent the wheel, use the **TFontCombo** component from Bohuslav Brandys (Poland) - from his package of "improved combi-lists" **enchcombos.zip**.

8.4.2 Find and replace dialog

The find / replace string feature can come in handy, for example, if you are implementing your own notepad type application. Use in this case to implement this dialog function from the **MHFindReplaceDialog** archive, as you can judge from the prefix - this is also the development of Dmitry Zharov aka Gandalf.

8.4.3 System dialogue "About the program"

Similarly, to display a beautiful system dialog "About", in case you are not satisfied with a simple call to the **Messageboxes** 86 function, there is an **MHAboutDialog** dialog by the same author as above.

8.5 Printing and Preparing Reports

Printing documents in Windows is not as trivial as it might seem at first glance. To do this, you must, at a minimum, provide in your application the ability for the user to select a printing device (printer), configure it (select the paper size and orientation, print quality, other options specific to this printer, for example, color or black and white printing) ... In addition, the application must

© 2024 Carl Peeraer

be able to print the document regardless of the technical characteristics of the device, such as resolution, for example.

This all looks a little tricky to program if you only use API calls directly. Not to mention the fact that when programming something tied to hardware, it is always better to use well tested code (in different conditions and on different equipment) than your own, well tested only for one or two configurations that you personally encountered ... As I remember now, my programs had several times problems with printing on unprecedented printing equipment - only due to the fact that at one time, when writing code, I could not foresee some deviations in the parameters of the device, such that I do not have was close at hand for testing the moment of writing the code.

8.5.1 Dialogs for choosing a printer and printing settings.

The first part of the task will help you to complete the dialogs for selecting the TPrintDlg printer and setting the **TPageSetupDialog** printer from Boguslav Brandys. The very same printing can be done more conveniently than directly through the API, if you use the **TPrinter** object of the same author, or **TMHPrinter** - from Dmitry Zharov. Both of these print objects provide a canvas for drawing and a set of properties and methods that are convenient for organizing the printing process.

8.5.2 Printing reports

An even more convenient printing interface is provided by the KOLReport printing package (this is my development, using one of the above modules - by choice - to perform low-level printing). In fact, this module is ported from the **NormalReport** component set, which I developed for the VCL.

The main difference between this package and all kinds of XXXXXReport, usually used by Delphi programmers, is its minimal visibility. For a programmer who is faced with printing time after time (that is, extremely rarely, like me), a thorough study of any reporting system, in which visuality is brought to deadly perfection, presents certain difficulties. Every time you need to prepare a report, you have to remember where what properties are and how to change them so that everything becomes the way you need it - this is not a task for the faint of heart. This is especially annoying if you really rarely have to deal with printing, once every couple of months or every six months.

So I took a different path than providing a fully visual design-time interface. In addition, no matter how super-sophisticated this interface is, it will still have some limitations. And then go and prove to the user that his order with the placement of the logo here in this place on the sheet is not feasible, or that this method of transferring cells to another line is not supported by the print component, and he must accept it.
The final result is actually more convenient for programmers who are great at writing code. It is much easier for a programmer to write loops and use unambiguously understandable language instructions to check all the necessary conditions than to set up a complex visual component, which also needs to be figured out beforehand.

© 2024 Carl Peeraer

So, working with **KOLReport** is structured as follows. First, you are on the form (for this you can select a separate form, or several separate forms), the cells and rows of cells of the report are "drawn". In general, any visual control, for example, a label or a panel, can work as a cell. But in order for cells to have a white background and can have custom borders without further complicating their existence, the same package implements cell constructors such as **TReportLabel** and **TBand**. Of course, they have MCK mirrors so that the report can be designed semi-visually using the MCK environment.

Those cells that are used to accommodate fixed information (text and images) can be customized at the design stage. Other cells will get their value just before printing. Of course, it is enough to "draw" one standard ruler, fill it in immediately before issuing it to the report, and reuse it after sending it, creating as many lines in the table as necessary. The process of displaying rulers and individual cells, as well as changing pages, is handled by your code placed in the **OnPrint** event handler. You have a wide range of methods for displaying cells, formatting them, managing the grouping of information on a sheet, and so on. But the most important tool remains the programming language.

Fulfilling the most incredible user requirements is a breeze with this reporting system. And, most importantly, such code is easy to modify in the future and is not at all difficult to maintain (as I have already had the opportunity to see from my own experience). For programmers developing projects using KOL, the argument of the compactness of the resulting application will not be superfluous. For example, the compiled application takes up no more than 40KB in compiled form.

8.6 Working with Databases

When I thought about KOL, I never thought that a small program made with this library would ever be able to work with databases. Typically, a minimal Delphi application that used the BDE engine to communicate with databases would start at 600K in size. In order to connect to the server, to perform simple work, it was required to connect to the application modules of enormous scale, the inner contents of which remained a Great and Incomprehensible Secret for mere mortals. The most important secret for me still remains: what else is being done there compared to the engines made for KOL that the programs turn out to be so bloated.

8.6.1 KOLEDB

I decided to make the first "real" engine for working with a database through the OLE DB interface. The main (and almost the only) source that turned me on this path was the documentation from Microsoft, namely MSDN. Based on what I read there, I decided that this is the most progressive of all interfaces. It is he who is used by the modern ADO system as the lower level for communication with the database.

In order to keep the size of the application as small as possible, I decided to sacrifice a few things. Namely: parameters in SQL queries. In reality, the use of parameters is not strictly

© 2024 Carl Peeraer

required, that is, you can do without them perfectly. Moreover, for some time now I have stopped using them altogether. The main reason why I didn't make friends with them is the inconvenience of debugging SQL queries containing such parameters. If the query has no parameters, and all values are inserted as string values, then the query can be copied unchanged into the same Query Analyzer, and run for testing in offline mode (in the sense, separately from the application). For comparison, try doing the same with a query containing more than a dozen parameters.

Note: Several methods can be used to copy a query generated dynamically in the application code as a result of concatenating several (or many) strings. For example, just before executing a query, when it has already been formed, its text can be saved in a debug text file.

The **KOLEDB** package contains a sufficient minimum of tools needed to connect to the database server and get started with it (**TDataSource** objects, **TSession** objects). At this stage, the greatest difficulty is the formation of the connection string: each server needs its own string, with its own set of parameters. And even the syntax for different servers can vary significantly. But an application is usually developed for one type of server, so this difficulty has to be overcome only once.

The **TSession** object allows you to organize transactions, and is the "parent" of all created SQL TQuery objects.

The main workhorse of the package is **TQuery**. It allows you to execute SQL queries, get results, and can work with almost all major data types. To avoid the use of cumbersome and slow variant data types, the properties for accessing data fields are separated by type. For example, to refer to string fields, use the **SField [idx]** and **SFieldByName ['name']** properties, for integer fields - **IField** and **IFieldByName**, and so on. This does not mean, however, that the value of a numeric field can only be accessed through a property whose name begins with the letter corresponding to its type. Access through the SField property is allowed: in this case, the field will be automatically converted to a string, if possible.

In practice, access to BLOB fields remained unimplemented for **KOLEDB** (of the really useful features). I think that when someone needs this module for serious work, then this programmer will not have too many problems to modify it to the desired functionality.

8.6.2 KOLODBC

Another no less (or even more) developed interface for communication with the database was made by me quite recently, at the end of 2005. This time, I needed to communicate with the database through an **ODBC driver**, bypassing ADO - that was the customer's requirement. The project was developed for VCL, respectively, the first version was made specifically for VCL. It was then that, amazed at the unexpected simplicity of the resulting code, I spent another couple of days making this interface available to KOL programmers as well. It is enough to define a conditional compilation symbol in the project options, and the classes become simple object types.

 © 2024 Carl Peeraer

As with OLE DB, there is no parameter support in this interface, and this greatly simplifies the code. The **TODBCDatabase** object serves both for connecting to the database and for managing transactions (there is no separate object for managing sessions). The **TODBCQuery** object is the main workhorse, and can work with almost all data types when retrieving result fields.

A feature of this interface is the availability of the More method, which allows you to easily get several results in one request. Sometimes this feature is very useful in terms of improving query performance. In general, connecting directly through **ODBC drivers**, bypassing other intermediaries like ADO, improves work efficiency, and sometimes quite significantly.

8.6.3 KOLIB

This interface is designed specifically for working with **Interbase**. Package author: Evgeniy Mikhailichenko aka ECM. Actually, I don't know anything more about this interface at the moment. (I'll try to find out, then I'll add it).

8.6.4 KOLSQLite

Component for working with SQLite database, author Boguslav Brandys (Poland).

8.6.5 Working with DBF files and other databases

In addition to the above, for KOL there are also two packages for working with DBF files and one package for working with text tables compatible with the MS Access database.

I. **TdkDBKOL** - the author of **Thaddy de Koning**, allows you to read fields from DBF files directly, bypassing the need to install drivers, solve problems with national encodings, configure connections, data sources, and so on (if only the files themselves).
II. **KOLxBase** - ported for KOL by Dmitry Matveev, also allows you to work with DBF files directly.
III. **StrDB** - by Mike Talcott (USA). This package allows you to work with text tables in files. The format of such files is compatible with the MS Access database (through import and export).

8.7 Graphics Extensions

In addition to basic objects for working with images such as raster (BMP), icons (ICO), the main composition of KOL + KOLadd also contains metafiles (WMF, EMF), which I did not describe above (however, I will try to find a place for them in this section). And on the KOL website you can find objects for working with all major graphic formats: JPG, GIF, PNG, PCX, Targa, TIFF and some others. Here I will only describe some of the packages without going into too much detail.

© 2024 Carl Peeraer

8.7.1 Metafiles WMF, EMF

Metafile support is included in the main KOL package. The **TMetafile** object type is presented in the **KOLadd** module, and allows you to fully work with images of the **WMF** and **EMF** types, namely: load such images into memory, generate new images, and display them on the canvas. Although metafiles are inherently quite different from bitmaps that store images pixel by pixel, the interface of the **TMetafile** object is as close as possible to the interface of a regular single-frame image, which includes **TBitmap**. For example, it also has **Width** and **Height** properties, methods for loading an image from various sources and saving it to an output stream and file 443
.

8.7.1.1 Metafiles - Syntax

TMetafile = object(TObj 92)
Object type to encapsulate metafile image.

```
type PMetafile = ^TMetafile;

type
  TMetafileHeader = packed record
    Key: Longint;
    Handle: SmallInt;
    Box: TSmallRect;
    Inch: Word;
    Reserved: Longint;
    CheckSum: Word;
  end;

const
  WMFKey = Integer($9AC6CDD7);
  WMFWord = $CDD7;
```

Metafile Properties

`property Handle: THandle;`
Returns handle of enchanced metafile.

`property Width: Integer;`
Native width of the metafile.

`property Height: Integer;`
Native height of the metafile.

© 2024 Carl Peeraer

Metafile Methods

```
destructor Destroy; virtual;
```

```
procedure Clear;
```

```
function Empty: Boolean;
```
Returns TRUE if empty

```
function LoadFromStream( Strm: PStream ): Boolean;
```
Loads emf or wmf file format from stream.

```
function LoadFromFile( const Filename: AnsiString ): Boolean;
```
Loads emf or wmf from stream.

```
procedure Draw( DC: HDC; X, Y: Integer );
```
Draws enchanced metafile on DC.

```
procedure StretchDraw( DC: HDC; const R: TRect );
```
Draws enchanced metafile stretched.

```
function NewMetafile: PMetafile;
```
Creates metafile object.

```
function ComputeAldusChecksum(var WMF: TMetafileHeader): Word;
```

```
function SetEnhMetaFileBits(p1: UINT; p2: PAnsiChar): HENHMETAFILE; stdcall;
```

```
function PlayEnhMetaFile(DC: HDC; p2: HENHMETAFILE; const p3: TRect): BOOL; stdcall;
```

8.7.2 JPEG images

Support for working with JPEG images is provided by the **JpegObj** module, the code for which is based on the source code provided by the Independent JPEG Group. Most of the code is precompiled and delivered in the form of object files, so, unfortunately, I cannot 100% guarantee that this code is error-free. Nevertheless, the experience of using this module in my applications shows a rather significant resistance of its code to faulty data.

The technique for working with JPEG images is not much more complicated than with bitmap images (**TBitmap**). In order to load an image, one of the methods of loading from a file or stream is called for the **TJpeg object**. Before an image can be drawn on the canvas or assigned to a bitmap object (TBitmap), a decoding procedure must be performed (for which the DIBNeeded method is explicitly or implicitly called), during which the JPEG image is decoded and restored to an internal TBitmap object.

Either immediately after decoding or before decoding starts, the **ConvertCMYK2RGB** property must be set to TRUE, otherwise, if the JPEG image is CMYK encoded, the colors will be incorrect.

© 2024 Carl Peeraer

However, if you are sure that images in this format will not come across to your application, you can leave this property untouched and save a few kilobytes of code.

In the process of loading and decoding large images (which can be decoded for quite a long time - up to several seconds), it makes sense to use the OnProgress event to display the progress of the loading process (or finished parts of the decoded display), and be able to interrupt the operation before the entire decoding operation is completed. if there is a need for it.

If you need to download not the image itself, but its reduced copy, then the JPEG format has special tools for accelerating the loading of such thumbnails. It is enough to set the required scale to the Scale property (the minimum possible value of jsEighth, which corresponds to 1/8 of the original size).

A TJpeg object allows you to reverse-convert a bitmap to JPEG and save it to a file or stream. To do this, simply assign the object's Bitmap property some other bitmap of the TBitmap type (using the ': =' operation, not the Assign! Method), and then execute the appropriate save method, for example, SavetoFile. Compression is performed immediately before the save is performed - automatically, or it is possible to call the Compress method at the desired moment. In this case, it is also possible to control the degree of compression by assigning a pre-required value to the CompressionQuality property (a number in the range from 1 to 100). When encoding an image, the **OnProgress** event can also be used to indicate progress and possible interruption of the operation.

If the application does not call methods that compress and save the image, then this saves about 30KB of code. In this case, the increase in the size of the application will be about 50K. If both upload and download are used, it is easy to calculate that the growth in the size of the application will be about 80K. By the way, this value for KOL applications cannot be considered negligible. I also cite these figures in order to make it clear that, for example, you should not store graphic images in resources in JPEG format if there are not many of them and they are small: the fat decoder code of such resources will eat up the size savings. Not to mention, JPEG is a lossy compression format, meaning the image is distorted compared to the original.

8.7.3 GIF Images, GIFShow, AniShow

The original **KOLGif** package was based on code from the famous RxLib library. But by dragging and dropping the code, I reduced my work, giving up compression and keeping only decompression. I did this for several reasons.
The first is the licensed purity of the product. As you know, any commercial project must deduct a certain (small, true) amount in favor of the patent owners. Not that I felt sorry for this money (especially not for my own, since I do not deal with commercial products), I just consider such a requirement unlawful, as well as the patent system itself - a brake on progress. / I think it's just unethical to trade in knowledge. Imagine, as an analogy, that a college degree could be bought on the market. /
The second consideration is the amount of source code. The smaller it is, the easier it is to understand, debug, fix all errors. By the way, in the RxLib code such errors, although not fatal,

© 2024 Carl Peeraer

were also found and eliminated by me during the adaptation of the code to the KOL requirements. (Which ones, I can't remember now, rather, these were not errors, but shortcomings that led to incorrect display of some GIF images).
The third is the quality of the compression. More precisely, the inability to control and improve it without the use of special methods. Let me explain what I'm talking about now. As you know, there are various software products that can create GIF images from other types of images. It is known that the result is a completely different size GIF-files, that is, the quality of compression is determined by a rather fine selection of parameters. The quality of compression, for example, provided by the RxLib library code (in turn borrowed from others by the author, as indicated in the library text), leaves much to be desired. And I'm not ready to write my own code that can compete with other GIF compressors and optimizers. It is both more honest and more convenient to give up this possibility right away (why make a code that is obviously the worst known analogs?).
However, if anyone wishes to complete the conversion or make their own compressor for Gif by extending **KOLGif**, I have no particular objection to that.

I note right away that the code of the **KOLGif** module is multivariate, and is compiled differently depending on the conditional compilation symbols declared in the project options.
Namely, it may or may not use the Animation module (symbol USE_ANIMATION_OBJ). If used, the TGif object does not inherit directly from **TObj**, but from **TAnimation**. The only purpose of such a change in the position in the type hierarchy is the ability to use the universal visual object TAniShow, capable of displaying (practically - with the same code) not only GIF images, but also frames compiled programmatically from TBitmap bitmaps and, in general, any objects inherited from **TAnimation**. For example, there is my implementation of TFlic, a successor from it, which provides decoding of AutoDesk animation files (FLIC files).
Another acceptable variation is the ability to abandon your own (converted from RxLib) GIF decoder, and use the **KOLGraphicCompression** module from the **KOLGraphic** package (**USE_KOLGRAPHIC** conditional compilation symbol) for this purpose. Experiments have shown that the difference can be observed only on a very small number of GIF images, moreover, most often, encoded in clear violations of the GIF format conventions, or simply flawed. Moreover, only one of these two compressors does not always win. As for the size of the final code added to the application, KOLGraphic, as a more versatile package, adds a lot more to the weight of the program. It should be used for GIF decompression only if this package is already used in the application to work with one or several other graphic formats:

I. **GIF decoder (TGifDecoder)**. The TGifDecoder object can be used on its own if all you need is to decode individual frames of a GIF animation, decode the first frame, or decode a single frame of a non-animated GIF image. In this case, a minimum of compiled code will be added to the application. I hope you can figure out its properties and methods on your own, they are quite simple, provided with comments and do not require special explanations.
II. **Frame object (TGifFrame)**. This object is a helper object, and is used in the TGif object implementation to represent individual frames of an animated GIF image. You have to resort to its properties very rarely if you want to analyze individual frames or provide some information for the user on them.
III. **Main object (TGif)**. The TGif object type provides all the basic functionality for working with animated GIFs. For example, it can draw the current frame (methods Draw, DrawTransp,

© 2024 Carl Peeraer

DrawTransparent, StretchDraw, StretchDrawTransp, StretchDrawTransparent), control frame switching (the Frame property), as well as load and decode the entire GIF image. But it does not provide time tracking to animate the image correctly.

IV. **Visual animation of the Gif image in the window (TGifShow)**. The TGifShow visual object is inherited from the main visual object of the KOL package - from TControl. It switches frames on its own, updating the animation on screen whenever possible when its Animate property is set to TRUE. / As I already said, instead of it there is an opportunity to use the TAniShow object from the Animation module, while other formats of animated images become available.

8.7.4 KOLGraphic Library

Through the efforts of Dmitry aka Dimaxx for the KOL library, Michael Lishke's **KOLGraphic library** was converted. This library provides support for a wide variety of graphics formats. Here is just a list of them:

- Silicon Graphic Images (* .bw, * .rgb, * .rgba, * .sgi)
- Autodesk Images (* .cel, * .pic)
- 1,8,16 & 24 (32) bits per pixel TIFF Images (* .tif, * .tiff)
- Enhanced PostScript images (* .eps)
- Targa Images (* .tga; * .vst; * .icb; * .vda; * .win)
- ZSoft PCX Images (* .pcx; * .dcx, * .pcc; * .scr)
- Kodak Photo CD Images (* .pcd)
- Portable Map Graphic Images (* .ppm, * .pgm, * .pbm)
- Dr. Halo Images (* .cut + * .pal)
- CompuServe GIF Images (* .gif)
- RLA Images (* .rla, * .rpf)
- Photoshop Images (* .psd, * .pdd)
- Paint Shop Pro Images file version 3 & 4 (* .psp)
- Portable Network Graphic Images (* .png)

If you are going to write another program for viewing images with support for a large number of graphic formats, then this library is very useful. At least in my Zoomer program, I tried it out with success. Not all declared formats are supported, though not 100% (for example, TIFF files encoded in FAX3 format are not displayed). But in any case, there is fish for fishlessness and cancer.

8.7.5 Using GDI + (KOLGdiPlus)

The orientation towards the use of the **new graphics library GDI** + is very promising. Thanks to **Thaddy de Koning** (Netherlands), who adapted Dave Jewell's code for us, programmers using KOL already have the opportunity to try out the capabilities of this library. As soon as the old operating systems leave the arena of personal computers, every user of the Windows operating system will have the gdiplus.dll library preinstalled, and it will be a sin not to take advantage of its capabilities. (But when will this happen? - people are still in no hurry to part with even Windows 95 ...).

© 2024 Carl Peeraer

8.7.6 Other image formats

In addition to the above, on the main site you can also find **KOLPcx**, **KOLTga** packages. As their names suggest, they handle (loading and displaying only, though) the PCX and Targa formats. However, if you need to get not only these two additional formats, then it is more profitable for working with PCX files, nevertheless, to use the **KOLGraphic library**, which also has support for this format.

8.7.7 Additional utilities for working with graphics

I. **SmoothDIB** - This package provides a TSmoothDIB object that provides a number of additional capabilities for drawing on DIB bitmaps. Namely, drawing lines and other shapes from line segments with so-called anti-aliasing. The peculiarity is that the line width is set as a floating point number, and the drawing itself is done by its own code through the Scanline property of the TBitmap object.
II. **KOLPlotter** - A package for drawing graphs of mathematical functions. Author: Alexander Shakhaylo.
III. **Sprite Utils** - Sprite engine. There is support for Direct Draw. Author: miek.
IV. **BIS** - Compress images containing large areas filled with a single color. Author: miek.
V. **KOLjanFx** - A set of additional graphic effects. Conversion for KOL: Dimaxx.

8.7.8 Open GL: KOLOGL12 and OpenGLContext modules

The **KOLOGL12** module not only provides all the Open GL interface declarations, you can also use the standard opengl module from the compiler distribution for this. Its peculiarity is that it makes it as economical as possible for the size of the application, which is especially important for programming with the KOL library. The author of the module, Vyacheslav Gavrik, has found a great way to bypass the need for mandatory connection of all functions of the Open GL library interface, and at the same time maintain a high speed of calls to them.

The **OpenGLContext** module was written by me, and is the simplest OOP add-on over many functions of the Open GL library API. It provides a Context object through which most of the work is done, as well as a set of object types for representing textures, vertex arrays, lights, materials, and more.

8.8 Sound and Video

8.8.1 KOLMediaPlayer

Initially, this object was in the main composition of KOL, but later it was moved to a separate module. The main reason for this decision was to reduce the dependence of applications on the MMSystem module, which contains the necessary definitions and is not required by most applications. Functionally, the **TMediaPlayer** object from this module is not much different from the component with which you may have happened in the VCL. But in KOL it is not visual, and does not automatically provide buttons to control the process of playing music or video. Since

© 2024 Carl Peeraer

the object uses MCI (Media Control Interface) commands, using this object it is possible to play any media file for which all the necessary drivers and decoders (codecs) are installed in the system.

The basic procedure for working with this object is as follows. It is created by the **constructor:**
NewMediaPlayer(filename, wnd) - the optional parameter file (can be an empty string) specifies the file to be played, the wnd parameter (it can also be zero for sound files) passes the object's constructor a handle to the window in which the video image will be played.

High-level properties and methods:

Display - window descriptor for displaying video images;
DisplayRect - rectangle in the Display window for the image;
Width, **Height** - return the own width and height of the open video file;
Filename - allows you to change the name of the file for playback;
DeviceType - returns the type of the open file, it is not necessary to set it, since in the case of a file the system itself is able to determine the data format;
DeviceID - returns the logical device identifier, which can be used in low-level commands;
TimeFormat - sets the time format for operating with the length and position in the stream being played (milliseconds, bytes, frames - for video, samples - for sound, as well as some formatted values, see the text and system documentation for more details);
Length - returns the length of the file (in units specified by the TimeFormat property);
Position - the current position in the file or on the device (it can be changed if the device contains several tracks; to get the starting position of the track, use the TrackStartPosition property);
TrackCount - number of tracks, for devices with multiple tracks;
Track - number of the current track (from 1 to TrackCount);
TrackStartPosition - returns the start position of the current track on the device;
TrackLength - the length of the current track;
Error - error code (if there are no errors, contains 0);
ErrorMessage - text describing the error;
State - the current state of the device (not ready, stopped, playing, recording, searching or rewinding, etc.);
Wait- set this property to TRUE to perform all operations with synchronization (i.e., return from any command will occur only upon completion of this operation). By default, all operations are performed asynchronously, that is, the command is passed to the system for execution, after which control is returned to your code;
OnNotify - this event is triggered when the previous asynchronous operation completes;
Open- method for opening the file specified by the Filename property. If, when creating an object, the Filename parameter pointed to an existing file, then the Open method is called for it automatically;
Alias- a string that, after opening a file, can be used in MCI commands as a device alias. Should be installed before opening the media file and before changing the file name;
Play(start, length) - starts playback of an open file from the specified position, the length parameter sets the portion to play. As the length parameter, you can specify the special value –

© 2024 Carl Peeraer

1, which means that the file will be played until the end of the file. For the start parameter it is also possible to specify a value of –1, which means "from the current position";
StartRecording(FromPos, ToPos) - starts recording on the recorder. Similar to the Play method, the special value –1 can also be used for both parameters, with the same semantic meaning;
Pause - returns TRUE if the device is suspended, it can also be used to programmatically suspend the device by assigning the TRUE value;
Stop - stops playback or recording;
Close - this method closes the device (but does not necessarily stop playback! To stop, you must first put the device in the "pause" or "stopped" state);
Ready - TRUE if the device is "ready" (start recording or playback);
Eject - ejects the media from the device, if such a command is provided for it (for example, for CDAudio);
DoorClose - inverse operation with respect to Eject. The peculiarity of this operation is that an automatic play or open operation can be performed for the media, if this setting is not disabled in the operating system. To prevent auto-opening, use the Insert operation;
Insert - similar to DoorClose, but automatic playback does not work;
Present - Returns TRUE if media is inserted into the device.

Next, I will skip a number of low-level properties and methods, I will only say that they allow you to get more detailed information depending on the type of device and media, control the on and off of individual channels, sound volume, and so on. I will dwell only on the lowest-level commands that allow you to send commands directly to the MCI system:
SendCommand(cmd, flags, buffer) - allows you to send a low-level command to the device, passing, if necessary, some parameters through the buffer, for example.
There is also a version of this method, **asmSendCommand**, which is executed with a deliberate violation of standard communication conventions (it is used in the object implementation itself, so it was convenient to do this for some code optimization).
In addition, you can always use the **mciSendCommand** and **mciSendString** API functions directly, passing the **DeviceID** or Device Alias, respectively, as a parameter identifying the device.

You can download the **KOLMediaPlayer** module here:
https://www.artwerp.be/kol/kolmediaplayer.zip

8.8.1.1 KOLMediaPlayer - Syntax

```
TMediaPlayer = object( TObj )
```


© 2024 Carl Peeraer

MediaPlayer encapsulation object. Can open and play any supported by system multimedia file. (To play wave only, it is possible to use functions PlaySound..., which can also play it from memory and from resource).
Please note, that while debugging, You can get application exception therefore standalone application is working fine. (Such results took place for huge video).

```
type PMediaPlayer = ^TMediaPlayer;
```

```
type TMPState = ( mpNotReady, mpStopped, mpPlaying, mpRecording, mpSeeking, mpPaused, mpOpen );
```
Available states of TMediaPlayer.

```
type TMPDeviceType = ( mmAutoSelect, mmVCR, mmVideodisc, mmOverlay, mmCDAudio, mmDAT, mmScanner, mmAVIVideo, mmDigitalVideo, mmOther, mmWaveAudio, mmSequencer );
```
Available device types of TMediaPlayer.

```
type TMPTimeFormat = ( tfMilliseconds, tfHMS, tfMSF, tfFrames, tfSMPTE24, tfSMPTE25, tfSMPTE30, tfSMPTE30Drop, tfBytes, tfSamples, tfTMSF );
```
Available time formats, used with properties Length and Position.

```
type TMPNotifyValue = (nvSuccessful, nvSuperseded, nvAborted, nvFailure);
```
Available notification flags, which can be passed to TMediaPlayer.OnNotify event handler (if it is set).

```
type TMPOnNotify = procedure( Sender: PMediaPlayer; NotifyValue: TMPNotifyValue ) of object;
```
Event type for TMediaPlayer.OnNotify event.

```
type TPlayOption = ( poLoop, poWait, poNoStopAnotherSound, poNotImportant );
```
Options to play sound. poLoop, when sound is playing back repeatedly until PlaySoundStop called. poWait, if sound is playing synchronously (i.e. control returns to application after the sound event completed). poNoStopAnotherSound means that another sound playing can not be stopped to free resources needed to play requested sound. poNotImportant means that if driver busy, function will return immediately returning False (with no sound playing).

```
type TPlayOptions = set of TPlayOption;
```
Options, available to play sound from memory or resource or to play standard sound event using PlaySoundMemory, PlaySoundResourceID, PlaySoundResourceName, PlaySoundEvent.

```
type TSoundChannel = ( chLeft, chRight );
```
Available sound channels.

```
type TSoundChannels = set of TSoundChannel;
```
Set of available sound channels.

TMediaPlayer Properties

© 2024 Carl Peeraer

`property` **`FileName:`** `String;`

Name of file, containing multimedia, if any (some multimedia devices do not require file, corresponding to device rather then file. Such as mmCDAudio, mmScanner, etc. Use in that case DeviceType property to assign to desired type of multimedia and then open it using Open method).
When new string is assigned to a FileName, previous media is closed and another one is opened automatically.

`property` **`DeviceType:`** `TMPDeviceType;`

Type of multimedia. For opened media, real type is returned. If no multimedia (device or file) opened, it is possible to set DeviceType to desired type before opening multimedia. Use such way for opening devices rather then for opening multimedia, stored in files.

`property` **`DeviceID:`** `Integer;`

Returns DeviceID, corresponded to opened multimedia (0 is returned if no media opened.

`property` **`TimeFormat:`** `TMPTimeFormat;`

Time format, used to set/retrieve information about Length or Position. Please note, that not all formats are supported by all multimedia devices.
Only tfMilliseconds (is set by default) supported by all devices. Following table shows what devices are supporting certain time formats:

tfMilliseconds	All multimedia device types
tfBytes	mmWaveAudio
tfFrames	mmDigitalVideo
tfHMS	(hours, minutes, seconds)> mmVCR (video cassete recorder), mmVideodisc. It is necessary to parse retrieved Length or Position or to prepare value before assigning it to Position using typecast to THMS.
tfMSF	(minutes, seconds, frames)> mmCDAudio, mmVCR. It is necessary to parse value retrieved from Length or Position properties or value to assign to property Position using typecast to TMSF type.
tfSamples	mmWaveAudio
tfSMPTE24, tfSMPTE25, tfSMPTE30, tfSMPTE30DROP (Society of Motion Picture and Television Engineers)	mmVCR, mmSequencer
tfTMSF	(tracks, minutes, seconds, frames) mmVCR

© 2024 Carl Peeraer

`property **Position:** Integer;`
Current position in milliseconds. Even if device contains several tracks, this is the position from starting of first track. To determine position in current Track, subtract TrackStartPosition.

`property **Track:** Integer;`
Current track (from 1 to TrackCount). Has no sence, if tracks are not supported by opened multimedia device, or no tracks present.

`property **TrackCount:** Integer;`
Count of tracks for opened multimedia device. If device does not support tracks, or tracks not present (e.g. there are no tracks found on CD), value 1 is returned by system (but this not a rule to determine if tracks are available).

`property **Length:** Integer;`
Length of multimedia in milliseconds. Even if device has tracks, this the length of entire multimedia.

`property **Display:** HWnd;`
Window to represent animation. It is recommended to create neutral control (e.g. label, or paint box, and assign its TControl.Handle to this property). Has no sense for multimedia, which HasVideo = False (no animation presented).

`property **DisplayRect:** TRect;`
Rectangle in Display window, where animation is shown while playing animation. To restore default value, pass Bottom = Top = 0 and Right = Left = 0.

`property **Error:** Integer;`
Error code. Is set after every operation. If 0, no errors detected. It is also possible to retrieve description string for error using property ErrorMessage.

`property **ErrorMessage:** String;`
Brief description of Error.

`property **State:** TMPState;`
Current state of multimedia.

`property **Pause:** Boolean;`
True, if multimedia currently not playing (or not open). Set this property to True to pause playing, and to False to resume.

`property **Wait:** Boolean;`

© 2024 Carl Peeraer

True, if operations will be performed synchronously (i.e. execution will be continued only after completing operation). If Wait is False (default), control is returned immediately to application, without waiting of completing of operation. It is possible in that case to get notification about finishing of previous operation in OnNotify event handler (if any has been set).

`property` **`TrackStartPosition:`** `Integer;`
Returns given track starting position (in units, specisied by TimeFormat property. E.g., if TimeFormat is set to (default) tfMilliseconds, in milliseconds).

`property` **`TrackLength:`** `Integer;`
Returns given track length (in units, specified by TimeFormat property).

`property` **`Width:`** `Integer;`
Default width of video display (for multimedia, having video animation).

`property` **`Height:`** `Integer;`
Default height of video display (for multimedia, having video animation).

`property` **`Alias:`** `String;`
Alias for opened device. Must be set before opening (before changing FileName).

`property` **`Ready:`** `Boolean;`
True if Device is ready.

`property` **`IsCompoundDevice:`** `Boolean;`
True, if device is compound.

`property` **`HasVideo:`** `Boolean;`
True, if multimedia has videoanimation.

`property` **`HasAudio:`** `Boolean;`
True, if multimedia contains audio.

`property` **`CanEject:`** `Boolean;`
True, if device supports "open door" and "close door" operations.

`property` **`CanPlay:`** `Boolean;`
True, if multimedia can be played (some of deviceces are only for recording, not for playing).

`property` **`CanRecord:`** `Boolean;`
True, if multimedia can be used to record (video or/and audio).

© 2024 Carl Peeraer

`property` **`CanSave`**`: Boolean;`
True, if multimedia device supports saving to a file.

`property` **`Present`**`: Boolean;`
True, if CD or videodisc inserted into device.

`property` **`AudioOn`**`[ Chn: TSoundChannels ]: Boolean;`
Returns True, if given audio channels (both if [chLeft,chRight], any if []) are "on". This property also allows to turn desired channels on and off.

`property` **`VideoOn`**`: Boolean;`
Returns True, if video is "on". Allows to turn video signal on and off.

For "CDAudio" only:

`property` **`CDTrackNotAudio`**`: Boolean;`
True, if current Track is not audio.

For "digitalvideo":

`property` **`DGV_CanFreeze`**`: Boolean;`
True, if can freeze.

`property` **`DGV_CanLock`**`: Boolean;`
True, if can lock.

`property` **`DGV_CanReverse`**`: Boolean;`
True, if can reverse playing.

`property` **`DGV_CanStretchInput`**`: Boolean;`
True, if can stretch input.

`property` **`DGV_CanStretch`**`: Boolean;`
True, if can stretch output.

`property` **`DGV_CanTest`**`: Boolean;`
True, if supports Test.

`property` **`DGV_HasStill`**`: Boolean;`
True, if has still images in video.

`property` **`DGV_MaxWindows`**`: Integer;`
Returns maximum windows supported.

`property` **`DGV_MaxRate`**`: Integer;`

© 2024 Carl Peeraer

Returns maximum possible rate (frames/sec).

`property` **`DGV_MinRate:`** `Integer;`
Returns minimum possible rate (frames/sec).

`property` **`DGV_Speed:`** `Integer;`
Returns speed of digital video as a ratio between the nominal frame rate and the desired frame rate where the nominal frame rate is designated as 1000. Half speed is 500 and double speed is 2000. The allowable speed range is dependent on the device and possibly the file, too.

For AVI only (mmDigitalVideo, AVI-format):

`property` **`AVI_AudioBreaks:`** `Integer;`
Returns the number of times that the audio definitely broke up. (We count one for every time we're about to write some audio data to the driver, and we notice that it's already played all of the data we have).

`property` **`AVI_FramesSkipped:`** `Integer;`
Returns number of frames not drawn during last play. If this number is more than a small fraction of the number of frames that should have been displayed, things aren't looking good.

`property` **`AVI_LastPlaySpeed:`** `Integer;`
Returns a number representing how well the last AVI play worked. A result of 1000 indicates that the AVI sequence took the amount of time to play that it should have; a result of 2000, for instance, would indicate that a 5-second AVI sequence took 10 seconds to play, implying that the audio and video were badly broken up.

For "vcr" (video cassete recorder):

`property` **`VCR_ClockIncrementRate:`** `Integer;`

`property` **`VCR_CanDetectLength:`** `Boolean;`
True, if can detect Length.

`property` **`VCR_CanFreeze:`** `Boolean;`
True, if supports command "freeze".

`property` **`VCR_CanMonitorSources:`** `Boolean;`
True, if can monitor sources.

`property` **`VCR_CanPreRoll:`** `Boolean;`
True, if can preroll.

`property` **`VCR_CanPreview:`** `Boolean;`
True, if can preview.

© 2024 Carl Peeraer

`property` **`VCR_CanReverse:`** `Boolean;`
True, if can play in reverse direction.

`property` **`VCR_CanTest:`** `Boolean;`
True, if can test.

`property` **`VCR_HasClock:`** `Boolean;`
True, if has clock.

`property` **`VCR_HasTimeCode:`** `Boolean;`
True, if has time code.

`property` **`VCR_NumberOfMarks:`** `Integer;`
Returns number of marks.

`property` **`VCR_SeekAccuracy:`** `Integer;`
Returns seek accuracy.

For mmWaveAudio:

`property` **`Wave_AvgBytesPerSecond:`** `Integer;`
Returns current bytes per second used for playing, recording, and saving.

`property` **`Wave_BitsPerSample:`** `Integer;`
Returns current bits per sample used for playing, recording, and saving PCM formatted data.

`property` **`Wave_SamplesPerSecond:`** `Integer;`
Returns current samples per second used for playing, recording, and saving.

TMediaPlayer Methods

`function` **`Open:`** `Boolean;`
Call this method to open device, which is not correspondent to file. For multimedia, stored in file, Open is called automatically when FileName property is changed.
Multimedia is always trying to be open shareable first. If it is not possible, second attempt is made to open multimedia without sharing.

`function` **`Play`**`( StartPos, PlayLength: Integer ): Boolean;`
Call this method to play multimedia. StartPos is relative to starting position of opened multimedia, even if it has tracks. If value passed for StartPos is -1, current position is used to start from.
If -1 passed as PlayLength, multimedia is playing to the end of media.
Note, that after some operation (including Play) current position is moved and it is necessary to pass 0 as StartPos to play multimedia from its starting position again. To provide playing the

© 2024 Carl Peeraer

same multimedia several times, call:

```
with MyMediaPlayer do
Play( 0, -1 );
```

To Play single track, call:

```
with MyMediaPlayer do
begin
  Track := N; // set track to desired number
  Play( TrackStartPosition, TrackLength );
end;
```

procedure **Close;**

Closes multimedia. Later it can be reopened using Open method. Please remember, that if CDAudio (e.g.) is playing, it is not stop playing when Close is called. To stop playing, first perform command `Pause := True;`

procedure **Eject;**

Ejects media from device. It is possible to check first, if this operation is supported by the device - see CanEject.

procedure **DoorClose;**

Backward operation to Eject - inserts media to device. This operation is very easy and does not take in consideration if CD data / audio is playing automatically when media is inserted. To prevent launching CD player or application, defined in autostart.inf file in rootof CD, use Insert method instead.

procedure **DisableAutoPlay;**

Be careful when using this method - this affects user settings such as 'Autoplay CD audio disk' and 'Autorun CD Data disk'. At least do not forget to restore settings later, using RestoreAutoPlay method. When You use Insert method to insert CD into device, DisableAutoPlay also is called, but in that case restoring is made automatically at least when TMediaPlayer object is destroying.

procedure **RestoreAutoPlay;**

Restores settings CD autoplay settings, changed by calling DisableAutoPlay method (which must be called earlier to save settings and change it to disable CD autoplay feature). It is not necessary to call RestoreAutoPlay only in case, when method Insert was used to insert CD into device (but calling it restores settings therefore - so it is possible to restore settings not only when object TMediaPlayer destroyed, but earlier.

procedure **Insert;**

Does the same as DoorClose, but first disables auto play settings, preventing system from running application defined in Autorun.inf (in CD root) or launching CD player application. Such settings will be restored at least when TMediaPlayer object is destroyed, but it is possible to call RestoreAutoPlay earlier (but there is no sence to call it immediately after performing Insert method - at least wait several seconds or start playing track first).

© 2024 Carl Peeraer

`function` **`Save`**`( const aFileName: String ): Boolean;`
Saves multimedia to a file. Check first, if this operation is supported by device.

`function` **`StartRecording`**`( FromPos, ToPos: Integer ): Boolean;`
Starts recording. If FromPos is passed -1, recording is starting from current position. If ToPos is passed -1, recording is continuing up to the end of media.

`function` **`Stop`**`: Boolean;`
Stops playing back or recording.

`function` **`SendCommand`**`( Cmd, Flags: Integer; Buffer: Pointer ): Integer;`
Low level access to a device. To get know how to use it, see sources.

`function` **`NewMediaPlayer`**`( const FileName: String; Window: HWND ): PMediaPlayer;`
Creates TMediaPlayer instance. If FileName is not empty string, file is opening immediately.

`function` **`PlaySoundMemory`**`( Memory: Pointer; Options: TPlayOptions ): Boolean;`
Call it to play sound already stored in memory. (It is possible to preload sound from resource (e.g., using Resurce2Stream function) or to load sound from file.

`function` **`PlaySoundResourceID`**`( Inst, ResID: Integer; Options: TPlayOptions ): Boolean;`
Call it to play sound, stored in resource. It is also possible to stop playing certain sound, asynchronously playing from a resource, using PlaySoundStopResID.
In this implementation, sound is played from memory and always with poWait option turned on (i.e. synchronously).

`function` **`PlaySoundResourceName`**`( Inst: Integer; const ResName: String; Options: TPlayOptions ): Boolean;`
Call it to play sound, stored in (named) resource. It is also possible to stop playing certain sound, asynchronously playing from a resource, using PlaySoundStopResName.
In this implementation, sound is played from memory and always with poWait option turned on (i.e. synchronously).

`function` **`PlaySoundEvent`**`( const EventName: String; Options: TPlayOptions ): Boolean;`
Call it to play standard event sound. E.g., 'SystemAsterisk', 'SystemExclamation', 'SystemExit', 'SystemHand', 'SystemQuestion', 'SystemStart' sounds are defined for all Win32 implementations.

`function` **`PlaySoundFile`**`( const FileName: String; Options: TPlayOptions ): Boolean;`
Call it to play waveform audio file. (This also can be done using TMediaPlayer, but for wide set of audio and video formats).

© 2024 Carl Peeraer

```
function PlaySoundStop: Boolean;
```

Call it to stop playing sounds, which are yet playing (after calling PlaySountXXXXX functions above to play sounds asynchronously).

```
function WaveOutChannels( DevID: Integer ): TSoundChannels;
```

Returns available sound output channels for given wave out device. Pass -1 (or WAVE_MAPPER) to get channels for wave mapper. If only mono output available, [chLeft] is returned.

```
function WaveOutVolume( DevID: Integer; Chn: TSoundChannel; NewValue: Integer ):
Word;
```

Sets volume for given channel. If NewValue = -1 passed, new value is not set. Always returns current volume level for a channel (if successful). Volume varies in diapason 0..65535. If passed value > 65535, low word of NewValue is used to set both chLeft and chRight channels.

TMediaPlayer Events

```
property OnNotify: TMPOnNotify;
```

Called when asynchronous operation completed. (By default property Wait is set to False, so all operations are performed asynchronously, i.e. control is returned to application without of waiting of completion operation). Please note, that system can make several notifications while performing operation. To determine if operation completed, check State property.
E.g., to find where playing is finished, check in OnNotify event handler if State <> mpPlaying.

Though TMediaPlayer works fine with the most of multimedia formats (at least it is tested for WAV, MID, RMI, AVI (video and sound), MP3 (soound), MPG (video and sound)), there are some problems with getting notifications about finishing MP3 playing: when OnNotify is called, State returned is mpPlaying yet. For that case I can advice to check also playing time and compare it with Length of multimedia.

8.8.2 PlaySoundXXXX

In the same module **KOLMediaPlayer** there is also a number of functions for playing sound files in WAV format, allowing you to do this without involving a complex object. Their additional advantage is that they allow you to play sound not only from a file, but also from other sources, including from a buffer in RAM.
PlaySoundMemory(mem, options) - plays sound from a buffer in RAM;
PlaySoundResourceID(instance, resid, options) - plays a sound from a resource by resource identifier in the specified module;
PlaySoundresourceName(instance, resname, options) - plays sound from the resource specified by the resource name;
PlaySoundEvent(event_name, options) - plays one of the standard system sounds (these sounds are associated with the names in the Control Panel, Sound, then see the assignment of sounds to system events and events of installed applications);

© 2024 Carl Peeraer

PlaySoundFile(filaneme, options) - plays sound from the specified file;
PlaySoundStop - stops playback of all sounds started for playback by all previous PlaySoundXXXX functions.

All of these functions ultimately refer to the PlaySound API function.

8.8.3 KOLMP3

If you want your application to be able to play MP3 audio files without depending on the presence of codecs installed on the machine, then this package is what you are looking for. By the way, as an example of an application, the archive contains a miniature MP3-player, in a compressed form only 80.5KB. The author of this treasure is **Thaddy de Koning** (Netherlands). All source codes are provided for study and use.

8.8.4 Other means for working with sound

In addition to the above, there are several more packages for creating more serious applications working with sound. I will list only a few:
FFTrealKOL - adaptation for KOL of one of the fastest implementations of the fast Fourier transform, author **Thaddy de Koning** (Netherlands);
Kol32Audio- a package for professional work with sound, including the creation of special effects. Surely useful if you want to create an editor for sound files. The author is the same;
MultiWave- contains an object for simultaneous playback of several dozen audio streams in **WAV** format. Mixing is done programmatically, the sound is reproduced using **DirectSound**.
KOLMidi- package for working with MIDI by **Thaddy de Koning**. There is a demo of **KOLMidiTest** (in a separate archive).

8.9 Working with Archives

8.9.1 TCabFile

Support for Microsoft cabinet files is available in KOL itself. The KOLadd module defines the TCabFile object type. It is needed in order to perform unpacking from archives of this type. Its interface is small enough to figure it out on your own, so I won't waste a lot of space on its description.

8.9.1.1 TCabFile - Syntax

TCABFile = object(TObj 92)
An object to simplify extracting files from a cabinet (.CAB) files. The only what need to use this object, setupapi.dll. It is provided with all latest versions of Windows.

type **PCABFile** = ^TCABFile;

© 2024 Carl Peeraer

TCABFile Properties

`property **Paths**[ Idx: Integer ]: KOLString;`
A list of CAB-files. It is stored, when constructing function OpenCABFile461 called.

`property **Names**[ Idx: Integer ]: KOLString;`
A list of file names, stored in a sequence of CAB files. To get know, how many files are there, check Count property.

`property **Count**: Integer;`
Number of files stored in a sequence of CAB files.

`property **CurCAB**: Integer;`
Index of current CAB file in a sequence of CAB files. When OnNextCAB462 event is called (if any), CurCAB property is already set to the index of path, what should be provided.

TCABFile Methods

`destructor **Destroy**; virtual;`

`function **Execute**: Boolean;`
Call this method to extract or enumerate files in CAB. For every file, found during executing, event OnFile is called (if assigned).
If the event handler (if any) does not provide full target path fora file to extract to, property TargetPath is applyed (also if it is assigned), or file is extracted to the default directory (usually the same directory there CAB file is located, or current directory - by a decision of the system).
If a sequence of CAB files is used, and not all names for CAB files are provided (absent or represented by a AnsiString '?'), an event OnNextCAB462 is called to obtain the name of the next CAB file.

`function **OpenCABFile**( const APaths: array of AnsiString ): PCABFile;`
This function creates TCABFile460 object, passing a sequence of CAB file names (fully qualified). It is possible not to provide all names here, or pass '?' AnsiString in place of some of those. For such files, either an event OnNextCAB will be called, or (and) user will be prompted to browse file during executing (i.e. Extracting).

© 2024 Carl Peeraer

TCABFile Events

`property` **`OnNextCAB:`** `TOnNextCAB;`
This event is called, when a series of CAB files is needed and not all CAB file names are provided (absent or represented by '?' AnsiString).
If this event is not assigned, the user is prompted to browse file.

`property` **`OnFile:`** `TOnCABFile;`
This event is called for every file found during Execute 461 method.
In an event handler (if any assigned), it is possible to return False to skip file, or to provide another full target path for file to extract it to, then default. If the event is not assigned, all files are extracted either to default directory, or to the directory TargetPath 462, if it is provided.

`property` **`TargetPath:`** `KOLString;`
Optional target directory to place there extracted files.

8.9.2 KOLZLib

This package contains functions for compressing and decompressing data streams and is based on the well-known zlib library. In this implementation, error handling is performed not through raising exceptional states, but through returning error codes from functions, which makes the application somewhat "lighter". Authors of adaptation: Alexey Shuvalov, later update to version zlib 1.1.4 was performed by Dimaxx.

The KOLZlib module has been used successfully, at least in the KOLPng package (see the section on KOL graphical extensions).

8.9.3 KOL_UnZip

Dimaxx also adapted this package for KOL. It is intended only for unpacking Zip archives, and not password-protected ones. But an important plus: no external DLLs are required, which is very important for those who adhere to the principle "I carry everything with me".

8.9.4 KOLZip

This is a free version of the TZip component adapted for KOL by Angus Johnson. Adapted by Boguslav Brandys (Poland). Disadvantages of this package: an external dll is required (it must be downloaded separately), password protection is not supported, there are other restrictions. However, you can create ZIP archives.

8.9.5 DIUCL

This package uses algorithms of the renowned UPX executable packer to compress and decompress data. For use in applications written with KOL, it is enough to add the conditional compilation symbol KOL to the project option. With a fairly good compression ratio, the unpacker is very compact, and if you use unpacking only, the application does not grow much at all. Written by Ralf Junker (Germany).

© 2024 Carl Peeraer

If we are only talking about packaging resources, then it makes sense to use the UPX application wrapper itself. If your application works with its large external files, then this package is indispensable in order to provide storage of such files on external media in a compressed form. Packing data before transmitting it over the network can also make sense.

8.9.6 KOLmdvLZH

A package for supporting the LZH compression and decompression algorithm, by Dmitry Matveev.

8.10 Cryptography

8.10.1 TwoFish

Package provided by: neuron. It is an adaptation for KOL of the well-known TwoFish algorithm. There is an example of use in the archive.

8.10.2 KOLMD5

The package is provided along with examples. Posted by **Thaddy de Koning**. The MD5 algorithm has been implemented.

8.10.3 KOLAES

One of the most robust cryptography methods (it is known to be used in RAR archives, which are practically unbreakable by traditional methods). The package was provided by Dimaxx.

8.10.4 KOLCryptoLib

A large set of cryptography methods (11 pieces) in one bottle. Credit: Dentall (Russia).

8.11 ActiveX

To enable KOL applications to use ActiveX components, the ActiveKOL package was developed, which includes the ActiveKOL.pas modules (a replacement for ActiveX.pas), KOLComObj and a special application Tlb2KOL. These modules are much "lighter" than the original ActiveX module from the Delphi distribution: direct use of such components after installing them in the usual way for the Delphi environment increases the size of the application so much that the meaning of using the KOL library would be lost (360K and higher).

This package allows you to install and use almost any ActiveX component for use with KOL, while keeping the executable file size within reasonable limits. True, sometimes a certain amount of manual work is required after the automatic generation of the interface code by the Tlb2Kol utility. But in general, the package is suitable, at least, for use with Delphi compilers from version

© 2024 Carl Peeraer

5 to version 7. The main site has a fair amount of ready-made adaptations of various ActiveX components, made mainly for Delphi 6.

8.11.1 Active Script

The KOLAxScript package was developed by **Thaddy de Koning** (Netherlands). The package is intended for working with Active Script.

8.12 OLE and DDE

8.12.1 KOL DDE

The KOLDDE package contains a DDE client and server with mirrored classes. Author Alexander Shakhaylo.

8.12.2 Drag-n-Drop

This package provides a drag-and-drop operation from the KOL application to the outside world. If you have encountered this task before, then you probably know that this operation is not programmed as elementary as the reverse *. For its organization, at least, work through OLE-interfaces is required.

The base class (namely, the interface classes are used, otherwise it will not work with OLE) is TDropSource. There are also classes inherited from it **TDropFileSource** and **TDropTextSource**, almost ready for dragging and dropping file objects or text fragments.

Package author: non. As a bonus, there is a **TClipboard** object for working with the clipboard.

8.13 NET

Networking in KOL applications is arguably the most disorganized area on the battlefield to avoid unnecessary code duplication. Especially in terms of tools for connecting over a network using various protocols. With all the variety of choices, it is difficult to call even half of the available packages completed. Nevertheless, you can still choose something **.

© 2024 Carl Peeraer

8.13.1 Sockets and protocols

Here is a short list of available packages, their composition and some features. I could not describe all of them in detail within the framework of this book, even if I wanted to. In general, there is no particular need for this: all packages are supplied in source code, many are provided with comments and sometimes good demo applications, so it will not be difficult to figure it out.

I. **KOLSocket**: This package contains a socket object, along with a visual mirror. As a demo application, an example of working with Telnet. Author Alexander Shakhaylo (Ukraine).
II. **TCPSocket**: The author of this package is Roman Vorobets. The package contains definitions of TCPServer and TCPClient objects, there are MCK mirrors for both objects.
III. **TKOLServerSocket** & **TKOLClientSocket**: Adaptation for KOL (with mirrors for MCK) TServerSocket and TClientSocket components from VCL. Author Alexey Sapronov.
IV. **XSocket**: Packet for connection via TCP / IP sockets, based on Marat Shamshiev's code. There are no visual mirrors. Perhaps the most compact in code size option, albeit somewhat in the Spartan style. The adaptation for KOL was done by Roman Vorobets.
V. **ClientServer**: Another socket option without visual mirrors. There is a demo client-server. Posted by Mike Sevbo.
VI. **KOL IPC Streams**: Objects for working with pipes and milslots * via streams based on the PStream object from KOL. Written by Thaddy de Koning (Nid.)
VII. **Synapse**: Library of network functions. The adaptation for KOL belongs to Bohuslav Brandys (Poland).
VIII. **KOL ICS**: Library of objects for developing network applications (Http, Smtp, Ping, etc.). Author Dmitry Zharov aka Gandalf.
IX. **KOLHttp**: An object for downloading the content of WEB pages from the network via the http protocol. Author Alexander Shakhaylo (Ukr.).
X. **KOLFTP**: Object for uploading files from an FTP server and uploading files to an FTP server. Author Alexander Shakhaylo (Ukr.).

Note: **KOLIndy** packages I deliberately did not include them in the list, since they are not completed and, most likely, will not be completed.

8.13.2 Working with ports

Several packages for working with COM-port and LPT-port from different authors.

I. **ComPort**: Object for working with a COM port, with a visual mirror (plus a bonus in the form of a visual LED control, like a light bulb). Written by Vasily Pivko.
II. **MHComPort**: Another component for working with a COM port. Author Dmitry Zharov aka Gandalf.
III. **ForLPT**: Object for working with LPT port in Windows 9x / ME operating systems. Author Alexander Rabotyagov.

© 2024 Carl Peeraer

8.13.3 CGI

It is quite possible with KOL to create very small CGI server applications. An example of how this is done is available in the WebCountExe application provided with the source code by Andrey Chebanov. (Look should be in the section "Applications" on the main site).

8.14 System Utilities

This section is for "advanced" programmers who write so-called "system" software. To write common applications, the packages offered here are not needed at all. Therefore, feel free to skip this chapter if you are writing regular application programs.

8.14.1 NT services

Windows NT services, unlike conventional applications, work closely with the operating system. Namely, they are called in the address space and in the context of the operating system, while getting some capabilities that are not available to regular applications. The list of services running on your machine can be obtained from the Control Panel, but it is available only for a user with administrator rights. Many important components of the NT operating system itself are organized as services. Or, to put it quite differently: the operating system is made up of components that are services. Thus, making an application a service means that your application becomes a component of the operating system.

In fact, the API for organizing your application as a service is not too difficult. But with the proposed set of objects, this work is so radically easier that not using it is just blasphemy.

So, the KOLService package, by Alexander Shakhaylo (Ukraine). Objects:

- **TServiceContol** - allows you to register a service and manage it (stop, start);
- **TService** - allows you to organize the application itself, in which it is used as a service;
- **TServiceEx**- allows the service to be interactive, that is, to have a visual interface with regular forms and other windows. Why should this object be used instead of TService.

The differences between the process of writing a service and writing a regular application are very small. The most significant of these is that your code, in the case of services, runs in the context of an operating system task, and is styled as a set of event handlers for a TService or TServiceEx object.

But we must not forget that the service actually becomes a component of the operating system. If it malfunctions or freezes, then this will greatly affect the performance of the entire system. In addition, there are practically no tools for debugging the service, except for the formation of your own log files or sound signals. Try, if possible, to debug the algorithms to whom, how to use them inside the service.

© 2024 Carl Peeraer

8.14.2 Control Panel Applet (CPL)

Boguslaw Brandys' CPL Applet (Poland) contains a TCPLApplet object with a visual mirror that allows you to organize your application as a control panel applet. In short, this is a way to put a shortcut to your application in the Control Panel, among other system settings. It is unlikely that you will need to do this for a regular application, but if you write any driver or system utility, then keep this possibility in mind.

8.14.3 Writing your own driver

By the way (continuing the previous paragraph), just the KOLDriver archive can help you write your own device driver (and not only for a mouse pad, but also a serious driver for a real device). This archive contains a code template, on the basis of which it is quite possible to complete the task. The author of the package is **Thaddy de Koning** (Netherlands).

8.14.4 NT Privilege Management

The author of the KOLNTprivileges package is Alexander Shakhaylo (Ukraine). The package, as you understand, is dedicated to managing NT privileges in operating systems starting with NT. As a bonus, there is a Pascal-translated isaapi module.

8.15 Other Useful Extensions

8.15.1 Working with shortcuts, registering file extensions

Some of the functions required to perform the tasks specified in the title of this paragraph have been moved to a separate module Lnk.pas.
CreateLink, **CreateLinkDesc** - allow you to create a shortcut;
ResolveLink, **IsLink2RecycleBin** - help in analyzing the existing label;
FileTypeReg, **FileTypeRegEx**, **FileTypeReg2** - useful for creating an application association with specified file extensions;
FileTypeGetReg - returns information about the current association of a file extension.

Another package, KOLNKDir, is designed to make it easier to locate system directories such as My Documents, Programs, etc. Author Dmitry Zharov aka Gandalf.

8.15.2 Sharing memory between applications

The KOL_ShareMem package makes it easy to share memory between applications. A memory-mapped file is used. The author of the adaptation to KOL is neuron.

8.15.3 Saving and restoring form properties

The KOLFormSave package allows an application to save 5 parameters that set the size and visibility of the form in the registry, and restore them when the application is restarted. Easily

© 2024 Carl Peeraer

expands if more parameters need to be stored. The author of the package is Alexander Shakhaylo (Ukraine).

8.15.4 Additional buttons on the title bar

The KOLGets package allows you to add a button (or several) to the header of a form. Author Alexander Shakhaylo (Ukraine).

8.15.5 Macroassembly in memory (PC Asm)

I created the PCAsm package for my own purposes (and used it in at least one application). Its purpose is very specific: compilation of assembly text into machine code in memory, with the possibility of subsequent execution of this code during the same session of the application. Sounds unusual, doesn't it? Delphi has its own built-in assembler, and it's pretty good. Unfortunately, it doesn't support macros. There are situations (and I just got one) when it is almost impossible to do without macros. In my case, it was about the need to optimize the code to the limit, if it was possible to configure it for a huge combination of all kinds of input parameters, and it was necessary to choose only one such combination, and exclude all condition checks from the code. It's about checking conditions in deeply nested loops, of course.

This assembler supports all major machine instructions IBP PC 486, Pentium, and even MMX instructions. By the way, one of the options is when, in a nested loop, the same operation should be performed, whenever possible, using MMX instructions from the available set. For different sets, code is written that does the same job, but using different instructions, and then this code is compiled with conditional compilation symbols (symbolic variables, if you like), the value of which is set before calling the compiler depending on the current hardware configuration.

From the point of view of reducing the application code, this approach may even be beneficial in the case when the direct writing out of all possible variants of the code and including them all in the final executable module significantly increases the size of the application due to the large number of variants. Although in my case the options were so great that the direct writing of all the algorithm options, depending on the hardware configuration and other input conditions, that I was physically unable to support all these configurations. Thus, the solution to compile the code into memory "on the fly" turned out to be the only suitable tool.

Perhaps one of the disadvantages of this approach is the need to keep part of the code in the form of source text (taken from an external file or from resources). Compilation also takes some time, although I tried to optimize it whenever possible. And the most unpleasant consequence is that in order to test even the fact of compilation of all code variants, it is necessary to perform extensive testing. Otherwise, a compilation error can occur in an application that has already been submitted to the user, and it can be problematic to fix this error. In general, maybe someone will find some other uses for this package.

© 2024 Carl Peeraer

8.15.6 Collapse Virtual Machine

The purpose of this package is to further compress application code. In fact, this package is supported at the source code generation level by the Mirror Classes Kit. But, by default, the generation of the P-code for the Collapse machine is disabled, and to use this extension, you need to download the Collapse package, read the instructions, and follow it.

Now, in essence. As you know, the machine code of the IBM PC is far from perfect (in terms of compactness). The Delphi compiler is also far from perfect. Even manually rewriting a large part of the KOL library code into assembler is not very helpful in miniaturization. So I decided to develop a minimal virtual machine that could do everything the processor does, but with more compact code. Collapse is the result of these efforts.

I will briefly talk about the fundamental structure of the Collapse virtual machine, or P-machine. As it turned out as a result of successive approximations, a minimal machine should not be able to do anything on its own, except how to transfer control to subroutines (either in the same bytecode or to machine procedures), and transfer control within its own code. For implementation, a two-stack architecture of a pseudo-machine was initially chosen, the high efficiency of which for the purpose of minimizing code has been known for a very long time (recall the Forth systems that appeared in the early 70s of the XX century). The computational stack in the Collapse machine is the same as the normal IBM PC machine stack, and a dynamically allocated block of memory is used for the return stack.

The result of the work is a really very small bytecode emulator, less than 0.5 Kbytes in size, and the ability to reduce machine code by 2 or more times. But the main drawback is that P-code has to be written by someone before it can be compiled into bytecode. So far, no one has undertaken to make a compiler from the Pascal language to the P-code, therefore, as a compromise solution, the generation of an alternative P-code was added to the MCK library, in parallel with the generation of the main Pascal code. Of course, the generation is only for the form design function, that is, for the code that generates the MCK. But on large forms with a very large number of visual and non-visual elements, the size of the machine code can exceed ten kilobytes.

One detail: in order for the Collapse machine to start functioning, the Baselib.pas module must be connected to the application, which contains a "library" of basic procedures that perform most of the operations. Initially, quite a lot of functions from this module are connected to the executable module, so there is no particular point in using the Collapse system for very small applications. With further growth of the application itself, the increase in growth due to the inclusion of functions from the Baselib module comes to naught, so when the program reaches 40KB and above, you can already try to use this package.

The Collapse system is tried and tested and fully functional. No significant slowdown in application performance was noted when using it. Unfortunately, it can only be applied to MCK forms containing only a basic set of components (all mirrored components on the form must generate a P-code for themselves, only in this case the P-code for the form can be generated). Maybe someday a universal compiler will be written from the Pascal language to the P-code of

© 2024 Carl Peeraer

the Collapse machine, and then the size of the entire application can be reduced by half more than the usual one.

8.15.7 FormCompact Property

The technology described in the previous chapter is very difficult to use, it requires many additional manipulations and fine adjustments. In short, it's best not to mess with her. Starting with version 3.00, an alternative mechanism for compressing the code for the initial construction of the form was proposed for MCK.

All you need to do is enable the **FormCompact** property for the TKOLForm component and recompile the project. The code that does the creation of the form is also roughly halved, but approximately 1K byte of additional code is added to provide property setting and parsing of the bytecode generated in this mode. In fact, it is also an interpreter, as in the case of collapse, but provided with a few more small functions.

As a result, if the construction of the form took less than 2 KB, there will be no noticeable reduction in the code (but it may increase). Savings will take place in the case of a very large number of components (especially visual) on the form, and / or a large number of forms. In general, FormCompact was developed for this case.

8.16 Additional Visual Objects

In this chapter, I will give a brief overview of additional visual elements based on the TControl object. Some of them were made using legacy techniques, by embedding them in the TObj object, but this does not mean that they cannot be used, they just turn out to be somewhat limited in this case.

It should also not be forgotten that the authors made some of their "components" at a time, in connection with their current needs, and since then they, most likely, have not been updated for a long time. It is possible that when you try to compile them, some small problems may arise that you will have to resolve, most likely on your own. Since all the source code is provided, I think this is not a big problem, especially since most of the extensions presented here are small in size.

As for the frequency of using the components below, I personally do not use all of them. Sometimes there is an urgent need for a component, and there is simply no time to make my own, and then I use something, for example, a list for choosing a font (despite using outdated technology, it is quite functional). The writing of this chapter, among other things, is also an attempt to revise the accumulated good, and add to the list of links on your website.

8.16.1 Progress bar

The main site presents the following two alternatives to standard progress:
KOLProgressBar (by A. Shakhaylo) - color progress control with a title (with an MCK mirror);

© 2024 Carl Peeraer

KOLRARProgress (by Dimaxx) - a progress line very similar to the one used in the RAR archiver.

8.16.2 Track bar (marked ruler)

There is no such visual element in the main KOL set. But there are two alternative implementations of such a control:
MHTrackBar (author D. Zharov aka Gandalf) and
KOLTrackBar(my own). It was this variant that I suggested as a demonstration of how to make extensions of the TControl object in the style of pseudo-inheritance. The implementation of this component uses a number of tricks that significantly reduce the size of the code added to the application. It may well be used as a visual aid for the construction of additional visual objects that implement new types of window elements.

8.16.3 Header (tables)

MHHeaderControl(by D. Zharov aka Gandlaf) - the table heading that is used in the list view, but without the list view itself. I've decided that I don't need such an object, and did not include it in the main KOL set. And so until now I have not yet come up with what I would need it for. Nevertheless, you never know who may have what needs.

8.16.4 Font selection

To select a font, it is quite possible to use API functions that allow you to enumerate the installed fonts in the system and add them to your combo box. This work is automated by a specially created TKOLFontCombobox object from the EnhCombos package (by Boguslav Brandys, Poland).

Also, there is a package MHFontDialog (author D. Zharov aka Gandalf), designed to call the system font selection dialog.

8.16.5 Color selection

In the main set of KOL objects there is only a dialog for choosing a color. But sometimes it is more convenient to choose a color from a combo box - when the range of colors is much more limited. For this it may be convenient to use the TKOLColorComboBox object from EnhCombos (by Boguslav Brandys, Poland).

8.16.6 Disk selection

There are at least two combo boxes available for selecting a drive name:
SPCDriveComboBox (author M. Besschetnov) and
BAPDriveBox (by A. Bartov).

© 2024 Carl Peeraer

8.16.7 Entering the path to a directory

For this task, you can try using the SPCDirectoryEditBox component (by M. Besschetnov).

8.16.8 Selecting a file name filter

The SPCFilterCombobox component by the same author implements a combo box for selecting a filter, similar to the one used in the file open dialog.

8.16.9 List of files and directories

There are several ready-made implementations of components for viewing lists of files and / or directories.
SPCDirectoryListBox, **SPCFileListBox** (by M. Besschetnov) - lists of directories and files based on a simple list box.
BAPFileBrowser (by A. Bartov) - almost ready-made file browser window, just paste it on the form and use it.
DirTreeView (I am the author, if I am not mistaken) - a tree for viewing (and selecting) directories on the disk.
KOLDirMon (mine again) - viewing the contents of a directory with tracking changes on the disk and automatically updating the contents.

8.16.10 IP Input

MHIPEdit(by D. Zharov aka Gandalf) - a special field for entering an Internet address (IP). Uses the appropriate API (it turns out that there is one).

8.16.11 Calendar and date and / or time selection

DateTimePicker (by Boguslav Brandys, Poland) - analogue of the corresponding component from VCL for date / time selection.
MHMonthCalendar (by D. Zharov aka Gandalf) - a calendar for placing on a form (this is exactly what is called from the DateTimePicker component by clicking on a special button, but in the form of a separate modal dialog).

KOLMonthCalendar (author E. Mikhailichenko aka ECM) - another implementation of the calendar.

8.16.12 Double List

Dialogue for working with a dual list: DualList. Author Boguslav Brandys (Poland) took the idea from the well-known RxLib package.

© 2024 Carl Peeraer

8.16.13 Two-position button (up-down)

KOLUpDown(the same author) - implements a button for scrolling values or positions up and down. I personally prefer to use two small picture buttons for this purpose, but each may have different preferences.

8.16.14 Button, non-rectangular

Tbitmapbutton from the archive KOL_HHC_Unit (by Tamerlan311) - provides drawing of free-form buttons.

8.16.15 Extended panel

KOLmdvPanel (by D. Matveev aka mdw) - a panel with additional features.

8.16.16 Label with image

A couple of examples of how you can put a picture on a form without writing your own OnPaint event handler:
KOLmdvGlyphLabel (by D. Matveev aka mdw) - a label with a picture instead of text.
SPLPicture (by A. Pravdin aka Speller) - a bitmap on the form.

8.16.17 Separator

KOLSeparator(by me) - something like the splitter component from the main set, but the principle of operation is somewhat different. The idea was carried over from the progenitor XCL library, which used just such a delimiter.

8.16.18 Table

KOLListEdit (by A. Shakhaylo) - allows you to edit all the cells of the table, which is displayed using the standard list view;
KOLListData (by A. Shakhaylo) - based on KOLListEdit, can be used as a database table view like TDBGrid in VCL;
StringGrid (author unknown) - TStringGrid component ported from VCL.

8.16.19 Syntax highlighting

Hilightmemo(by me) - an analogue of a multi-line editor with syntax highlighting, auto-completion, undo and undo rollbacks. Compact, there are over a dozen different compilation options to configure the desired set of features;
VMHSyntaxEdit (by D. Zharov aka Gandalf) - another editor with syntax highlighting, ported from the VCL component;
VMHPasHighlighter - based on VMHSyntaxEdit, specially designed for syntax highlighting of Pascal source text.

© 2024 Carl Peeraer

8.16.20 GRush Controls

The author of this package is Alexander Karpinsky aka homm. The package contains a set of controls with an extremely beautiful appearance (with "live" buttons in the style of "fused metal"), these visual elements work very quickly, up to the use of MMX for graphics acceleration. Of course, using this set of components slightly increases the size of the application (from about 20K extra). But sometimes this increase in size can be justified by a significant improvement in appearance, making your development more competitive in the eyes of users spoiled by beautiful Windows interfaces. Also important is the fact that the look of the interface will be the same "cool" regardless of whether XP themes are included or not. It doesn't even depend on the version of the operating system, and it looks just as attractive in Windows 95.

Seduced by the appearance of these controls, I also decided to have a hand in the implementation of this package. I have developed a special migration module ToGrush.pas, which allows you to significantly simplify the transition to using this interface from the usual boring appearance. Now it is enough to add a link to this module to the end of the list of used modules, and after recompilation, almost all controls take on a new look. Moreover, if you enclose this addition in brackets `{$ IFDEF USE_GRUSH}, ToGrush {$ ENDIF}`, now the presence of the conditional **compilation symbol USE_GRUSH** will determine whether this package is used, or the controls retain their standard interior. This allows, in particular, to quickly compare the size of the application in the case of using and not using a package, or quickly rebuild the application for the case of the standard interface, if the application suddenly began to behave incorrectly, and you suspect that this is the package (although this, of course, is hardly the case). And, of course, this allows you to very quickly switch to a new interface without completely redesigning it for new controls.

At the same time, there are a number of peculiarities associated with the absence of some standard controls in the GRushControls package. For example, it does not have a toolbar. This visual element, however, can be easily imitated by creating a panel with buttons (fortunately, the buttons in the package under consideration can contain both a picture and a text).

In the **ToGrush module**, the toolbar construction function actually takes over the creation of an analogue of the toolbar control according to this scheme: a panel is created and buttons are placed on it. An important point is that you can no longer use system images for the control bar icons, and images whose width differs from the height, as well as images from the image list. In this case, for example, MCK generates such a code to create a ruler, in which the image descriptor is passed to the NewToolbar function as a parameter. / In any of the other cases, in the first call to NewToolbar, 0 is passed in place of this parameter, and images are added later, and as a result, the NewToolbar function from the ToGrush module cannot associate icons with the / buttons.

I even went to make changes to the KOL modules, KOLadd and the MCK package so that the transition to GRushControls can happen without any problems in the case of using MCK. Namely, the TEdgeStyle, which determines the appearance of the panel border, has been enriched with the esTransparent and esSolid values. Typically, these styles are no different from esNone (and correspond to the absence of a visible indented or extruded border). In the case of using

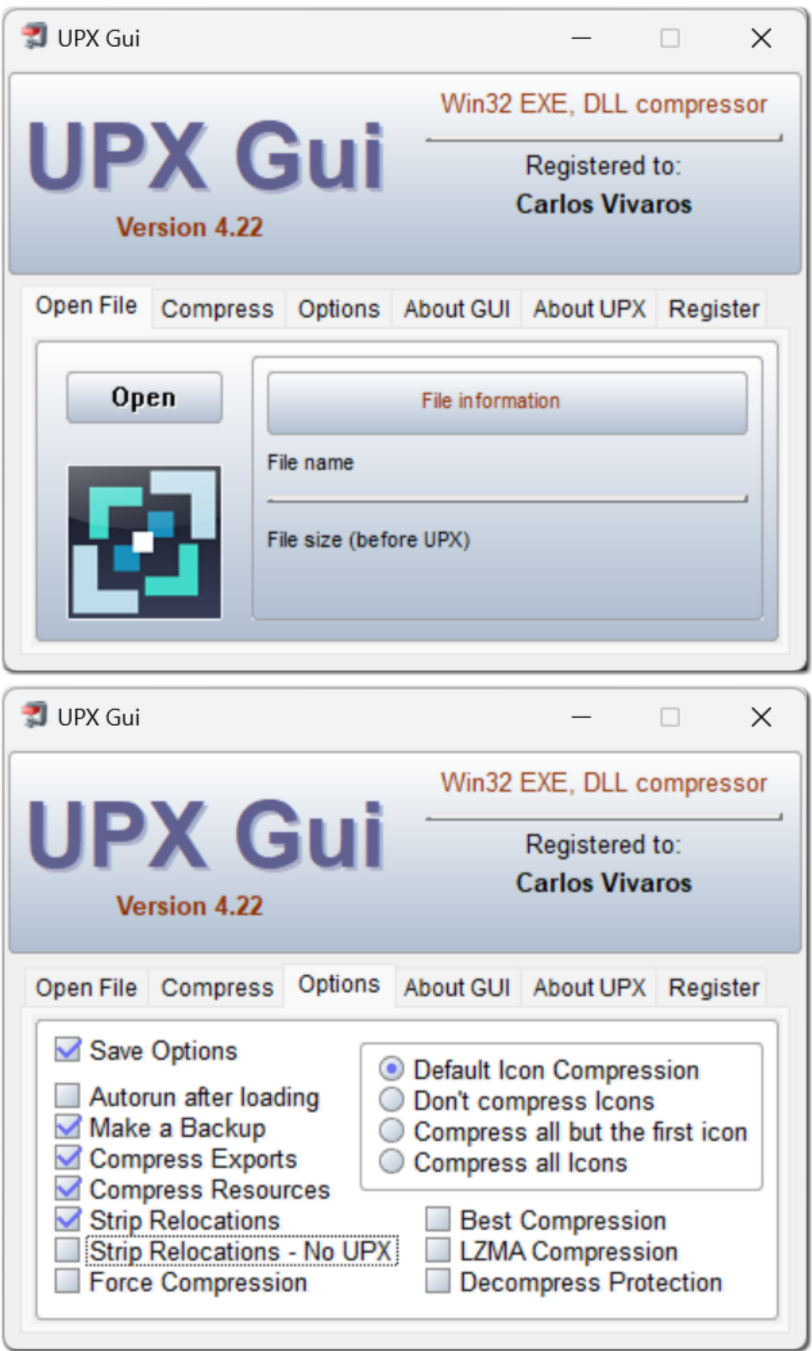
© 2024 Carl Peeraer

GRushControls through the ToGrush transitional module, this style is used to create the previous (not in the GRushControls style) panel (in the case of esTransparent, such a panel is immediately made transparent) and again, without selected borders. The fact is that nested panels are often used to group controls, but their borders or fill should not make them visible against the background of parent panels or shapes.

In addition, I moved the **ShowQuestionEx** function and the **ShowQuestion** and **ShowMsgModal** functions that use it from KOL.pas to the KOLadd.pas module, and added the above reference to the ToGRush module in conditional compilation brackets to the USES list of the KOLadd module. Thus, dialogs created by these functions will also automatically use the GRushControls package if the USE_GRUSH symbol is defined in the application. A similar link to the ToGrush module has also been added in the **KOLDirDialogEx** module, so this dialog for quick directory selection now uses the GRushControls style, if it is defined for the entire application.

© 2024 Carl Peeraer

Example of an application with **Grush Controls**:



8.16.21 Other additional visual elements

Unfortunately, not all extensions created for KOL can be downloaded from the main Web site, and many of the archives listed here are also represented by links to the original author's archives. A large number of additional links can be found on friendly sites. For example, the implementation of elements such as cool bar, all kinds of Gauge and indicators, and many others.

© 2024 Carl Peeraer

8.16.22 Tooltips

As you know, tooltips, if configured by the developer, are shown when you hover over some visual elements of the form. In the VCL, in order for such hints to appear, it was enough to enable the ShowHint property, and assign some text to the Hint property of the control.

In KOL, properties such as Hint and **ShowHint** were not originally provided. We got by (at least personally, I got by) with the so-called tooltips *, which are provided by the system for the toolbar. On this ruler, tooltips can really be useful, at least in the display mode without text labels on the buttons. Simply because it is often not always possible to find out from the pictogram what is done when the corresponding ruler button is pressed, especially when the user is just getting to know the application or rarely works with it, and is not obliged to remember the purpose of each button from time to time.

As for the ability to organize floating tips for any controls on the form, I personally take this idea more negatively than positively. A well-designed interface does not require such a means for constant reminders, especially since tooltips often obscure controls and interfere with work, even if they remain transparent to mouse clicks. If the user is constantly using the application, then such pop-ups can even be annoying (and therefore it would be nice to be able to turn off such prompts if the user so wishes).

By the way, tooltips are historically preceded by the idea of displaying explanatory information in a special field, for example, in the status bar. At the same time, at least such prompts are not intrusive, and do not obscure the "working surface" of the form.

Nevertheless, tips are still needed, and this is evidenced by the fact that the site contains several packages of various authors that implement a similar opportunity. Ultimately, I agreed with the requirement to include official support for pop-up labels in KOL, but as conditionally compiled code. In order for the hints to become available for the KOL application, it is now enough to include the conditional compilation symbol **USE_MHTOOLTIP**, and the Hint field, in particular, automatically becomes available. Of course, to use the hints mechanism from D. Zharov (aka Gandalf) in the project, you will need to download and unpack a small archive with additional included code (**KOLMHTooltips.pas** module).

The **KOLMHTooltips** module, is located in the ***https://www.artwerp.be/kol/kol-mck-master_3.23.zip*** archive.

8.17 XP Themes

Beauty is a terrible force.
Consider especially the meaning of the word scary.

To connect to an XP themes application, you just need to add a special resource containing a manifest to it, or put a manifest file named **<your_application_name> .exe.manifest**. As a result, however, the application does not always work correctly and displays as it should. It is for

© 2024 Carl Peeraer

this reason that I always prefer the second use case for the manifest - with an external file. It can always be deleted, including if the user has problems due to the connection of the manifest, you can simply advise in response to his complaint to delete this file - which is much faster than passing him a version of the distribution kit in which the manifest resource is not included in executable module.

It's no secret that by adding an external manifest file, you can try to "improve" the appearance of any 32-bit application, even made and released before the advent of the XP operating system. I wonder how many of you have tried to improve the Delphi shell interface this way. For example, I managed to achieve a positive effect for Delphi 6 by putting the manifest file Delphi32.exe.manifest in its \ bin directory. Version 5, however, could not start after this improvement. But this trick passed with a bang for Delphi 2. At least, the bookmarks of the pages of the edited modules have become more distinguishable, and now you can clearly see which bookmark is active.

In fact, things are not so simple, and in order for the themes to start working correctly, something must be added in the code. At a minimum, you need to call the **InitCommonControls API** function (which is executed by itself if you already use "general" controls like list view or tree view in the application, but which must be called additionally if there are no such visual objects in the application).

To correctly connect themes to the XP MCK application, it is recommended to use the **TKOLMHXP** component, the author of which is Dmitry Zharov aka Gandalf. This component will provide both the connection of the manifest (moreover, it is possible to choose the connection method: as a resource or as an external file), and call the required API functions. In addition, with its help, you can correctly fill in the fields of the manifest (which is actually an XML document). Although the values of these fields have no effect on the functioning of the manifest.

The appearance of some controls may differ from what was expected to be seen when connecting the manifest. For example, **RichEdit does not want to render using XP themes** until you add the conditional compilation symbol **GRAPHCTL_XPSTYLES** (remember to rebuild - Build - the project). This is done because additional code is required to work correctly with the manifest, and this code is not needed at all if you do not intend to use the manifest (or the appearance of the RichEdit window with a regular frame does not upset you, and saving a few dozen bytes of code is more important).

Similar problems are found for tabbed pages - **TabControl**. When creating it, I decided to use a regular panel as a background for each page, and it is in no way affected by the manifest. As a result, the inside of the page looks a little faded compared to the colorful off-page views. To fix this problem, add the conditional compilation symbol NEW_ALIGN (after sufficient testing, and possibly very soon, this version of the code will be standard, while the previous version of the alignment will, on the contrary, become optional).

It should also be noted that when connecting the manifest, there may be problems with the transparency of many types of controls (for which the Transparent = true property). Some of them turn black, some are not displayed correctly. To solve problems with the toolbar, they had

© 2024 Carl Peeraer

to go to unprecedented measures altogether: his MCK mirror had to add the **FixFlatXP** property, set by default, which controls code generation in such a way as to avoid the appearance of a black bar instead of the toolbar. That not only slightly increases the code, but also prohibits the use of certain combinations of options.

8.18 Extensions of MCK itself

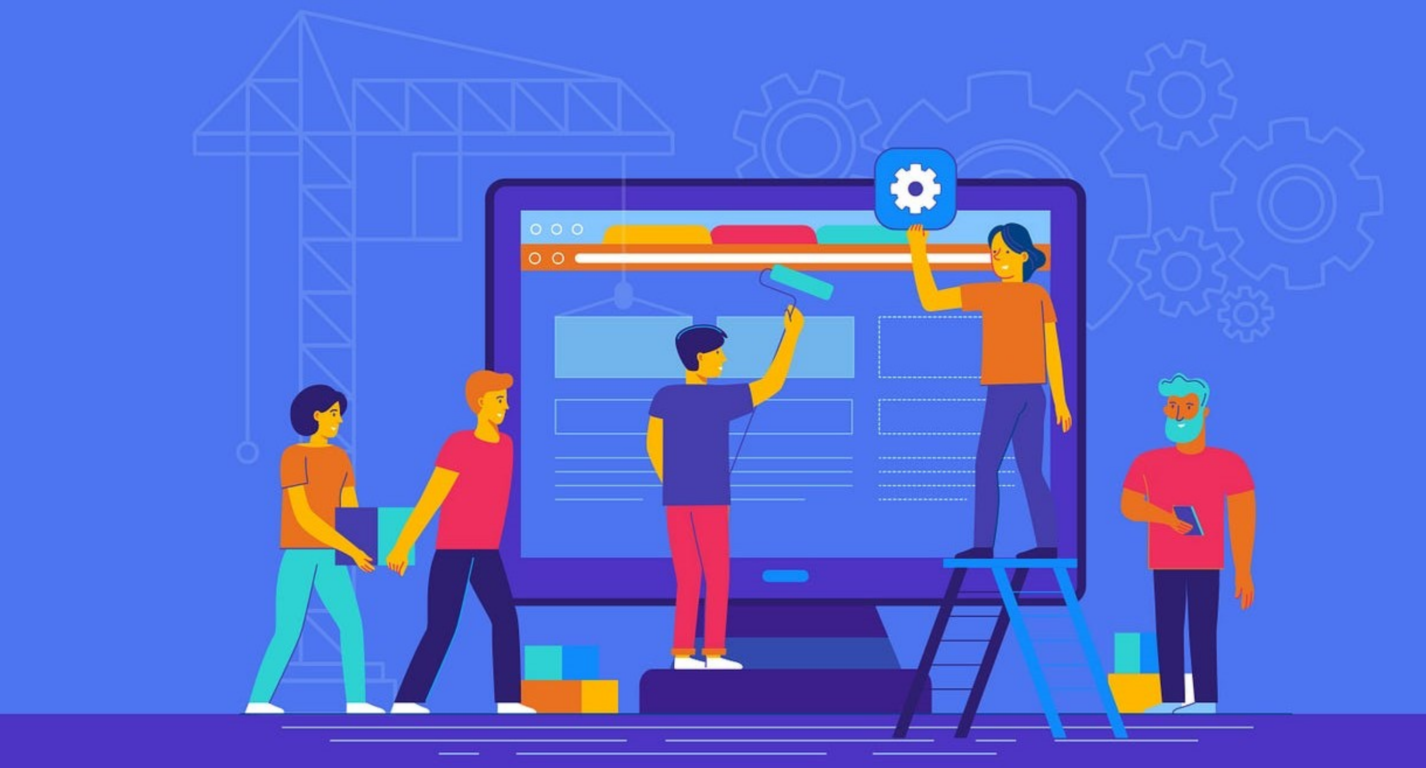
8.18.1 Improved font customization

The **KOLFontEditor** package builds in the Delphi environment a font editor of the TKOLFont type, which allows you to customize the font visually in the dialog (at the development stage). Without this magic extension, the name of the font will have to be typed on the keyboard. The author of the package is Alexander aka Speller (Russia, Primorye).

8.18.2 Alternative component icons

The **KOLmirrosGem** package contains an alternative set of icons for MCK mirror components. Author Roman Vorobets.

© 2024 Carl Peeraer



Working with Extensions

To install extensions that have mirrored MCK classes, you can install packages containing mirrors of these (visual or non-visual) objects.

9 Working with Extensions

- Enter topic text here. 482
- Using Extensions 482
- Developing your own Extensions 483
  - Development of non-visual extensions 483
  - Development of visual extensions (controls) 484

9.1 Installing Extensions

To install extensions that have mirrored MCK classes, you can install packages containing mirrors of these (visual or non-visual) objects. If such packages are not contained in the distribution (supplied) archive, or there is no package version suitable for your version of Delphi, then you can create them yourself by adding modules containing the Register function to them. It is the Register function that provides registration of the components for placing them on the Delphi component line.

But it is very important not to forget that with each update of KOL and MCK, all packages that depend on the MCK must be fully compiled (Build). If you have a large number of such additionally installed packages, it is not surprising to forget to rebuild any of the packages. In this case, I can offer a simple solution: do not install the MirrorKOLPackageXX.dpk package, and a bunch of additional packages, but create your own MCK package, where you include all the modules from the main package, and from all the packages that you need to install. In this case, updating the version will be greatly simplified, since it will be enough to rebuild only this one package.

In addition, extensions, even with design-time mirrors, are quite possible to use without setting them to the component palette. Of course, you have to write the code for constructing such objects yourself. Or, having installed such extensions temporarily, copy (and correct if necessary) the code generated by MCK from the inc file into the OnFormCreate event, and then remove the extension from the ruler.

9.2 Using Extensions

In order to use the extension, you first need to refer to it. Namely, to register a link to the extension in the compiler's uses directive, while specifying the correct path to the source (or compiled) extension files in the project options. And then create extension objects.

Usually, for extended objects, as well as for basic KOL objects, the constructor function NewXXXXX is defined, which returns a pointer to the created object. But sometimes there is no such function, and then the call to the constructor should look like the old Turbo Pascal: `new (name_of_object_variable, Create);` Here the new function is built in and the Create constructor is provided by all simple Object Pascal objects. Instead of an identifier `name_of_object_variable` you must provide your own name for the constructed variable.

 © 2024 Carl Peeraer

If you have an MCK mirror, if you've installed it, it's even easier to use: just drop the mirror component on the form and configure its options. In this case, from the above list of priorities, there is usually only the indication of the path to the source files in the project property (if these paths are not already known to the compiler).

Many components can themselves correctly ensure the presence and correct placement of a reference to the required unit in the uses directive. But sometimes the machine malfunctions, which can lead to misunderstandings. Namely, in the directive uses the reference to the modules of the extension itself and its mirrors fall into the wrong positions when the addition is performed by the Delphi environment itself at the moment of throwing the component onto the form.
In this case, the project refuses to compile, begins to require files that are completely unnecessary for it (the same designintf.pas) or resent the mismatch of the version of the system files. All you need to do is tweak the uses cluster by sending references to VCL units (and mirrors are exactly VCL units) inside protective brackets `{$ IFNDEF KOL_MCK}`... `{$ ENDIF}`...

9.3 Developing your own Extensions

If you like to ride - love to carry sledges.
(Russian folk proverb)

This topic was brought up by KOL programmers almost immediately after the project was born. There are already a number of articles and so-called tutorials * devoted to this topic. In this book, I will venture to dwell on this issue once again, at least briefly.

9.3.1 Development of non-visual extensions

First of all, I will immediately note that the development of a non-visual extension is not a problem at all. It is enough to inherit your object from TObj, and then do whatever you want with it. Or rather what you need. Likewise, there are no major problems creating an MCK mirror for an extension built in this way. For this, another module is formed, in which the mirror class itself, inherited from TKOLObject, and the Register procedure are located. That's it, you can create a package and install it (do not forget to draw an icon and place it in a dcr file, as you usually do when developing Delphi VCL components). The problem is easily approximated for the case when several objects are defined in one module, for each of which it is necessary to have its own mirror class.

For example, you can look at the source codes of any available non-visual extension, of which there are countless numbers.

© 2024 Carl Peeraer

9.3.2 Development of visual extensions (controls)

A much more serious topic is the development of your own visual object. It should be noted right away that it is not customary in KOL to create such visual extensions simply because you wanted to slightly tweak the setting of the initial properties of any existing visual component.

I remember the early days of Delphi's triumphant march. Every aspiring programmer was happy to take advantage of the incredible simplicity of the new mechanism for creating their own components: hundreds of "round button" or "light bulb" components appeared. More often than not, there were much fewer really useful components that really extend the capabilities of the standard library.

Initially, many developers of visual components took the path of "embedding" the TControl visual object inside its successor from TObj. But this method is fraught with the fact that the "component" formed in this way will not be able to provide all the necessary levers to control the object, hiding its renderer inside. Or you have to write an incredible amount of code that will provide access to all the necessary properties of the new visual object. Or, to expose the TControl object to the outside through a property or a field open to all the winds, which is also not very nice (And the call to any property will now look like `MyObj.Control.Width`, eg).

Another problem that arises when using the method of "injecting" a TControl object inside its inheritor from TObj is the problem of correctly deleting used objects. You can add your object to the parent using the Add2AutoFree method, but then you cannot directly destroy it using the Free method or the Free_And_Nil procedure: when the parent is destroyed, the destructor will be executed again, and the program will most likely break at the exit. It is more correct in this case to add the container object to the list of objects of automatic destruction of the most controlled visual control included in it. And then it will be possible to destroy it by calling either a special method of the enclosing object, which will call the Free method for the control, or provide direct access to this control in its descendant TObj,

Unfortunately, there were even "tutorials" and the first visual extensions using this technique of embedding inside TObj before I made the necessary efforts and sent component creativity on the right track.

The correct mechanism for creating your own visual extension is inheritance after all. If a fundamentally new visual element is created that is absent in the library, then it must be inherited from the TControl object, and if an existing extension is used as the base one, then it must be inherited from it. Although, in life, examples of inheritance from extensions have not yet been noticed (which, by the way, is not bad at all for the purpose of saving application size: the lower the hierarchy level, the less memory is allocated for virtual method tables).

But there is one "BUT" here. Unlike VCL, KOL does not use virtual constructors. It is customary here to define NewXXXX functions (parameters) for constructing objects. It would seem, so what: we create a new type of object, write a function for it `NewMyControl (…): PMyControl` and ...

© 2024 Carl Peeraer

And here the problem begins: what to write in the code of this function, if the creation of the TControl ancestor object should be at least the _NewControl function, and it can only create an object of the TControl type, but not its successor.

In fact, the problem arises only if new fields are added to the inherited object (not to mention new virtual methods, practice has shown that you can do without them just fine). If only non-virtual methods are added, then the problem is solved very simply: it is enough to call _NewControl inside your constructor, and return exactly this result at the output, casting it to the required type (PMyControl). Unfortunately, creating a truly new object is almost never complete without adding new fields, so the problem persists.

As a result, the following convention was adopted: for an object inherited from TControl, new fields are added through an additional structure or object, for which the CustomData and CustomObj properties are used.

You can use either the CustomData property (if a simple structure is enough to store new fields), or CustomObj (if you want to get some kind of benefit from the object as a keeper of new fields), or both (but usually one of them is enough). When an object of type TControl is destroyed, its CustomData and CustomObj fields are destroyed automatically, and the FreeMem function is used to delete the structure pointed to by CustomData, and the Free method is used to destroy the CustomObj object.
// Note: The benefit of using the CustomObj object may be, at least, the ability to define its own destructor for it, in which any other resources allocated during operation will be freed //.

As a result, the number of fields of the TControl object in the descendant does not change. This makes it possible to do the same as in the previous case: in your "constructor" NewMyControl, call the _NewControl function to construct a visual object, and at the output, cast the resulting object to the type `PMyControl`...

An additional plus of this approach is that your extension continues to use the same single table of virtual methods of its successor TControl (we have already agreed that you will not add new virtual methods, otherwise the above trick is impossible). This means that the increase in the code will occur only by the size of the code of the new methods used in the application.

And, as I said before, there is a certain amount of reading material specifically for those who wish to create their own extensions, visual and non-visual. You can find them on the KOL and MCK sites, as well as numerous examples of such extensions. So I will take the liberty of reducing the length of this document and avoiding unnecessary duplication of information.

© 2024 Carl Peeraer

Appendix

The idea of the need for this chapter came to me when, once again, looking at the forum, I found a question that had already been asked many times, was answered many times, entered into the FAQ and FAQ, but continues to be asked again and again, and sometimes even by people, who started writing on KOL not yesterday.

10 Appendix

- Errors of programmers starting to learn KOL 488
- Developer Tools 491
- Demonstration Examples 491
- KOL with Classes instead of Objects 494

10.1 Errors of programmers starting to learn KOL

I didn't even notice the elephant ...
(Krylov's Fable)

The idea of the need for this chapter came to me when, once again, looking at the forum, I found a question that had already been asked many times, was answered many times, entered into the FAQ and FAQ, but continues to be asked again and again, and sometimes even by people, who started writing on KOL not yesterday. I hope that having another source with answers to such questions will help reduce the number of such errors, leaving room on the forum for more meaningful discussions. Some of the problems presented here have already been described in the relevant sections of this book. However, it will probably not be superfluous to bring them all together and discuss in even more detail. I'll try to sort the errors by their repetition rate: the first will be the ones with the highest rating in the list of questions from the developers.

1. Assigning an event handler using the MakeMethod function and typecasting to TOnSomeEvent. ("Why isn't my handler responding to the event?")

Usually a similar problem occurs among people who are even quite familiar with the Pascal language, but do not think especially about what machine code is formed as a result of the compiler's work. / Don't worry, the vast majority of programmers don't think about it. And they do the right thing: that is why compilers exist to think not about what machine code will be obtained from the source code in a High Level Language, but to think only about the problem to be solved. /

It looks like this. You are not trying to assign an object method to an event handler, but a regular procedure. For example, like this:

```
procedure MyOnClick (Sender: PObj);
begin
 ... some code ...
end;
...
MyControl.OnEvent: = TOnEvent (MakeMethod (nil, @ MyOnClick));
```

At first glance, everything is correct here, and the description of the MyOnClick function is quite consistent with the description of the handler type.

```
TOnMessage = procedure (Sender: PObj) of object;
```


© 2024 Carl Peeraer

But in fact, this description is not entirely correct, namely, the matter is in the boldface phrase of object, which indicates to the compiler that a handler of this type should not be a simple procedure, but a method of an object. The difference between a method and a simple procedure is, in fact, in the presence of a hidden additional (first in order) parameter - a pointer to the object for which the method is called. In order to formally reconcile this discrepancy in the number and order of parameters, it is enough to add one more parameter, just the first in order, to a simple procedure. Its name, of course, does not matter; the type can be a pointer or any other type of the same size. For example, the following fix to the handler header will put everything in place:
procedure MyOnClick (DummySelf, Sender: PControl);

Why didn't the previous version work? The explanation is quite simple: instead of the formal Sender parameter, an object pointer nil fell (exactly in accordance with how your method was "created" by the MakeMethod function), and the Sender parameter itself did not reach the handler at all, being left out.

And if we remember about the implementation of the parameter passing mechanism in Pascal procedures, it becomes clear why the incorrect version of the header did not necessarily lead to much more fatal consequences. The first three parameters, which fit into double words (32 bits on the PC platform, or 4 bytes), are transferred through the processor registers EAX, EDX and ECX. The absence of one parameter in the description of the handler led to the confusion in the procedure of the order of the parameters passed through the registers, and nothing more. Note that in the case of an agreement on passing parameters through the machine stack, as it happens in C, or when using the stdcall directive, the very first attempt to access an incorrectly declared function would most likely lead to a stack level violation and an immediate application crash. Perhaps, in this case, there would be more sense in such behavior: at least, it would become clear immediately that something is really wrong on this site - compared to the tacit disregard of the processor of his duties, in our case. However, if an incorrectly passed parameter is used in the handler, then access to the fields of the object, which was replaced by the nil pointer, will also be immediately detected when trying to execute the handler.

2. "Can't install MCK", "compile MCK application", asks for some designintf file, proxies ", and the like

Some people who ask this question sometimes get straight to the point and ask me to send them this ill-fated file. In fact, such a file is not needed at all, and the only problem is not having read the instructions for creating an MCK project carefully enough. All you need to do is close the VCL project prototype and open the "true" MCK project - with the name specified in the projectDest property of the TKOLProject component.

Sometimes the same trouble happens with the previously created MCK project. The reason for the breakdown is usually the loss of the conditional compilation symbol KOL_MCK. It could get lost, for example, as a result of deleting an unnecessary, at first glance, file with the drc extension in the project directory (or you could forget to take it with you, transferring the project to

another machine or to another folder). The cure is very simple: open a dialog with project properties and add the KOL_MCK symbol manually to the list of conditional compilation symbols (Conditional Defines, on the Directories / Conditionals tab - for all Delphi versions, at least from Delphi2 to Delphi7, this tab takes place).

The reason designintf is starting to be required is, as it should be clear by now, in the presence of a reference to it in the mirror.pas module, which is the main one in the MCK package. All MCK modules have a link to the mirror.pas module, closed from the prying eyes of the compiler by the conditional construct {$ IFNDEF KOL_MCK}... {$ ENDIF}. This is, so to speak, part of a trick that allows MCK to exist and trick the Delphi environment at design time by passing off a KOL project as a respectable VCL project.

A completely similar phenomenon can occur with an application in which the KOL_MCK symbol is present, and has not disappeared anywhere. For example, if, when adding any MCK component to a form, a reference to the MCK module (for example, to mckCtrls.pas) will be inserted outside the above brackets {$ IFNDEF KOL_MCK}... {$ ENDIF}. In this case, the compiler's message may be somewhat different: for example, that a certain module was compiled with a different version of the VCL, and as a result, compilation cannot be continued. The solution is the same: find the module in the USES section, the link to which is in the wrong position, and move it inside the brackets.

Interestingly, the experience of the first struggle with such mistakes does not set you on the right path once and for all. There is a very high chance that after successfully working on one or even several projects developed using MCK, you will again come across this message, and you will not be able to immediately remember what the solution to the problem is.

3. A KOL project containing two or more forms is not working properly

More often than not, the problem is in the use (or rather, non-use) of the Applet object. If the project uses MCK, this means that the TKOLApplet component must be dropped onto the main form. In case of programming without using MCK, you need to execute the following code:
```
Applet: = NewApplet ('Title');
```
And only then create all forms as children of the applet:
```
Form1: = NewForm (Applet, 'Form1');
```
And most importantly, the Run procedure must be called with this special purpose object as a parameter:
```
Run (Applet);
```

In the section on this object, I have already explained that the Applet is needed, among other things, to provide control over the flow of messages between multiple forms. With a separate Applet object, all forms in the project become "children" of the Applet object, which in this case acts as the application button on the taskbar.

In KOL / MCK projects, the Applet global variable is similar to the Application global variable of type TApplication in VCL projects. But, unlike the VCL, the use of the Applet is optional. In the

© 2024 Carl Peeraer

case of a simple application from one form, you can often do without using this object. At least, if you do not use other special features of this object, namely: hiding the application button on the taskbar, working with the icon in the system tray, and others. This will save another half a kilobyte of code, which can be noticeable for a small application.

10.2 Developer Tools

Of course, the main developer's tool is the programming environment. I prefer to use Delphi, and among all the other versions, Delphi 5 and Delphi 6 are the optimal ones. Someone prefers licensed free products such as Free Pascal and Lazarus. But this chapter will focus on additional tools that can be useful for all sorts of utilitarian work, even those that, it would seem, have nothing to do with programming itself. For example, to publish your work on the World Wide Web.

Part of the described toolkit exists in nature by itself, but I will allow myself to mention such means, because, to some extent, this is also an experience, and it can and should be transmitted and disseminated. Other tools have been developed using the KOL library by various authors and are often indispensable as well. Most of these tools can be found in the Tools section of the main KOL WEB site.

- **DClear** - utility for cleaning Delphi project folders.
- **DfmUn2An**- converts dfm file resource (s) from Unicode to Ansi, allowing you to port applications developed in new versions of Delphi (6, 7) to an older format (4, 5). Author: Bartov Adeksandr.
- **DiffLines** - a program for line-by-line comparison of very large files (up to 4 gigabytes).
- **MCKAppExpert** - creates an MCK application template. Author: Thaddy de Koning.
- **MCKAppExpert200** - similar to the previous one, but compatible with Delphi2009-2010.
- **xHelpGen** - help generator based on source code.

10.3 Demonstration Examples

Demos are one of the primary sources for exploring a library, component suite, and other development toolkit. First of all, because, unlike conventional documentation, they not only describe what can be done, but also show how it is done correctly. For KOL, there is also a set of such examples, made, most often, hot on the heels of discussing a problem, or in the process of explaining how to use this or that library feature for any urgent tasks.

All of these examples (and a few others) can be downloaded from the main KOL site: http://f0460945.xsph.ru/ - in the Downloads section:

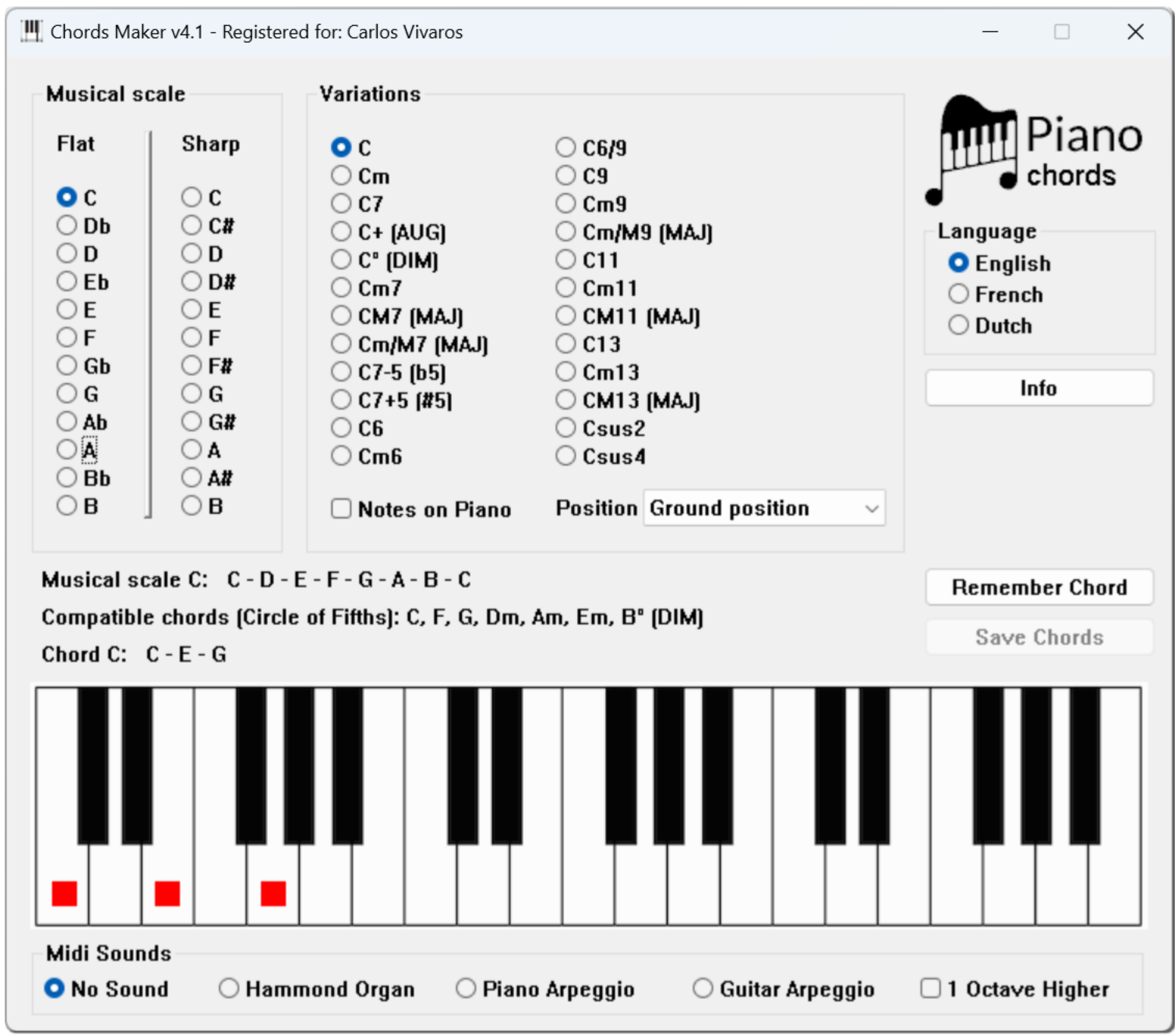
DemoEmpty - empty application;
DemoKOLonly - a few simple projects on KOL without using MCK;
adv - demonstration of a rotating font;
Demo2Forms - two forms per project;

© 2024 Carl Peeraer

DemoModalForm - calling a subform modally;
DemoModalHide - calling a modal form with its hiding at the end of the dialogue;
DemoModalVCL2KOL - Calling a KOL form from a VCL application - modal calling a KOL form from a VCL application (the KOL form is located in the DLL);
DemoVCL2KOLdll - Calling the KOL form from a VCL application is modeless;
Demo2NonModalForms - two modeless forms are called from the main form - two forms are called modeless (but once);
DemoFrames - use of frames;
DemoSplash - splash form;
DemoStayOnTop - on top of all windows;
DemoMDI - simple MDI application;
DemoDynamicMenus - dynamic menus;
DemoBitmap2PaintBox - drawing a bitmap in the drawer;
DemoKOLBitmap - loading and drawing a bmp file;
DemoOGL1 - demonstration of calling Open GL functions from a KOL application;
ActionsDemo - demonstration of TActions;
DemoCABExtract - an example of unpacking a CAB file;
DemoClientServer - network connection (sockets);
DemoListViewCheckboxes - general list with selection flags;
DemoTreeViewDrag - dragging tree nodes;
DemoProgressBar - progress;
DemoRichEdit - formatted text;
DemoWordWrapBitBtn - BitBtn button with text wrapping;
DemoMyException - exceptions in KOL;
DemoShellBrowser - list of files;
DemoNoFlicks
DemoThread - multithreading;
DemoTrayIcon - tray icon;
DemoTrayOnly - only the tray icon, the form is not visible;
tictactoe - tic-tac-toe.

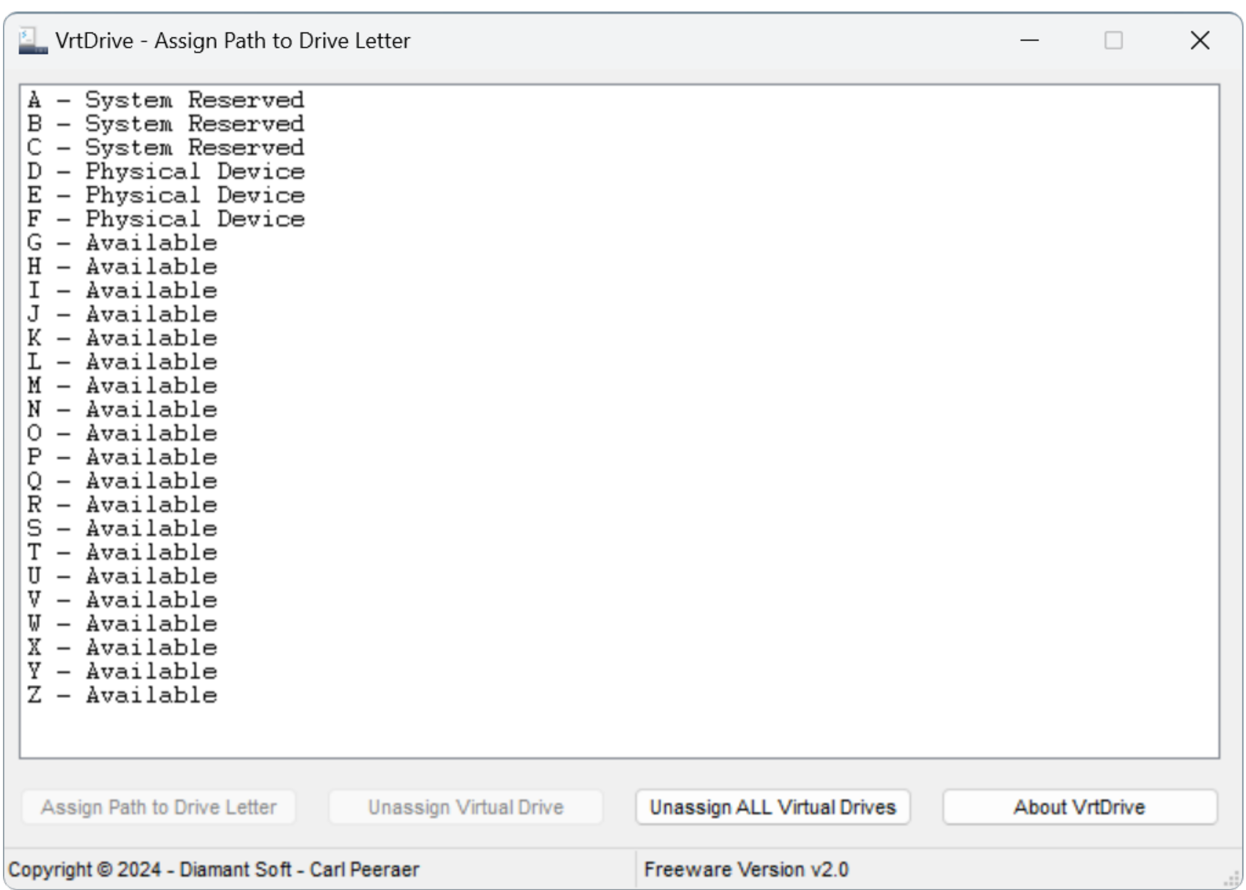
© 2024 Carl Peeraer

Piano Chords Maker - Program by Carl Peeraer, the translator of this manual. Info and Download: https://www.artwerp.be/akkoorden/



VrtDrive - Assign Path to Drive Letter: Program by Carl Peeraer, the translator of this manual. Info and Download: https://www.artwerp.be/vrtdrive/

© 2024 Carl Peeraer



10.4 KOL with Classes instead of Objects

Attention: everything that is written in this application, starting from version 3.00, is no longer relevant at all - classes in KOL are no longer needed and are not supported. The main thing is that this mechanism existed, is available in the archives of the source codes of previous versions, and you can always resume it in case of urgent need.

In order for KOL projects to be successfully compiled by the Free Pascal compiler version 1.xx.xx, that is, even when this wonderful compiler did not support simple Object Pascal objects, a version of the KOL library code was specially "created" for this purpose.

I put "created" in quotation marks because, in fact, the second version of the same KOL.pas module was not created, it would be simply unimaginably difficult to constantly synchronize all changes in two such large modules (I do not always succeed in one all agree). At the same time, the use of constructions {$ IFDEF...}... {$ ELSE}... {$ ENDIF} to implement such a variation of the code, when in the case of adding any symbol of conditional compilation, it was also rejected. First, it would clutter up the source code, which is already difficult to understand. And secondly, it could have confused the Delphi IDE and could well have confused its Code Completion and Code Navigation subsystems.

© 2024 Carl Peeraer

As a result, a combined approach was chosen using a specially designed external preprocessor GLUECUT (this name has nothing to do with glitch, and is a combination of two English words "glue" and "cut"). This application can be used not only for the purpose of converting the KOL module (and related) into classes, but also for any other tasks of a similar nature that require a powerful text processor.

Why is the approach combined? The answer is that in the process of processing the text of the module, a set of rules is used to control the transformation of the input text into the output, which can change in different parts. Sections are set by "tags" in the source file, which are comments of the form // [...], located from the beginning of the line and occupying the entire line. The rules allow you to set the replacement of specified substrings, as well as manipulate whole strings. That is, the content of the input text is partially used, and some of the information about what and how to replace is in an external command file.

For example, if you have to write @ Self in KOL, because Self is the object itself, and you need a pointer to this object, then in the case of using classes, the @ sign should be eliminated, since the class instance variable itself is already a pointer. View rule

```
REPLACE [@Self] [Self]
```

- will perform all such replacements.

In addition, the rules allow you to bypass or replace specially marked sections of code, and in this case, short comments in curly braces are used as labels, for example, all sections {-} ... {+} when copying the source text into the output can be skipped by the rule

```
SKIP [{-}] [{+}]
```

In a sense, an analogue of conditional compilation is the opposite construction, which allows you to insert text for the future input variant for classes into the source code, but so that in the source code itself, this text will be a pure comment from the point of view of the Pascal compiler:

```
{++} (* TObj = class; *) {-}
```

Such a replacement is implemented in the command preprocessor language with a couple of rules

```
REPLACE [{++} (*] []
REPLACE [*) {-}] []
```

As you can see, everything is simple. A developer using an old version of the Free Pascal compiler, or simply preferring classes, having received (downloaded, updated) a new version of the KOL.pas file, simply runs a bat file that calls the GLUECUT utility with the required parameters, and as a result, a KOL version with classes.

Finally, I will add that, starting from version 2.10, the Free Pascal compiler fully supports simple Object Pascal objects, and there is no longer a special need for converting KOL to classes. Nevertheless, using classes for some purpose may still be useful to someone, so I felt it necessary to provide information about this possibility.

© 2024 Carl Peeraer

Made on rainy days in 2021, 2022, 2023 and 2024...

KOL / MCK User Guide

Vladimir Kladov
Creator of KOL / MCK and ALL the documentation

© 2024 Carl Peeraer