Objective Grid User’s Guide

This Manual

© Copyright 1997-2014 Rogue Wave Software, Inc. All Rights Reserved.

Rogue Wave and Stingray are registered trademarks of Rogue Wave Software, Inc. in the United States and other countries. All other trademarks are the property of their respective owners.

Acknowledgments

This documentation, and the information contained herein (the “Documentation”), contains proprietary information of Rogue Wave Software, Inc. Any reproduction, disclosure, modification, creation of derivative works from, license, sale, or other transfer of the Documentation without the express written consent of Rogue Wave Software, Inc., is strictly prohibited. The Documentation may contain technical inaccuracies or typographical errors. Use of the Documentation and implementation of any of its processes or techniques are the sole responsibility of the client, and Rogue Wave Software, Inc., assumes no responsibility and will not be liable for any errors, omissions, damage, or loss that might result from any use or misuse of the Documentation.

ROGUE WAVE SOFTWARE, INC., MAKES NO REPRESENTATION ABOUT THE SUITABILITY OF THE DOCUMENTATION. THE DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. ROGUE WAVE SOFTWARE, INC., HEREBY DISCLAIMS ALL WARRANTIES AND CONDITIONS WITH REGARD TO THE DOCUMENTATION, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. IN NO EVENT SHALL ROGUE WAVE SOFTWARE, INC., BE LIABLE, WHETHER IN CONTRACT, TORT, OR OTHERWISE, FOR ANY SPECIAL, CONSEQUENTIAL, INDIRECT, PUNITIVE, OR EXEMPLARY DAMAGES IN CONNECTION WITH THE USE OF THE DOCUMENTATION.

The Documentation is subject to change at any time without notice.

Rogue Wave Software, Inc.

Address: 5500 Flatiron Parkway, Boulder, CO 80301 USA

Product Information: (303) 473-9118 (800) 487-3217
Fax: (303) 473-9137
Web: http://www.roguewave.com
Chapter 1  **Introduction to Objective Grid**

1.1 Welcome to Objective Grid 1
1.2 Product Features 1
1.3 Supported Platforms 3
1.4 Location of Samples 4
1.5 Licensing Restrictions 4
  1.5.1 Stingray Naming Conventions 4
  1.5.2 File Redistribution 5
1.6 Getting Help 5
  1.6.1 Documentation 5

PART I  **User’s Guide**

Chapter 2  **Design Overview**

2.1 Design Introduction 11
  2.1.1 The Style Object 11
  2.1.2 The Range Object 13
  2.1.3 The Parameter Object 14
  2.1.4 Conclusion 14

2.2 Styles Architecture 15
  2.2.1 Style Implementation 15
  2.2.2 CGXGridCore - Styles Interface 17
  2.2.3 CGXStyle - Class Interface 18

2.3 Drawing and Updating 20
  2.3.1 CGXGridCore - Commands Interface 22
2.3.2 Accessor Methods 23
2.3.3 CGXGridCore - Drawing Overridables 25

2.4 Control Architecture 26
2.4.1 Current Cell Interface 26
2.4.2 Current Cell Notifications 26
2.4.3 Predefined Controls 29

2.5 Browser Architecture 31
2.5.1 CGXBrowserGrid - Interface 31
2.5.2 ODBC Browser 33
2.5.3 ADO Browser 34

2.6 Formula Engine 35
2.6.1 Adding Formula Support to Your Application 35
2.6.2 Initializing the Grid With Data and Formulas 36
2.6.3 Changing Cell Data and Formulas at Run Time 36

2.7 Run-Time Initialization 39

2.8 Choosing Between Grid Types 42

Chapter 3  Technical Overview

3.1 Introduction 45

3.2 Class Summary 46

3.3 Drawing Classes 47
3.3.1 CGXGridCore 47
3.3.2 CGXView 48
3.3.3 CGXGridView 48
3.3.4 CGXGridWnd 48
3.3.5 CGXGridHandleView 48
3.3.6 CGXGridHandleWnd 48
3.3.7 CGXGridHint 48

3.4 Control Classes 49
3.4.1 CGXControl 51
3.4.2 CGXBitmapButton 51
3.4.3 CGXCheckBox 51
3.4.4 CGXCheckBoxEx 51
3.4.5 CGXCheckListComboBox 52
3.4.6 CGXComboBox 52
3.4.7 CGXComboBoxWnd 52
3.4.8 CGXCurrencyEdit 53
3.4.9 CGXDateTimeCtrl 53
3.4.10 CGXEditControl 54
3.4.11 CGXHeader 54
3.4.12 CGXHotSpotEdit 54
3.4.13 CGXListBox 55
3.4.14 CGXMaskControl 55
3.4.15 CGXPasswordControl 55
3.4.16 CGXProgressCtrl 55
3.4.17 CGXPushbutton 55
3.4.18 CGXRadioButton 56
3.4.19 CGXRadioButtonEx 56
3.4.20 CGXRichEditCtrl 57
3.4.21 CGXSpinEdit 57
3.4.22 CGXStatic 57
3.4.23 CGXTabbedComboBox 57
3.4.24 CGXTabbedComboBoxWnd 58
3.4.25 CGXVScrollEdit 58
3.4.26 CGXWndWrapper 58
3.4.27 CGXChild 59
3.4.28 CGXButton 59
3.4.29 CGXBitmapButtonChild 59

3.5 Style Classes 60
3.5.1 CGXStyle 60
3.5.2 CGXStylesMap 61
3.5.3 CGXStylesDialog 61
3.5.4 CGXStyleSheet 61
3.5.5 CGXFont, CGXBrush, CGXPen 61
3.5.6 CGXAbstractUserAttribute 61
3.5.7 CGXUserAttribute 61

3.6 Data Structures 62
3.6.1 CGXGridParam 63
3.6.2 CGXData 63
3.6.3 CGXProperties 63
3.6.4 CGXCommand 63
3.6.5 CGXAbstractControlFactory 63
3.6.6 CGXControlFactory 63
3.6.7 CGXGridDropTarget 64
3.6.8 CGXNoOleDataObjectProxy 64
3.6.9 CGXNoOleDataSourceProxy 64
3.6.10 CGXDisplayPropertiesDialog, CGXHeaderFooterDialog,
CGXPrintPropertiesDialog 64
3.6.11 CGXRange 64
3.6.12 CGXRangeList 65
3.6.13 CGXPrintDevice 65

3.7 Browser Classes 66
3.7.1 CGXBrowserGrid 67
3.7.2 CGXBrowserView 67
3.7.3 CGXBrowserWnd 67
3.7.4 CGXBrowseParam 67
Chapter 4  **Getting Started**

4.1 Creating a Starter Application 77
   4.1.1 Using the Objective Grid Static Libraries 77
   4.1.2 Using the Objective Grid Extension DLLs 78
   4.1.3 Converting an Application to Use the Objective Grid Extension DLL 78

4.2 Using Objective Grid in a Dialog Box 79
   4.2.1 Objective Grid vs. ActiveX Controls 79
   4.2.2 Using Objective Grid in a Dialog Template 79
4.2.3 Troubleshooting 81

4.3 Miscellaneous Tips 82
   4.3.1 Checklist for Build Configurations 82
   4.3.2 Objective Grid and VC++ 83
   4.3.3 Objective Grid and Windows Look and Feel 84

4.4 Using Objective Grid in a WinForm 86
   4.4.1 Direct Use of Objective Grid in a WinForm 86
   4.4.2 Use of Objective Grid in a .NET Control 87

Chapter 5  Using Objective Grid

5.1 Cell Value 91
   5.1.1 Filling the Grid with Values 91
   5.1.2 Getting Values from the Grid 92
   5.1.3 Clearing Cell Values 93

5.2 Cell Attributes 94
   5.2.1 Reusing Style Objects 94
   5.2.2 Disabled and Read-Only Cells 94
   5.2.3 Covered Cells 96
   5.2.4 Floating Cells 96
   5.2.5 Merged Cells 97
   5.2.6 Registration and Usage of Base Styles 99
   5.2.7 Registration and Usage of User Attributes 101

5.3 Controls 104
   5.3.1 Registration and Usage of Controls 104

5.4 Columns and Rows 105
   5.4.1 Column and Row Count 105
   5.4.2 Column and Row Sizing 106
   5.4.3 Hidden Columns and Rows 107
   5.4.4 Frozen Columns and Rows 108

5.5 Cell and Window Coordinates 110
   5.5.1 Cell Coordinates 110
   5.5.2 Window Coordinates 111
   5.5.3 Hit-Testing 113

5.6 Grid Lines 114
   5.6.1 Grid Line Drawing 114

5.7 Events 115
   5.7.1 Keyboard Events 115
   5.7.2 Keyboard Shortcuts 116
5.8 Deriving a Grid 117

5.9 Virtual Grids 118
  5.9.1 Supplying Data 118
  5.9.2 Storing Data 120

5.10 Formula Support 121

5.11 Undo and Redo 121
  5.11.1 CGXGridCore - Undo/Redo interface 121
  5.11.2 Transaction-Like Blocks of Commands 122
  5.11.3 Long Operations 122

5.12 Sorting 125
  5.12.1 Basic Sorting 125
  5.12.2 Natural Sorting 125
  5.12.3 Sorting Example 126

5.13 Pre-Built Dialogs for Accessing Objective Grid Objects 126
  5.13.1 Display Settings Dialog 126
  5.13.2 Header / Footer Dialog 129
  5.13.3 Page Setup Dialog 131

5.14 Number Formatting 133
  5.14.1 Value Types 133
  5.14.2 Formatting Number Values for Static Cells 133
  5.14.3 Recognizing Locale Settings in Static Formatting 136

5.15 Miscellaneous 138
  5.15.1 Using CGXDrawingAndFormatting 138
  5.15.2 IntelliMouse Support 139
  5.15.3 Rotated Text in Cells 140
  5.15.4 Character Sets 140
  5.15.5 Direct Copying and Moving of Cells 141
  5.15.6 AutoScrolling 141
  5.15.7 OLE Drag and Drop Integration 142
  5.15.8 Scroll Tip Support 143
  5.15.9 CellTip Support 143

5.16 Tools 143
  5.16.1 Objective Grid Designer 143
  5.16.2 Stingray AppWizard 150

Chapter 6 Reducing the Size of Your Application

6.1 How Objective Grid Affects the Size of Your Application 155
  6.1.1 Using Objective Grid As a Shared DLL 155
  6.1.2 Linking Objective Grid Statically 156
6.2 The Control Factory Class 157
   6.2.1 Registration of Cell Types 157
   6.2.2 Registration of Concrete Implementations for Grid Features 158
   6.2.3 Registration of Window Classes 159

6.3 The Build Wizard 161
   6.3.1 DLL Naming 162
   6.3.2 Building the Libraries 162

6.4 Component Reference 163
   6.4.1 Control Factory Class 163
   6.4.2 Library Linking 176
   6.4.3 Resources 180
   6.4.4 A Note On the Implementation 180

Chapter 7  1stGrid Tutorial

7.1 Introduction 183

7.2 1stGrid - Step 1  185
   7.2.1 Create a New 1stGrid Project 185
   7.2.2 1stGrid AppWizard Options 186
   7.2.3 1stGrid Project Settings 186
   7.2.4 Modify 1stGrid's stdafx.h 187
   7.2.5 Include Objective Grid Resources in 1stGrid 188
   7.2.6 Modify the 1stGrid Application Implementation File 190
   7.2.7 Modify 1stGrid View's Header File 190
   7.2.8 Modify 1stGrid View's Implementation File 191
   7.2.9 Override CMy1stGridView::OnInitialUpdate() 191
   7.2.10 Compile and Run 1stGrid - Step 1 192

7.3 1stGrid - Step 2 195
   7.3.1 Modify 1stGrid View's Menu Resource 195
   7.3.2 Map the 1stGrid Menu Messages 198
   7.3.3 Map the 1stGrid Accelerator Keys 200
   7.3.4 More Formatting 202
   7.3.5 Compile and Run 1stGrid - Step 2 203

7.4 1stGrid - Step 3 205
   7.4.1 Modify the 1stGrid Document's Header File 205
   7.4.2 Modify the 1stGrid Document's Implementation File 205
   7.4.3 Modify the 1stGrid View's Implementation File 206
   7.4.4 Add a Splitter Frame Class for 1stGrid 207
   7.4.5 Change the 1stGrid View to the Splitter Frame Class 208
   7.4.6 Compile and Run 1stGrid - Step 3 209
Chapter 8  DbQuery Tutorial

8.1 Introduction 211

8.2 DbQuery - Step 1 213
  8.2.1 Create a New DbQuery Project 213
  8.2.2 DbQuery AppWizard Options 213
  8.2.3 DbQuery Project Settings 214
  8.2.4 Modify DbQuery's stdafx.h 217
  8.2.5 Include Objective Grid Resources in DbQuery 218
  8.2.6 Modify the DbQuery Application Implementation File 220
  8.2.7 Register a Database 220
  8.2.8 Add a CRecordset Derived Class 223
  8.2.9 Modify DbQuery View's Header File 226
  8.2.10 Modify DbQuery View's Implementation File 226
  8.2.11 Compile and Run DbQuery - Step 1 228

8.3 DbQuery - Step 2 229
  8.3.1 Modify the View's Menu Resource 229
  8.3.2 Map the DbQuery Menu Messages 232
  8.3.3 Compile and Run DbQuery - Step 2 234

8.4 DbQuery - Step 3 236
  8.4.1 Add a New Frame Class 236
  8.4.2 Change the DbQuery View to the New Frame Class 236
  8.4.3 Compile and Run DbQuery - Step 3 237
Chapter 9  **DlgGrid Tutorial**

9.1  Introduction 239
9.2  Create a New DlgGrid Project 241
9.3  DlgGrid AppWizard Options 242
9.4  DlgGrid Project Settings 242
9.5  Modify DlgGrid's stdafx.h 243
9.6  Include Objective Grid Resources in DlgGrid 244
9.7  Modify the DlgGrid Application Implementation File 246
9.8  Create a New CGXGridWnd Derived Class 246
9.9  Put a Custom Control on the Main Dialog Template 247
9.10 Modify CDlgGridDlg's Declaration 248
9.11 Include the Derived Grid's Header 248
9.12 Modify CDlgGridDlg::OnInitDialog() 249
9.13 Compile and Run DlgGrid 250

Chapter 10  **VirtGrid Tutorial**

10.1 Introduction 251
10.2 VirtGrid Overview 251
10.3 The Program 252
    10.3.1 Initialization and Column Count 252
    10.3.2 Row Count 253
    10.3.3 Making the Virtual Connection Between Grid Object and Data Source 253
    10.3.4 VirtGrid Conclusion 255
Chapter 11 Build and Runtime Troubleshooting

11.1 Troubleshooting 257

PART II Programmer’s Guide

Chapter 12 Advanced Design Overview

12.1 Advanced Design Overview Introduction 261

12.2 Objective Grid Defined 262

12.3 Objective Grid Core Architecture In Brief 263

12.4 Objective Grid Control Architecture 264
    12.4.1 The pure CGXControl approach 266
    12.4.2 The Window Wrapper Approach 268

12.5 Control Sharing In the Grid 271
    12.5.1 External State Functions 274
    12.5.2 Current Cell State Functions 274

12.6 Objective Grid Integration With Document/View Architecture 276
    12.6.1 Push Model Details 279

12.7 Undo/Redo Architecture 282
    12.7.1 Other Issues Related to the Undo/Redo Architecture 285

12.8 Objective Grid Style Architecture 286
    12.8.1 The styles map 291
Chapter 13 **Objective Grid Concepts and Features**

13.1 Programming Concepts 293

13.2 Extending Functionality in Objective Grid 293

13.3 Multiple Inheritance 295

13.4 Returning References 296

13.5 Automatic Linking 297

13.6 MFC Messages Versus Objective Grid Virtuals 297

13.7 Objective Grid and Windows XP Visual Styles 298
   13.7.1 Grids 298
   13.7.2 Controls 298
   13.7.3 Giving Objective Grid an Excel XP Look 300

Chapter 14 **The Formula Engine**

14.1 Introduction 301
   14.1.1 What is the Objective Grid Formula Engine? 301
   14.1.2 Implementing the Formula Engine 301
   14.1.3 Adding formula support 302
   14.1.4 Storing expressions and values in cells 304
   14.1.5 Formula Engine Support in CGXGridCore 305
   14.1.6 Drawing 306
   14.1.7 Clipboard 307
   14.1.8 Undo/Redo 308
   14.1.9 Display Formula Expression in Cells 308
   14.1.10 Formula Engine Limitations 309

14.2 CGXFormulaSheet Class 310
   14.2.1 Cell Values 310
   14.2.2 Row and Column Coordinates 311
   14.2.3 Sheet Settings 312
   14.2.4 Recalculation 313
   14.2.5 Named Ranges 313
   14.2.6 Range Specification 314
   14.2.7 Expression Evaluation 314
   14.2.8 Clipboard Cut/Copy and Paste 315
   14.2.9 Cell Operations 315

14.3 Cells and Ranges 317
   14.3.1 Text Cell Values 317
   14.3.2 Numeric Values 318
   14.3.3 Dates and Times 318
   14.3.4 Formulas 319
14.3.5 Built-in Functions 319
14.3.6 Ranges 320
14.3.7 Copying and Moving Cells 322

14.4 Calculations 323
14.4.1 How Objective Grid Calculates 323
14.4.2 Mode of Recalculation 323
14.4.3 Method of Recalculation 324
14.4.4 Iterative Calculations 324
14.4.5 Constraint Checking 325
14.4.6 Precision 325

14.5 Objective Grid Formulas 327
14.5.1 Formula Syntax 327
14.5.2 Formula Values 328
14.5.3 Formula Operators 328
14.5.4 Referencing Other Cells in Formulas 330
14.5.5 Cell Referencing in Objective Grid 330
14.5.6 Constraint Expressions 333
14.5.7 Explicit Dependency 333

14.6 Built-in Functions 334
14.6.1 Mathematical Functions 334
14.6.2 Statistical Functions 335
14.6.3 Conditional Statistical Functions 335
14.6.4 String Functions 335
14.6.5 Logic Functions 335
14.6.6 Digital Logic Functions 335
14.6.7 Financial Functions 336
14.6.8 Date and Time Functions 337
14.6.9 Miscellaneous Functions 338
14.6.10 Embedded Tools 338
14.6.11 Using Objective Grid Built-in Functions 338

14.7 Quick-Reference Guide to Built-in Functions 341
14.7.1 Mathematical Functions 341
14.7.2 Statistical Functions 343
14.7.3 Conditional Statistical Functions 344
14.7.4 String Functions 345
14.7.5 Logic Functions 346
14.7.6 Financial Functions 347
14.7.7 Date and Time Functions 349
14.7.8 Miscellaneous Functions 350
14.7.9 Embedded Tools 351

14.8 Error Messages 353
14.8.1 Types of Errors 353
14.8.2 Error Messages Reference 353
Chapter 15 **Advanced Concepts**

15.1 Objective Grid DLL Architecture 363
   15.1.1 Dynamic Link Libraries 363
   15.1.2 MFC DLLs 363
   15.1.3 Module State 367
   15.1.4 Manage State 371

15.2 Multithreaded Applications 372

15.3 Sharing Extended Functionality Across Classes 372
   15.3.1 Shared Source 372

15.4 Plug-in Components 376
   15.4.1 Motivation 376
   15.4.2 Existing Approaches 376
   15.4.3 The Plug-in Approach 377
   15.4.4 The CGXPluginComponent Class 380
   15.4.5 Implementing a Sample Plug-in Component 380
   15.4.6 Using the IntelliMouse Plug-in Component 384

15.5 Extending the Undo/Redo Mechanism 386

15.6 Print, Print Preview 388
   15.6.1 Implementation 388
   15.6.2 Setting Print and Print Preview Functionality in a CGXGridWnd-Derived Grid 388
   15.6.3 Examples 388
   15.6.4 Related Samples 390

15.7 Using This Product Outside MFC 391

15.8 Internationalization 395
   15.8.1 Fonts 395
   15.8.2 Far Eastern Languages 395
   15.8.3 Resources 395

15.9 Struct Member Alignment 396

Chapter 16 **Optimization**

16.1 Grid Architecture Review 397

16.2 Initializing Formula Grids 398
   16.2.1 Using Formula Sheet Methods 398
   16.2.2 Copying Cells 399
16.3 Populating the Grid 401
  16.3.1 Improving Performance 401

16.4 Speeding Up Computations 404

16.5 Using Profiling in Visual Studio 404

16.6 Reducing Application Size 404

Chapter 17 **CSliderCtrl Tutorial**

17.1 Introduction 405

17.2 CSliderCtrl Overview 405

17.3 The CSliderCtrl Program 406
  17.3.1 The Declaration 406
  17.3.2 Activating and Deactivating the Control 407
  17.3.3 Drawing the Active and Inactive State 409

Chapter 18 **BrwsGrid Tutorial**

18.1 Introduction 413

18.2 BrwsGrid Overview 413

18.3 The BrwsGrid Program 414
  18.3.1 Initialization and Setting Column Count 414
  18.3.2 Supplying the Row Count Dynamically 415
  18.3.3 Managing Sorted Rows and Moved Columns 415
  18.3.4 Supplying the Value To the Grid and Storing the Changes 415
  18.3.5 Deleting Rows 416
Chapter 19 **Dynamic Splitter Window Support**

19.1 Introduction 419

19.2 Process 419

Chapter 20 **Page Break and Scaled Printing Support**

20.1 Introduction 427

20.2 Adding Page Break and Scaled Printing Support 428

Chapter 21 **Database Engine Architecture**

21.1 Introduction to Database Engine Architecture 431

21.2 Database Engine Architecture 431

21.2.1 DBEngine 432

21.2.2 Initializing APIs 432

21.2.3 Fetching APIs 432

21.2.4 Updating APIs 433

21.2.5 Miscellaneous APIs 433

21.2.6 Summary of Database Architecture 433

21.3 Layout Manager 434

21.3.1 Browser-Like Layout 434

21.3.2 Browser-Like Editing 434

21.3.3 Hierarchical Browser Layout Manager 435

21.4 Database Engine Grid 435

21.4.1 Hierarchical Database Engine Grid 435

21.4.2 Initializing the Grid with CGXDBEngine 435

21.5 ADOEngines 436

21.5.1 CGXADOEngine 436

21.5.2 CGXHierADOEngine 436

21.6 Database connectivity with Objective Grid 437

21.6.1 Microsoft Access 437

21.6.2 Microsoft SQL Server 437

21.6.3 Oracle 437

21.6.4 Re-Enabling DAO in Objective Grid 439
Chapter 22 **Grid XML Serialization Architecture**

22.1 Overview 441

22.2 Extensions 441

22.2.1 Formatters for Objective Grid Components 441

22.2.2 Tag Names for Data Components 442

22.2.3 Selective Linking via Build-Wizard 442

22.3 Usage Example 442

22.4 Sample 442

Chapter 23 **XML Read/Write Library**

23.1 Introduction 443

23.2 Library Purpose 443

23.3 Implementation 445

23.4 Macros and Tools Ease Development 448

23.4.1 Sample WRITE_MAP 449

23.5 Usage 450

Chapter 24 **Hierarchical Grid Architecture**

24.1 Introduction 451

24.1.1 Absolute Logical Coordinate Transformation 453

24.2 Regular Grid 453

24.3 Wrapped Grid 455

24.3.1 Seamless Current Cell Movement Between Wrapped Rows 456

24.3.2 Row/Column Layout Interfaces 456

24.4 Multiple Row Type Grid 457

24.4.1 Visually Resizing Columns 458

24.4.2 Row/Column Count API 458
Chapter 25 **The ATL and Objective Grid**

25.1 Introduction 459

25.2 Wrapping Objective Grid In an ATL ActiveX Control 459

25.3 Using Objective Grid Inside ATL Applications 461

25.4 Module State Issues 461

Chapter 26 **Excel Data Exchange Support**

26.1 Introduction 463

26.2 Structure of a BIFF 8 file 464
   26.2.1 Basic Structure of a BIFF File 464
   26.2.2 Record Identifier 465
   26.2.3 Record Length 466
   26.2.4 Structure of the Read/Write Mechanism 466

26.3 Shared and Unshared Data 470
   26.3.1 Shared Objects 470
   26.3.2 Unshared Objective Grid Data 470

26.4 Usage Samples 470
   26.4.1 Sample 1 471
   26.4.2 Sample 2 475

26.5 Advanced Section 478
   26.5.1 Extending the Reader 478
   26.5.2 Extending Tables 482
   26.5.3 Supported Excel Fields 483
   26.5.4 Background Color 483

26.6 Important Notes 485

26.7 Excel Automation with Objective Grid 486
   26.7.1 Enabling Excel Integration in a Grid Project 487
   26.7.2 Testing the ExcelGrid Sample 491
   26.7.3 Benefits of Excel-Grid Integration 491

Chapter 27 **GridEx Feature Pack Extension**

27.1 Overview 493
   27.1.1 Architecture 493
   27.1.2 GridEx Features 494
   27.1.3 System Requirements 494
   27.1.4 Code Source and Sample Locations 494
Chapter 1

Introduction to Objective Grid

1.1 Welcome to Objective Grid

Objective Grid is a full-featured grid control built completely from object-oriented technology. It can be extended quickly and easily, and it provides support for embedding grid objects virtually anywhere that a window can be embedded. It includes a comprehensive set of controls to embed in cells and provides the ability to add your own. Using Objective Grid, you can bind to any data source, including ODBC and ADO.

1.2 Product Features

Objective Grid provides Microsoft Foundation Class (MFC) developers with a feature-rich grid control, implemented as a complete MFC class extension library. A grid control is a user interface component that displays data in rows and columns.

Unlike other grid controls, Objective Grid has been built from the ground up with MFC compatibility as a major goal. The Objective Grid classes fit seamlessly with MFC, and inherit from existing MFC classes such as `CView` or `CWnd` in many cases. Objective Grid is fully compatible with MFC*

* Compatibility with MFC Feature Pack has been partially implemented. Please see GridExReadme.pdf and Stingray Feature Pack Migration.pdf for details.

Objective Grid is compatible with the latest 32-bit and 64-bit releases of Visual Studio, and ability to use it in C++/CLI is provided.

When used as a view, Objective Grid fits seamlessly into the MFC document/view framework. Objective Grid can also be used as a window, enabling the user to embed grid controls in other windows, such as dialogs and pop-ups. In C++/CLI program, Objective Grid can be used with WinForm directly or wrapped into a .NET control usable in any .NET language, such as C# or VB.NET.
Objective Grid includes the following features:

- **Granular implementation** - The Build Wizard gives you the ability to exclude unwanted functionality from the compiled Objective Grid libraries.

- **Wide range of built-in cell types** - Objective Grid offers more than 30 cell types, ranging from simple edit controls to formatted date-time and currency controls.

- **Ability to embed your own controls** - If one of the built-in controls does not fit your needs, then Objective Grid enables you to embed your own controls.

- **Database connectivity** - Objective Grid ships with built-in support for ODBC and ADO databases. It also provides the ability to create new browser grids for unsupported or proprietary databases.

- **Undo/Redo** - Objective Grid fully supports Undo/Redo for built-in grid operations, including transactions and rollbacks. The architecture can be easily extended to include application-specific operations.

- **Printing and print preview** - Objective Grid supplies full MFC print and print preview support for `CView`-derived grids and additional regular DLL for print and print preview support for `CWnd`-derived grids. A dialog is supplied for manipulating page header and footer text.

- **Find/Replace** - Objective Grid provides support for finding and replacing cell data.

- **Cut/Copy/Paste** - Objective Grid supports cut/copy/paste using the Windows clipboard as well as an internal direct cut/copy/paste.

- **Pre-built style dialogs** - A set of pre-built dialog boxes is supplied to allow your end user access to the styles architecture. These dialogs enable the end user to make changes in the appearance of the grid.

- **Object-oriented cell architecture** - Objective Grid uses a control sharing architecture in which each type of cell shares a single control with all similar types. Cell data is stored separately from the cell control. This system saves space and resources.

- **Internationalization** - Objective Grid supports both Unicode and DBCS. Resource files for Objective Grid are supplied in English, French, Dutch, and German.

- **Excel-like interface** - Objective Grid enables the programmer to mimic many of the user interface features in Microsoft Excel.

- **Floating and merged cells** - Objective Grid provides support for both floating and merged cells. Floating cells are cells that automatically grow over adjacent cells to accommodate text too large for a single cell. Merged cells are adjacent cells that contain the same value and are drawn as a single large cell.

- **Dynamic splitter window as in Excel** - Objective Grid has Excel-like tabs and splitter windows. End users can create windows using splitter bars, just as in Excel.

- **Page Break UI as in Excel** - Objective Grid has a user interface that enables your end user to set page breaks.

- **Formula support** - The Objective Grid formula engine ships with more than 200 built-in worksheet functions. If that's not enough, the formula engine can be extended with your own custom worksheet functions.
Ships with full source code - Objective Grid ships with full source code.

HTML persistence - With the standard edition of Objective Grid, you can save a rectangular block of the grid as an HTML table.

```csharp
CGXHtmlW Html(pGridView);
Html.WriteToFile("Output.html");
```

XML Stream Binding - Objective Grid offers support for reading and writing XML. XML is ideal for exchanging information over the Internet, while Objective Grid is ideal for presenting and editing information. Together, they make a powerful combination. You can bind Objective Grid to an XML data source, enabling you to read and write grid data in XML form.

Scaled Printing on A x B pages - Scaled printing allows users to specify the number of pages on which to print. The grid is scaled to fit the specified dimensions.

Tree-Grid Functionality - You can create advanced hierarchical grids with a variety of different styles. With ADO support, you can bind a Tree-Grid directly to your database.

Improved ATL Compatibility - If you’ve ever tried to use Objective Grid in an ATL project, you’ll appreciate the work we’ve done to make Objective Grid ATL-friendly. We now provide an Object Wizard, as well as base implementation support that can generate a custom ATL-based Objective Grid ActiveX control. And we’ve made it easier to use Objective Grid inside an ATL project by resolving global data and module state issues you may have encountered in the past.

Works with supported versions of Microsoft Visual Studio - All of the Grid libraries, wizards and other external tools can be compiled under supported versions of Microsoft Visual Studio. For more information, see Section 4.3.2, “Objective Grid and VC++.”

Supported on Windows XP, Vista, and 7 - All OG applications executed under XP and Vista can take advantage of their visual styles based on Windows themes and skins. For details, see Section 4.3.3, “Objective Grid and Windows Look and Feel.”

Excel integration support - Objective Grid includes support for reading and writing Microsoft Excel files in BIFF8 format, and Microsoft Automation for runtime interaction between Grid and Excel.

1.3 Supported Platforms

For a list of supported operating systems and compilers, see http://www.roguewave.com/products/stingray.aspx, then click on the link “Supported Platforms” to download a PDF.
1.4 Location of Samples

Stingray Studio ships the most basic and commonly used samples with the product itself. The less commonly used and more specialized samples have been removed from the product distribution, but are available from the Rogue Wave web site.

If you are looking for a sample and it does not appear under `<stingray-installdir>\Samples\Grid\<sample-name>`, you can download the sample bundle from the Knowledge Base on the Rogue Wave Web site, as described in Section 3.6.1, “Location of Sample Code,” in the Stingray Studio Getting Started Guide.

1.5 Licensing Restrictions

Please read the license agreement that was shipped with this package. You are bound by the licensing restrictions contained in that document. Do not use this product unless you can accept all the terms of the license agreement.

You can use all the files accompanying this product for development of an application. You can distribute the Objective Grid Dynamic Link Libraries (DLLs) according to the terms of the license agreement. Your applications can also statically link to the grid, in which case you do not need to redistribute any Objective Grid files.

1.5.1 Stingray Naming Conventions

The library naming conventions are illustrated below for the default library names.

Figure 1 – Library Naming Conventions

```
OG ## a s u d

Debug Build
Unicode Build
Stingray DLL
MFC DLL (AFXDLL)
Product Version Number

Base Name
```
### 1.5.2 File Redistribution

The following files can be shipped with your application when needed, where `<ver>` stands for the current product version number:

<table>
<thead>
<tr>
<th>File</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>OG&lt;ver&gt;as*.dll</td>
<td>Win32, x64</td>
</tr>
<tr>
<td>OG&lt;ver&gt;us*.dll</td>
<td>Win32, x64, Unicode</td>
</tr>
</tbody>
</table>

To get a better understanding of how these filenames are generated and used, please refer to Section 1.7.1, “Library Dependencies for Distribution,” in the *Stingray Studio Getting Started Guide*.

You may not redistribute Objective Grid header files, source files, or sample files. You also may not expose the Objective Grid programming interface in your program (e.g., a DLL or OCX/ActiveX that lets a user create a grid).

### 1.6 Getting Help

Several avenues of help are available to you when working with Objective Grid.

#### 1.6.1 Documentation

The following documents are available for Objective Grid:

- **User's and Programmer's Guide** - This manual. The *User's Guide* is an introductory how-to manual for Objective Grid. Its main focus is to introduce the user to Objective Grid and to provide a foundation for using Objective Grid “out-of-the-box.” There are several tutorials included to help new Objective Grid users learn how to create Objective Grid applications quickly.

  The *Programmer’s Guide* is an advanced manual for Objective Grid. It includes a discussion of the design and architecture of Objective Grid and an advanced how-to section. There are several advanced tutorials to help users create new browser grids and cell controls.

- **Class Reference** - The class reference is a detailed description of the classes and methods in Objective Grid.

- **Samples** - Objective Grid ships with a wide variety of samples.

- **Knowledge Base** - The Rogue Wave Knowledge Base contains a large body of useful information created by the Support Services team. It has been created to provide our customers with easy access to technical information. Knowledge Base entries range from common programming problems to complete sample programs and are constantly being updated and expanded. This information is available to any user of the Rogue Wave Web site, and no login or registration is required.

  [kb.roguewave.com/kb/](kb.roguewave.com/kb/)
Objective Grid Designer - The Objective Grid Designer comes with a Help file (ogdesign.chm) that maps options in the Objective Grid Designer to API calls in Objective Grid.

### 1.6.1.1 Available Formats

The following is a list of available documentation formats. After each format is a list of the documents available in that format.

- **Portable Document Format (PDF) documents** - The following documents are located in the Docs subdirectory of your Objective Grid directory.

- **HTML help 1.0** - The following are available as compiled help files (.chm).
  - User's and Programmer's Guide - ogug.chm
  - Class Reference - ogref.chm

- **HTML help 2.0** - The following are available as compiled help files (.hxs) integrated with the MSDN Infoviewer.
  - User's and Programmer's Guide - ogug.hxs
  - Class Reference - ogref.hxs

For more information on the documentation, including all Stingray documentation, an index to the Help files, and document type conventions, see Section 1.4, “Product Documentation,” in the Stingray Studio Getting Started Guide.

### 1.6.1.2 Common Terms

**Base Style** - Base styles are grid-wide styles that make it possible to group specific kinds of cells and give them similar attributes. The predefined base styles are: row-header-style, column-header-style and standard-style. Row header cells inherit their attributes from row-header-style. Column headers inherit from column-header-style. Standard-style is the base style for all cells in the grid.

**Cell** - Cells display information in the grid. Each cell has a unique coordinate (row, column). Cells are associated with a control and a style object. The control is responsible for handling user events and drawing the information provided through the style object.

**Control** - Controls handle the interface between the end user and the grid. Each cell is associated with a control. The control interprets user events and is responsible for drawing the cell.

**Control Child** - Controls can have small children in the cell. For example, the spin control has an up-arrow button child and a down-arrow button child. The control child is normally a small rectangular area in the parent control’s area.

**Covered Cells** - Objective Grid lets you cover cells. This means that one cell can span several other cells. This is very useful for headings in reports.
Current Cell - The grid manages the current cell as the user navigates through the grid by clicking or using arrow keys. The current cell lets the user modify the cell’s contents through its associated control. The end user can interact directly with the control.

Data source - Data source is a general term that can mean either an ODBC/DAO/ADO query result, a database, or any other external data structure or medium.

Properties - Properties are settings in the grid that can be modified with pre-built dialogs and can be loaded/archived using the registry/profile. Properties are maintained by the CGXProperties class.

Range - A range defines a rectangular area of cells in the grid. A range is specified through a top and bottom row, and left and right columns. Ranges can represent a selection of cells, columns, rows, or all cells.

Style - A style contains all the information necessary for formatting a cell. A style consists of several attributes such as text color, borders, control type, and font. Each cell determines the style information at run time and passes this information to the control for drawing the cell.

Workbook - A workbook lets the user switch between several views connected to the same document by clicking on a tab at bottom-left of the window.

Worksheet - Worksheet is used to refer to each of the individual views displayed in a workbook.
Chapter 2

Design Overview

2.1 Design Introduction

Objective Grid has an object-oriented architecture. A brief overview of some of these objects and their place in the Objective Grid world helps when working with Objective Grid for the first time.

Three essential objects form the core of the Objective Grid architecture:

- The Style object
- The Range object
- The Parameter object

Each of these objects is discussed in detail in this section.

2.1.1 The Style Object

The style object forms the core for formatting cells in the grid. The style can be seen as an object that completely defines a cell. This means that the style object has all the information necessary for the grid object to draw the cell as well as manage its interaction with users. This includes information such as the font used to draw the cell, the color of the interior, the size of the font, the value displayed, and the type of control in the cell, for example. Style objects for each cell are stored internally by the grid. These can be modified and stored with calls to SetStyleRange(). The sequence of calls to store a style looks like this:

```c++
// Declare a style and range object
CGXStyle style;
CGXRange range(1,2,3,4);

// Change it to get the appearance that you want
style.SetInterior(RGB(0,0,255));

// Set the value of the cell
style.SetValue(_T("Test"));

// Store the style
SetStyleRange(range, style);
```
_T() is an MFC macro used to create strings that are portable for ANSI and Unicode builds.

The style object is also capable of initializing itself from other objects, each with a specific function. For example, the font object defines all font specific attributes.

```cpp
CGXFont font;
font.SetFaceName(_T("Times New Roman"));
Style.SetFont(font);
```

On a related note, it is common to see code like the following in Objective Grid sample programs.

```cpp
SetStyleRange(CGXRange().SetCols(1),
    CGXStyle()
    .SetFont(CGXFont().SetFaceName("Arial")
    .SetValue(_T("test"))
    );
```

This code may seem cryptic at first, but it is actually simple once you understand the concept. Each of these calls returns a reference to the object itself. This allows the programmer to chain several method calls together without explicitly referencing the object each time.

For example, `Style.SetInterior()` returns a reference to `CGXStyle`, making it possible for us to call `Style.SetInterior().SetValue()` all in one stroke.

If you want to remove only the changed background color for a cell style put in place with a `SetInterior()` call, you can simply call `SetIncludeInterior(FALSE)`.

At this point, it should be clear that cells in a grid are defined by internally stored style objects. These objects can be changed with calls to `SetStyleRange()`.

The styles architecture in Objective Grid goes a little further. Objective Grid divides the styles that it stores into three categories:

- A system-wide standard style.
- A set of user-defined base styles.
- Cell-specific styles.

When the grid draws a cell, it gets the style data in a certain order.

1. The cell-specific style is obtained. The cell-specific style settings override all other style settings.
2. The row and column styles are used to fill any attributes left empty by the cell-specific style.
3. The table-wide style is used to fill any attributes left empty by the row or column styles.
4. The standard grid style is used to continue filling holes.
5. If any base styles are available they are applied. It is important to note that base style settings will always override the grid-wide styles.

Remember, the more specific a style is, the more overriding power it has. Suppose the grid system interior color is white and a base style specifies an interior color of red. The base style color will override the standard style. In short, this architecture allows the grid to have a default initialization...
hierarchy that is very flexible. This is the basis of the grid virtual mode of operation. The process of gathering these styles is loaded with virtuals that can be overridden to handle the style initialization in a dynamic manner. This is what drives the virtual mode.

### 2.1.2 The Range Object

This object defines a rectangular range of cells in the grid. It can be set to represent an entire column, an entire row, or the entire table. Explicit coordinates can also be initialized with the top, bottom, left, and right members of the `CGXRange` object.

Use one of the following methods to specify the range of cells.

- Use `CGXRange::SetRows()` to specify a range of rows.
- Use `CGXRange::SetCols()` to specify a range of columns.
- Use `CGXRange::SetTable()` to specify the full table.
- Use `CGXRange::SetCells()` to specify individual cells.

One common point of confusion is the difference between the `SetCells()` call and the `SetRows()`, `SetCols()`, and `SetTable()` calls. `SetCells()` defines a rectangular region with well defined top-left and bottom-right cells. This is very similar to defining a rectangular region on the screen using `CRect`.

`SetRows()`, `SetCols()`, and `SetTable()` define regions without a definite top-left or bottom-right cell. Instead they define the range as being entire rows, entire columns, or the entire table, independent of the actual grid dimensions. This means that as the dimensions of the grid grow or shrink, the realized dimensions of the range will grow or shrink. For example, let’s say there is a grid with 10 rows and 10 columns, and a range has been defined using `SetCols(2,4)`. The realized coordinates of this range are top = 0, bottom = 10, left = 2, and right = 4.

**Figure 2 – CGXRange::SetCols(2,4) in a 10x10 grid**

If the last three rows of the grid are removed, the realized coordinates of the same range are now top = 0, bottom = 7, left = 2, and right = 4. Note that no changes were made to the original range. It is still defined as `SetCols(2,4)`.  

---

Chapter 2   Design Overview 13
Another common misconception is that you can combine calls to `SetRows()` and `SetCols()` to define a cell or range of cells. This is not true. Calling `SetRows()` defines the range as the specified rows. Following up with a call to `SetCols()` redefines the range as the specified columns. It does not take the union of the two ranges. To define a cell or range of cells, you must explicitly use `SetCells()`.

### 2.1.3 The Parameter Object

The parameter object is similar to the document in the MFC document/view paradigm. The parameter object contains or points to all necessary data for persisting the state of the grid object. Parameter objects can be stored in documents and serialized or can be used as members of other classes. Parameter objects can also be shared between grid objects.

### 2.1.4 Conclusion

The above discussion should make the basic structure of Objective Grid clear. There are several additional objects that Objective Grid uses, but the underlying design philosophy is similar. Understanding these concepts will help in understanding and working with Objective Grid.

The following sections provide a detailed description of the grid architecture.
2.2 Styles Architecture

As you probably noticed in the overview of Objective Grid classes, styles are the key to the display of the grid and also define most of its behavior. Let’s proceed by taking a look at how the styles architecture is implemented. We can then discuss how it ties in with the drawing of the grid.

2.2.1 Style Implementation

The CGXStyle class contains all the information necessary for formatting a cell. A style consists of several attributes, such as text color, borders, control type, and font. All of these styles can be modified by the end user via the CGXStyleSheet dialog.

A very important feature of CGXStyle is support for combining style objects. For example, you can copy only those attributes from one style to a second style that are not initialized in the second style. Objective Grid uses this feature to enable a kind of inheritance. By specifying a base style, you can tell Objective Grid that it should inherit attributes from a base style at run time.

Attributes in the CGXStyle class are combined with an include bit. This include bit is TRUE when an attribute is initialized. If it is FALSE, the attribute is not initialized. When drawing the grid, Objective Grid fills up all uninitialized attributes of the cell style object with values inherited from the base styles.

CGXStyle also supports user-defined style attributes. You can also extend the CGXStyle class with additional attributes. The end user can change these attributes through the CGXStyleSheet. Each CGXStyle object maintains a map of user attributes and provides a method to change their values.

The following attributes are provided by the CGXStyle class:

- Value with the cell’s text
- Control ID
- Base style
- Text color
- Cell pattern and color using a LOGBRUSH object
- Borders with LOGPEN objects
- Font as a CGXFont object
- Horizontal alignment
- Read-only state
- Enabled/disabled state (disabled cells cannot become the current cell)
- Auto-size feature (this allows a cell to grow automatically when the user enters large text)
- Maximum length of the text
- 3D-effect (raised, inset, normal)
- Vertical scrollbar specifies if a multiline edit control should display a vertical scrollbar
- Wrap text (also known as Wordbreak)
- Allow enter specifies if the edit control should insert a new line when the user presses Enter
- Choice list specifying the list of items for a combo-box or list-box, or the text for radio-buttons, push buttons, or checkboxes
- Tri-state property for checkboxes
- A user-defined item data pointer, similar to the item data pointer in list boxes.

Style objects are created for those cells that should be drawn and initialized at run time in the CGXGridCore member function ComposeStyleRowCol(). ComposeStyleRowCol() takes a row, a column, and a pointer to a style object as arguments. ComposeStyleRowCol() first calls the GetStyleRowCol() function to get any cell-specific style attributes. ComposeStyleRowCol() adds to the style object any inherited attributes from the row, the column, and the entire grid. Finally, the styles map is checked for any base styles.

The following simplified version of ComposeStyleRowCol() shows how styles are composed.

```cpp
void CGXGridCore::ComposeStyleRowCol(ROWCOL nRow, ROWCOL nCol, CGXStyle* pStyle)
{
    // Copy the cell style first
    GetStyleRowCol(nRow, nCol, *pStyle, gxCopy);

    // Apply the row style next
    GetStyleRowCol(nRow, 0, *pStyle, gxApplyNew, -1);

    // Apply the column style
    GetStyleRowCol(0, nCol, *pStyle, gxApplyNew, -1);

    // Apply the table style
    GetStyleRowCol(0, 0, *pStyle, gxApplyNew, -1);

    // Inherit any base styles
    pStyle->LoadBaseStyle(GetStylesMap());

    // Finally apply the standard style
    pStyle->ChangeStyle(*m_pStyleStandard, gxApplyNew);
}
```

You may be wondering why all of the access to styles is controlled by one function. This technique allows the developer to bind or dynamically tie the grid object to data such as a database or a live data feed. All the developer has to do is override the virtual CGXGridCore::GetStyleRowCol() function. By centralizing all style operations in one virtual function, the developer can modify the grid in any way imaginable at run time.

What if the developer wants the user to be able to dynamically modify the data? The grid provides a virtual function CGXGridCore::StoreStyleRowCol() that is called before data is stored. By overriding this function, the developer can intercept the end user's changes.
2.2.2 CGXGridCore - Styles Interface

The following CGXGridCore member functions define the interface for style operations.

**ChangeBaseStyle()**

Changes a specified base style. All base styles are maintained by the styles-map object.

**ChangeColHeaderStyle()**

Changes the column-header-style.

**ComposeStyleRowCol()**

Composes the style for a specific cell by inheriting unspecified attributes from its base styles as explained before.

**GetColStyle()**

Determines the base style for a specific column.

**GetExpressionRowCol()**

Returns the cell value as a string if the cell is text or a number. If the cell is a formula expression, `GetExpressionRowCol()` will return the expression string.

**GetRowStyle()**

Determines the base style for a specific row.

**GetTableStyle()**

Determines the base style for the table.

**GetValueRowCol()**

Determines the value for a cell. If the cell is a formula expression, the result of the evaluated expression will be returned.

```c
// Get the number at cell 1,1
double dValue = atof(GetValueRowCol(1,1));

// Get the string to cell 1,2
CString sValue = GetValueRowCol(1,2);
```

**ChangeRowHeaderStyle()**

Changes the row-header-style.

**ChangeStandardStyle()**

Changes the standard-style settings.

**SetExpressionRowCol()**

Lets you assign expressions, numbers, or strings to a cell. Objective Grid will interpret the given string and determine if it is a formula expression, date/time string, number, or text.
SetStyleRange()

Applies a given style to the specified range of cells. With `SetStyleRange()`, you can apply a style to a specific cell, rows, columns, or the entire table.

```csharp
// Apply style to range of cells
SetStyleRange(CGXRange(1,1,2,2), style);

// Apply style to columns. Cells in the column will be
// filled up with the column's style.
SetStyleRange(CGXRange().SetCols(1,2), style);

// Apply style to rows. Cells in the row will be
// filled up with the row's style.
SetStyleRange(CGXRange().SetRows(1), style);
```

SetValueRange()

Applies a value to a range of cells.

```csharp
// Apply a number to cell 1,1
SetValueRange(CGXRange(1,1), 2.22);

// Apply a string to cell 1,2
SetValueRange(CGXRange(1,2), "Hello");
```

The following are overridable methods:

**GetStyleRowCol()**

Called to retrieve the pure style for the specified cell. A pure style does not inherit attributes from its base styles.

**StoreStyleRowCol()**

Called to store the pure style for the specified cell into the associated data object.

### 2.2.3 CGXStyle - Class Interface

The `CGXStyle` class provides accessor methods for all its attributes. The key point is that attributes can be marked as uninitialized. Attributes that are uninitialized will be filled up using base style attributes at run time when drawing the grid.

Each attribute state can be determined or modified with four methods. An example is the text color.

**GetIncludeTextColor()**

Returns the include bit. This is `TRUE` if the attribute is initialized and `FALSE` if it is not.

**SetIncludeTextColor()**

Changes the include bit.

**TextColor()**

Returns the text color. The method will assert if the attribute is not marked as initialized.
SetTextColor()

Changes the text color. This method automatically marks the attribute as initialized.

The various style member functions can be cascaded together to make a very clear and concise block of code. For example, the following code illustrates the formatting of some Objective Grid columns:

```cpp
SetStyleRange(CGXRange().SetCols(2, 4),
              CGXStyle()
              .SetHorizontalAlignment(DT_LEFT)
              .SetVerticalAlignment(DT_BOTTOM)
              .SetBorders(gxBorderLeft,
                          CGXPen().SetStyle(PS_DOTTED)
                          .SetFont(CGXFont()
                                    .SetSize(10)
                                    .SetBold(TRUE)
                          )
              .SetFont(CGXFont())
              .SetControl(GX_IDS_CTRL_COMBO)
              .SetChoiceList("one\ntwo\nthree\nfour\nfive\n");
```

The `CGXStyle` class allows you to operate and combine style objects. The most important member function is `ChangeStyle()`.

ChangeStyle()

Used by Objective Grid to compose style objects.

`ChangeStyle()` can perform the following operations with style objects:

- **gxApplyNew** - **Apply only new attributes.** Only those attributes that are included in the source style and not yet included in the destination style are copied from the source style to the destination style.
- **gxOverride** - **Override included attributes.** Those attributes which are included in the source style (whether they are included in the destination style or not) are copied from the source style to the destination style.
- **gxCopy** - **Copy the style.** All attributes are copied from the source style to the destination style. Attributes that are not included in the source style will be removed from the destination style object.
- **gxExclude** - **Reset attributes.** Attributes that are included in the source style will be reset in the destination style. When the grid draws, the reset attributes will be filled using the base style attribute settings.

`SetStyleRange()` takes an argument that lets the developer specify the operation to be performed on the style object. This parameter is internally passed to `ChangeStyle()`. The default argument is `gxOverride`. 
SetStyleRange(CGXRange(1,1,5,5),
  CGXStyle()
  .SetValue("xxx"),
  gxCopy);

// Next call: The interior and the font attribute
// will be applied to the cell only if they have not
// yet been initialized for the cells:
SetStyleRange(CGXRange(1,1,5,5),
  CGXStyle()
  .SetInterior(RGB(255,255,255))
  .SetFont(CGXFont().SetBold(TRUE)),
  gxApplyNew);

2.3 Drawing and Updating

The MFC document/view architecture upon which the grid is built takes care of getting paint messages from Windows to Objective Grid via the CView::OnDraw() routine. When the grid object draws, it first calculates the range of cells that need to be drawn. Then it creates a style object for each cell in that range and initializes the cell styles with ComposeStyleRowCol(). After all the style information has been gathered, the inactive cells are drawn by the grid object and then the active cell control draws itself.

In addition to drawing in response to standard Windows paint messages, the grid object also has to keep the document and views synchronized. The MFC document/view model handles most of the updating work for the grid.

CGXGridView can be used with a document or stand-alone. If the view is associated with a document, it is necessary to update all views in response to a change of the document data.

If the user interacts with the grid by typing text in a cell and moving the current cell, the view will call a command to change the value of the current cell. All associated views have to update this cell.

To provide maximum flexibility, the grid provides three overridable functions for each command in the update path. The overridable functions are marked in Figure 4 with M1, M2, and M3.
The developer can intercept updates at the command stage, the storing stage, and even the update stage. For example, changing the value of a cell is done by calling `SetStyleRange()`. This method calls `StoreStyleRowCol()` to store the cell contents and `UpdateStyleRange()` to redraw the cell. The three member functions are all virtual and the member function that reads the grid’s data is also virtual.

By leveraging the document/view architecture and making sure that the programmer can override the functions involved in updating at every critical stage, the uses of Objective Grid have no limit.
2.3.1 CGXGridCore - Commands Interface

The CGXGridCore class provides the following commands. These commands are all associated with a Store-method and an Update-Method as explained above.

HideCols()

Hide/restore the specified columns. If you hide columns and restore them later, the column widths are restored to their old values.

HideRows()

Hides/restores the specified rows. If you hide rows and restore them later, the row heights are restored to their old values.

InsertCols()

Inserts columns.

InsertRows()

Inserts rows.

MoveCols()

Moves columns.

MoveRows()

Moves rows.

RemoveCols()

Removes columns.

RemoveRows()

Removes rows.

MoveCells()

Moves cells from a given source range to a new destination.

CopyCells()

Duplicates cells from a given source range and inserts them at a given destination.

SelectRange()

Turns the selection on/off for a range of cells.

SetColCount()

Inserts or removes columns by calling either InsertCols() or RemoveCols().

SetColWidth()

Changes the column-widths for specific columns.
SetDefaultColWidth()

Changes the default column-width.

SetDefaultRowHeight()

Changes the default row-height.

SetFrozenCols()

Changes the number of frozen columns and columns to be used for row headers.

SetFrozenRows()

Changes the number of frozen rows and rows to be used for column headers.

SetReadOnly()

Changes the read-only state.

SetRowCount()

Inserts or removes rows by calling either InsertRows() or RemoveRows().

SetRowHeight()

Changes the row-height for a specific row.

SetStyleRange()

Applies a given style to the specified range of cells. See previous section.

SetZoom()

Changes the zoom-factor.

### 2.3.2 Accessor Methods

The following methods encapsulate the access to data and you can easily override them when needed:

GetColCount()

Returns the number of columns in the grid.

GetColWidth()

Returns the width of the specified column.

GetDefaultColWidth()

Returns the default column-width.

GetDefaultRowHeight()

Returns the default row-height.
GetFrozenCols()

Returns the number of frozen columns.

GetFrozenRows()

Returns the number of frozen rows.

GetHeaderCols()

Returns the number of columns to be used for row headers.

GetHeaderRows()

Returns the number of rows to be used for column headers.

GetRowCount()

Returns the number of rows in the grid.

GetRowHeight()

Returns the height of the specified row.

GetStyleRowCol()

Retrieves the pure style for the specified cell (not inheriting attributes from its base styles).

GetZoom()

Returns the zoom factor of the current view.

IsReadOnly()

Returns the read only state.

StoreColWidth()

Stores the width for a specific column.

StoreDefaultColWidth()

Stores the default column-width.

StoreDefaultRowHeight()

Stores the default row-height.

StoreHideCol()

Stores the hidden state for the specified column.

StoreHideRow()

Stores the hidden state for the specified row.

StoreInsertCols()

Inserts columns into the data structure.
**StoreInsertRows()**

Inserts rows into the data structures, such as the row-heights-map, the cells-map, and the range lists for selected and covered cells.

**StoreMoveCols()**

Moves columns in the data structure.

**StoreMoveRows()**

Moves rows in the data structure.

**StoreMoveCells()**

Moves cells from a given source range to a new destination.

**StoreCopyCells()**

Duplicates cells from a given source range and inserts them at a given destination.

**StoreReadOnly()**

Stores the read-only attribute.

**StoreRemoveCols()**

Removes columns from the data structure.

**StoreRemoveRows()**

Removes rows from the data structure.

**StoreRowHeight()**

Stores the height for a specific row.

**StoreStyleRowCol()**

Stores the pure style for the specified cell into the associated data object.

**StoreZoom()**

Stores the zoom factor.

---

### 2.3.3 CGXGridCore - Drawing Overridables

**OnGridDraw()**

Called when the grid needs to be drawn.

**OnDrawTopLeftBottomRight()**

Called to draw a specified range of cells.

**OnDrawItem()**

Called to draw an individual cell.
2.4 **Control Architecture**

To completely comprehend the way Objective Grid draws, let’s take a look at how it implements and manipulates the controls for each cell.

Each composed style object for a cell contains an identifier for a control. Like any other style attribute, this identifier can be changed at run time allowing the developer maximum flexibility. The `CGXGridCore` class maintains a map of registered control types and identifiers. In the `CGXGridCore::OnGridInitialUpdate()` member function, the developer can register any additional control types with their identifiers by calling `RegisterControl()`.

Another interesting facet of the control implementation is the concept of the current cell. Objective Grid manages the current cell as the user navigates through the grid by clicking or using arrow keys. When the user changes the current cell, the grid resets the attributes of the control bound to the previous current cell and initializes the control of the new current cell. Finally, Objective Grid passes all other events to the control so the end user can interact directly with the control. When the user changes the current cell, the grid object takes over again and the current cell process starts over.

### 2.4.1 Current Cell Interface

`CGXGridCore` offers several methods that let the developer manipulate the current cell:

- **SetCurrentCell()**
  
  Selects the current cell at the given coordinates.

- **GetCurrentCell()**
  
  Returns `TRUE` if a current cell exists and retrieves the cell coordinates of the current cell.

- **IsCurrentCell()**
  
  Returns `TRUE` if a current cell exists or is the specified cell.

- **MoveCurrentCell()**
  
  Selects a new current cell based on an offset from the existing current cell in a specified direction.

- **TransferCurrentCell()**
  
  Stores and deactivates the current cell or reinitializes the current cell’s contents.

### 2.4.2 Current Cell Notifications

There are many notification methods the developer can override in a `CGXGridCore`-derived class. Many of the notification methods have related virtual methods in the `CGXControl` class. You can choose whether you want to override the method for all cells in the grid or only for specific controls.
CanSelectCurrentCell()

Called to determine if the specified cell can be selected or deselected. If the cell is deselected, CanSelectCurrentCell() calls OnValidateCell(); otherwise it returns TRUE.

OnCanceledEditing()

Called after the current cell’s content is canceled and the control has been reinitialized with the previous cell’s data.

OnCancelEditing()

Called when the user presses <ESC> on the current cell. You should return TRUE if the control can be deactivated. If you return FALSE, the control cannot be deactivated.

OnClickedButtonRowCol()

Called when the user presses some sort of button (e.g. a pushbutton, hotspot, or radio button) inside of a grid cell.

OnDeleteCell()

Called when the user presses <DELETE> in the inactive current cell. Returns TRUE if the cell's data can be deleted.

OnEndEditing()

Called when the user confirms the contents of the current cell and they are valid. You should return TRUE if the data can be accepted. If you return FALSE, the data will not be accepted, and the cell will remain in edit mode.

OnInitCurrentCell()

Called when current cell is initialized. CGXControl::Init() fires this event.

DoLButtonDown()

Delegates the standard MFC OnLButtonDown() event from (CGXGridWnd and CGXGridView) to the correct spot of the grid.

OnLButtonClickedRowCol()

Called when the user clicks and releases the left mouse button in a cell.

OnLButtonDownCol()

Called when the user double-clicks the left mouse button in a cell.

OnLButtonHitRowCol()

Called when the user clicks the left mouse button in a cell. Subsequent calls occur when the mouse is dragged and when the user releases the mouse button.

OnLeftCell()

Called after the grid has deselected the old current cell. You may override this method and redraw the cell or row. If you return FALSE, the cell will not be redrawn by the grid. If you return TRUE (default), the grid will refresh the cell itself.
OnMButtonClickedRowCol()  
Called when the user clicks and releases the middle mouse button in a cell.

OnMButtonDblClkRowCol()  
Called when the user double-clicks the middle mouse button in a cell.

OnMButtonHitRowCol()  
Called when the user clicks the middle mouse button in a cell. Subsequent calls occur when the mouse is dragged and when the user releases the mouse button.

OnModifyCell()  
Called when the user changes text in the current cell.

OnMovedCurrentCell()  
Called when a new current cell has been selected, but the change is not yet visible.

DoRButtonDown()  
Delegates the standard MFC OnRButtonDown() event from (CGXGridWnd and CGXGridView) to the correct spot of the grid.

OnMButtonClickedRowCol()  
Called when the user clicks and releases the right mouse button in a cell.

OnRButtonDblClkRowCol()  
Called when the user double-clicks the right mouse button in a cell.

OnRButtonHitRowCol()  
Called when the user clicks the right mouse button in a cell. Subsequent calls occur when the mouse is dragged and when the user releases the mouse button.

OnStartEditing()  
Called when the user presses a key or the mouse on the current cell. Returns TRUE if the control can be activated for editing.

OnValidateCell()  
Called after the user has left a modified cell and before another cell is selected. OnValidateCell() can be overridden to perform cell-level validation.

ProcessKeys()  
Interprets keystrokes sent either from the current cell’s control or the grid-window.
### 2.4.3 Predefined Controls

The following control IDs are predefined in Objective Grid. You can use one of the following control IDs in a call to a `SetStyleRange()` command, without explicitly registering them. Objective Grid will register the control automatically as needed.

<table>
<thead>
<tr>
<th>Control ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GX_IDS_CTRL_EDIT</td>
<td>Edit control</td>
</tr>
<tr>
<td>GX_IDS_CTRL_SCROLLEDIT</td>
<td>Edit control with a vertical scroll bar</td>
</tr>
<tr>
<td>GX_IDS_CTRL_HOTSPOT</td>
<td>Hot spot or drop edit</td>
</tr>
<tr>
<td>GX_IDS_CTRL_SPINEDIT</td>
<td>Spin edit</td>
</tr>
<tr>
<td>GX_IDS_CTRL_RICHEDIT</td>
<td>Rich edit control</td>
</tr>
<tr>
<td>GX_IDS_CTRL_MASKEDIT</td>
<td>Mask edit control</td>
</tr>
<tr>
<td>GX_IDS_CTRL_STATIC</td>
<td>Static text</td>
</tr>
<tr>
<td>GX_IDS_CTRL_PUSHTBTN</td>
<td>Push button</td>
</tr>
<tr>
<td>GX_IDS_CTRL_RADIOBTN</td>
<td>Radio button</td>
</tr>
<tr>
<td>GX_IDS_CTRL_RADIOBTN3D</td>
<td>Radio buttons with 3D effect</td>
</tr>
<tr>
<td>GX_IDS_CTRL_CHECKBOX</td>
<td>Check box</td>
</tr>
<tr>
<td>GX_IDS_CTRL_CHECKBOX3D</td>
<td>Check box with 3D effect</td>
</tr>
<tr>
<td>GX_IDS_CTRL_COMBOBOX</td>
<td>Regular combo box; allows user to input any text</td>
</tr>
<tr>
<td>GX_IDS_CTRL_TEXTFIT</td>
<td>Regular combo box with text fit; limits user input to entries in the choice list</td>
</tr>
<tr>
<td>GX_IDS_CTRL_ZEROBASED</td>
<td>Combo box with zero-based indices; stores and displays the index value</td>
</tr>
<tr>
<td>GX_IDS_CTRL_ONEBASED</td>
<td>Combo box with one-based indices; stores and displays the index value</td>
</tr>
<tr>
<td>GX_IDS_CTRL_ONEBASED_EX</td>
<td>Combo box with one-based indices; stores the index value but displays the associated choice list text</td>
</tr>
<tr>
<td>GX_IDS_CTRL_ZEROBASED_EX</td>
<td>Combo box with zero-based indices; stores the index value but displays the associated choice list text</td>
</tr>
<tr>
<td>GX_IDS_CTRL_CBS_DROPDOWN</td>
<td>MFC CComboBox with the CBS_-DROPDOWN style</td>
</tr>
<tr>
<td>GX_IDS_CTRL_CBS_DROPDOWNLIST</td>
<td>MFC CComboBox with the CBS_-DROPDOWNLIST style</td>
</tr>
</tbody>
</table>
Table 2 – Predefined control IDs (Continued)

<table>
<thead>
<tr>
<th>Control ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GX_ID_CTRL_TABBED_COMBOBOX</td>
<td>Tabbed combo box</td>
</tr>
<tr>
<td>GX_ID_CTRL_CBS_TABBED_DROPDOWN</td>
<td>Tabbed combo box based on an MFC CComboBox with the CBS_DROPDOWN style</td>
</tr>
<tr>
<td>GX_ID_CTRL_CBS_TABBED_DROPDOWNLIST</td>
<td>Tabbed combo box based on an MFC CComboBox with the CBS_DROPDOWNLIST style</td>
</tr>
<tr>
<td>GX_ID_CTRL_CHECKLIST_COMBOBOX</td>
<td>Drop down checklist combo box</td>
</tr>
<tr>
<td>GX_ID_CTRL_LISTBOX</td>
<td>List box</td>
</tr>
<tr>
<td>GX_ID_CTRL_HEADER</td>
<td>Header</td>
</tr>
<tr>
<td>GX_ID_CTRL_BROWSEARROWHEADER</td>
<td>Row header for browser grids; displays the arrow or pencil icon as appropriate</td>
</tr>
<tr>
<td>GX_ID_CTRL_PROGRESS</td>
<td>Progress bar</td>
</tr>
<tr>
<td>GX_ID_CTRL_PASSWORD</td>
<td>Password edit</td>
</tr>
<tr>
<td>GX_ID_CTRL_DATETIME</td>
<td>Date time control with a pop-up calendar</td>
</tr>
<tr>
<td>GX_ID_CTRL_DATETIMENOCAL</td>
<td>Date time control without a pop-up calendar</td>
</tr>
<tr>
<td>GX_ID_CTRL_CURRENCY</td>
<td>Formatted currency control</td>
</tr>
</tbody>
</table>

Here is an example of using these settings:

```c
// Assign a combo box to a cell
SetStyleRange(CGXRange(3,3),
    CGXStyle()
    .SetControl(GX_ID_CTRL_COMBOBOX)
    .SetChoiceList("one\ntwo\nthree\n")
);
```
2.5 Browser Architecture

The Objective Grid browser classes let you easily browse any external data source by overriding virtual methods. Objective Grid already provides special implementations of the browser classes for displaying ODBC `CRecordset` and ADO results.

The key principle for browsing external data sources is that the grid object maintains one current record with the current cell. When the user edits data in the current record, all changes are stored in an internal buffer. When the user moves the current cell to another record, changes for the current record are written to the data source before the current cell is positioned to a new record. When the user clicks on the last row, a new record will be added to the data source (if the data source supports appending records).

The main features of Objective Grid for browsing data sources are:

- Row headers show the state of the current record. An arrow indicates the current record as unmodified, while a pencil indicates that the current record has been modified.
- Provides support for adding rows to the data source. When the user enters data in the last row (marked with an asterisk in the row header), a new record will be appended.
- Provides Undo support for changes and the ability to cancel all pending changes in the current record.
- Provides improved abstraction for loading cell data from the data source and storing all changed cell data into the data source when the user has moved to a new record.
- Allows the programmer to validate user input at the record level. This means the user can input data into the record and when the current cell moves to a new record, all changes for the previous record will be validated.
- Synchronizes data changes and current cell movement between different views on the same data.
- Provides support for rearranging columns and rows. This lets you add support for dragging columns and sorting rows.
- Provides support for maintaining a high-water mark for the last seen record. `CGXAdoGrid` and `CGXODBCGrid` use this high-water mark to determine the record count as the user navigates through the records.

2.5.1 CGXBrowserGrid - Interface

`CGXBrowserGrid` abstracts the basic functionality for browsing databases. Two important groups of methods of the `CGXBrowserGrid` class allow you to:

- Modify cells in the grid independent of the specific data source
- Bind the grid to the external data source with a set of overrides
2.5.1.1 Current Record

The following member functions allow you to modify the cells in the grid, independent of the data source you are browsing:

**AddNew()**

Prepares the grid for adding a new record. The grid appends a new line in the grid. Call **Update()** to complete the addition.

**CancelRecord()**

Cancels all pending changes in the current record and resets the edit- or append-mode. If an **AddNew()** is in progress, the pending row is removed from the grid.

**Edit()**

Prepares for changes to the current record. Call **Update()** to complete the edit.

**UndoRecord()**

Cancels all changes for the current record (by calling **CancelRecord()**) and empties the Undo and Redo list.

**Update()**

Completes an **AddNew()** or **Edit()** operation by saving the changed values to the data source.

2.5.1.2 Data Source Binding

The following member functions should be overridden to bind the browser grid to your external data source.

**CanAppend()**

Called to determine if the user should be able to append rows to the data source.

**CanUpdate()**

Called to determine if the user should be able to change data in the data source.

**DeleteRows()**

Called to delete rows from the data source.

**MoveTo()**

Called to position the recordset to the given row. You should return **FALSE** if the move failed.

**OnAddedNewRecord()**

Called from **Update()** when a new record has been added to the data source.
OnFlushCellValue()

Called from within CGXBrowserGrid::OnFlushRecord() for all values changed in the current record buffer. You should override this method to store changed values in your data source.

OnFlushRecord()

Writes “dirty” fields (those with unsaved changes) back to the data source and prepares the data source at record level before changes are written to it with OnFlushCellValue().

OnGetRecordCount()

Called to determine the number of records in the data source, if the last record has been accessed. If the record count is unknown, the method should return LONG_MAX.

OnLoadCellStyle()

Called to load a data value from the data source or apply additional formatting at run time. This method provides a better abstraction for loading data from data sources at the record level. OnLoadCellStyle() is called from GetStyleRowCol() when needed.

Requery()

Called to requery the data source. The default implementation resets the high-water mark for the last seen record.

2.5.2 ODBC Browser

As already mentioned, the MFC ODBC classes do not directly support the ability to determine database schema information at run time. Objective Grid provides support to allow the end user to specify and view the results of an SQL query at run time (where schema is unknown). Objective Grid also supports the MFC paradigm where the developer uses ClassWizard to create recordsets with a known schema.

CGXODBCGrid is a special derivative of the CGXBrowserGrid class for browsing ODBC data sources. CGXRecordWnd and CGXRecordView inherit from CGXODBCGrid and represent the ODBC browser as a view or dialog control, respectively. CGXODBCGrid is implemented only through overriding the methods named in the previous “Data Source Binding” section of the CGXBrowserGrid interface.

When initializing your grid, you should store a pointer to your recordset object inside of your grid. The recordset will be of type CRecordset (or a class derived from it).

Objective Grid’s CGXDynamicRecordset class derives from CRecordset and lets you easily construct a recordset object that can be browsed in the grid.
CGXDynamicRecordset m_QuerySet;
m_QuerySet.SetSqlQuery("SELECT * FROM STUDENT");

// Overriden OnInitialUpdate method
void CMyGridView::OnInitialUpdate()
{
    SetRecordset(&m_QuerySet);
    CGXRecordView::OnInitialUpdate();
}

See the Objective Grid Class Reference and the GxQuery sample for further information on ODBC classes. Also check out the tutorial for using the ODBC classes.

2.5.3 ADO Browser

CGXAdoGrid is a special derivative of the CGXBrowserGrid class for browsing any database through ADO. CGXAdoRecordWnd and CGXAdoRecordView inherit from CGXAdoGrid and represent the ADO browser as a view or dialog control. CGXAdoGrid is implemented only through overriding the methods named in the previous “Data Source Binding” section of the CGXBrowserGrid interface.

See the Objective Grid Class Reference and the AdoQuery sample for further information about ADO classes.
2.6 Formula Engine

Because of the complex interdependencies among cells, Objective Grid treats the formula engine as a separate entity. You might consider Objective Grid as a front end to this engine. It loads values from the engine when cells need to be drawn and that stores values or formulas back into the engine whenever cells are changed. This is very similar to the virtual approach, in which you override certain grid methods, like `GetStyleRowCol()` and `StoreStyleRowCol()`, to browse and modify data of any external data source. Feeding the engine with data and loading data from the engine allows the engine to easily track any cell dependencies. Whenever a cell is changed, the formula engine will update all cells with formulas that reference the modified cell. The grid object responsible for drawing the cells will query the engine to determine which cells need to be redrawn. Cells that need to be redrawn are marked with a special flag so that unnecessary drawing/flickering does not occur when a cell is changed. Only cells with changes in value are redrawn.

The formula engine is accessed through the `CGXFormulaSheet` class, which is derived from `CGXData` and replaces the `CGXData` object that holds all the cell data. `CGXFormulaSheet` has special overrides for `GetStyleRowCol()` and `StoreStyleRowCol()` so that whenever cells are changed in the grid the actual change is directly stored in the formula engine. Other operations like moving, removing, and inserting cells are forwarded to the engine by overrides in `CGXFormulaSheet`.

As the programmer, you should only call functions and methods provided by the `CGXFormulaSheet` class when you interact with the engine from your grid object. This will keep your code more readable and more understandable, as you will be interacting with a class object. The `CGXFormulaSheet` class is a “wrapper” class to the engine. The code for the engine itself should be considered separate. Objective Grid ships with full source code for the engine but should be used for reference only. There is rarely a reason to access any of the engine functions directly with the exception of adding new worksheet functions. The engine itself is based on legacy C code and is not documented. Only the `CGXFormulaSheet` class is documented in the Objective Grid Class Reference.

The engine is extensible. Although you can’t change the grammar of the formula parser, you will be able to add your own worksheet functions. When you add your own worksheet functions, you may have to deal with the engine directly and use the low-level engine functions directly instead of Objective Grid functions. If you need to add your own worksheet functions, use existing worksheet functions as template.

2.6.1 Adding Formula Support to Your Application

Formula support is enabled by calling the function `EnableFormulaEngine()` at the beginning of your `OnInitialUpdate()` routine or before you call `CGXGridWnd::Initialize()`. The `EnableFormulaEngine()` call will force Objective Grid to replace the default `CGXData` object with the `CGXFormulaSheet` object. This establishes the connection to the formula engine.

Furthermore, you might want to enable all the built-in worksheet functions that come with the formula engine. To do this, call `GXEnableWorksheetFunctions()` from your `InitInstance()` method.
If you link your application statically with the Objective Grid libraries, the engine code will not be linked into your application unless you call EnableFormulaEngine(). This avoids unnecessary overhead induced by the formula engine when you don’t need formula support in your application.

### 2.6.2 Initializing the Grid With Data and Formulas

After you have called EnableFormulaEngine(), you can fill the grid with SetValueRange(), SetStyleRange(), and other grid functions. In order to store formulas, numeric, or date values to cells you should call SetExpressionRowCol().

SetExpressionRowCol() differs from SetValueRange() and SetStyleRange() as it will parse the string value. If the string value is a formula, a formula is stored. If the string value is a string, a string value is stored, and if it is a number, a number is stored.

Date values are of special interest. If you pass a date or time as a string of numbers (e.g. “01/01/99”) to SetExpressionRowCol(), Objective Grid will interpret this string, store a date value in the cell and set the cell format to date. Cell formats are specified through CGXStyle::SetFormat() and the precision is specified via CGXStyle::SetPlaces().

Please note that the formula engine distinguishes between numeric and string values. If you pass a numeric value as a string via SetValueRange(), as in

```
SetValueRange (... , "100");
```

the number will be stored and treated as a string. If you pass this string via SetExpressionRowCol(), the string is parsed and recognized and stored as a number. If you pass the numeric value directly as a number to SetValueRange(), as in

```
SetValueRange (... , 100.00);
```

the value will also be stored as number.

### 2.6.3 Changing Cell Data and Formulas at Run Time

The user has the ability to change data and formulas at run time. Whenever the user edits a cell, the data will be stored in the grid by a call to SetExpressionRowCol(). This gives the user the ability to enter formulas in the current cell.

This following example shows how you can enable formula support in your grid view:

```cpp
void CGridSampleView::OnInitialUpdate()
{
    EnableFormulaEngine();

    BOOL bNew = ConnectParam();

    // Create all objects and link them to the grid
    CMyGridView::OnInitialUpdate();

    // ... and now you can execute commands on the grid

    if (bNew)
    {
        EnableHints(FALSE);
    }
```
// Lock any drawing
BOOL bOld = LockUpdate();

// initialize the grid data

// disable Undo mechanism for the following commands
GetParam()->EnableUndo(FALSE);

// no iteration for circular references
GetSheetContext()->SetIterationLimit(0);

// automatic/manual recalculation
GetSheetContext()->SetRecalcMode(GX_RECALC_AUTO);

// reevaluate cells on demand
GetSheetContext()->SetRecalcMethod(GX_RECALC_AS_NEEDED);

// turn off constraint checks
GetSheetContext()->SetConstraintCheck(FALSE);

// Initialize grid with 1000 rows and 40 columns
SetRowCount(1000);
SetColCount(40);

// Insert an array with numeric data
ROWCOL nRow, nCol;
double d = 0.0;
for (nRow = 7; nRow <= 12; nRow++)
{
    d *= 2.0;
    for (nCol = 1; nCol <= 4; nCol++)
    {
        d += 1.0;
        SetStyleRange(CGXRange(nRow, nCol),
                      CGXStyle()
                      .SetValue(d)
                      .SetHorizontalAlignment(DT_RIGHT));
    }
}

// Some string data
SetValueRange(CGXRange(7, 6), _T("Hello "));
SetValueRange(CGXRange(7, 7), _T("world "));
SetValueRange(CGXRange(8, 6), _T("Stingray "));
SetValueRange(CGXRange(8, 7), _T("Software "));
SetValueRange(CGXRange(9, 6), _T("Objective "));
SetValueRange(CGXRange(9, 7), _T("Grid "));

nRow++;
nRow++;
SetStyleRange(CGXRange(nRow, 1),
              CGXStyle()
              .SetValue(_T("String Functions"))
              .SetEnabled(FALSE)
              .SetFont(CGXFont().SetBold(TRUE)));

nRow++;
<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>STRCAT</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>=STRCAT(F7, G7)</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>LENGTH</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>=LENGTH(F7)</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>FIND</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>=FIND(&quot;l&quot;, F7, 0)</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>MID</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>=MID(F9&amp;G9, 3, 5)</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>LOWER</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>=LOWER(F9&amp;G9)</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>REPEAT</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
<td>=REPEAT(=&quot;&quot;, 10)</td>
</tr>
</tbody>
</table>

See Chapter 14, “The Formula Engine,” for more detailed information about the formula engine and a reference of worksheet functions.
2.7 Run-Time Initialization

The overridable CGXGridCore::OnGridInitialUpdate() method is the core of the initialization process of a grid view or grid control. You should override this method if you have any one-time initialization such as registering controls, setting the number of rows and columns, or initializing properties or styles.

If you use Objective Grid as a dialog control or an embedded window, you are responsible for calling the CGXGridWnd::Initialize() method yourself. This can be done after creating the window with two-phase construction (like CWnd), or in the OnInitDialog() method of your dialog class. Initialize() will call OnGridInitialUpdate().

If you use the grid as a view, OnInitialUpdate() will be called by the framework after the view is first attached to the document, but before the view is initially displayed. In a workbook, the OnInitialUpdate() method is called for each worksheet, even if it is not visible. The OnInitialUpdate() method will call OnGridInitialUpdate().

The grid component distinguishes between shared and unique attributes. CGXGridCore maintains attributes that are unique to each grid object. Shared attributes such as row heights and cell contents have been moved to an extra class, the CGXGridParam class. Each grid object owns a pointer to a parameter object which is an instance of the CGXGridParam class. The parameter object maintains several other objects that can be shared by associated views of one document. The properties object with profile settings, the data object with the cells map, the print device object with the printer settings, the range lists with selected and covered cells, the Undo/Redo lists, and the column widths and row heights are all maintained by CGXGridParam. The parameter object can be maintained by the document or by the grid object itself.

A main task of OnInitialUpdate() is to attach a parameter object to the grid object. If you do override OnInitialUpdate(), you can create a parameter object of your own and attach it to the grid using CGXGridCore::SetParam(). However, if you do not want to maintain the object yourself, the base class version of OnInitialUpdate() calls OnGridInitialUpdate(), which then creates all necessary objects on the heap. OnGridInitialUpdate() creates only those objects which you did not previously initialize.

Here is an example of allocating the parameter object from the heap when using a stand-alone grid view or grid window.

CGXSampleView::OnInitialUpdate()
{
    // When called no reference to a parameter object
    // is stored in the grid
    ASSERT(GetParam() == NULL);

    // Store a reference to a parameter object
    SetParam(new CGXGridParam);

    // Now, you could attach the properties,
    // the stylesmap and the data object to
    // the parameter object
    ...
}
and call the base class OnInitialUpdate method
// (this will call OnGridInitialUpdate)
CGXGridView::OnInitialUpdate();

// Now, OnInitialUpdate has stored references
// to all other objects
ASSERT(GetParam()->GetData());            // Data object
ASSERT(GetParam()->GetProperties());      // Properties object
ASSERT(GetParam()->GetStylesMap());       // StylesMap object
}

Another common requirement is to use the grid view attached to a document. In this case, all data should be maintained by the document. You can embed the parameter object in the document.

// document class
class CMyDocument
{
  ...

  // Attributes
  CGXGridParam* m_pParam; // parameter object
  ...
};

CMyDocument::CMyDocument()
{
  m_pParam = NULL;
}

void CMyDocument::DeleteContents()
{
  delete m_pParam;
  m_pParam = NULL;
}

// view class
CGXSampleView::OnInitialUpdate()
{
  BOOL bFirstView = FALSE;
  if (GetDocument()->m_pParam == NULL)
  {
    // bFirstView = TRUE indicates that this
    // is the first view connected to the
    // document and therefore the data need
    // to be initialized.
    bFirstView = TRUE;

    // construct parameter object
    GetDocument()->m_pParam = new CGXGridParam;
  }
  else
  {
    // bFirstView = FALSE indicates that this is
    // only another view connected to the document
    // No data need to be initialized. They are
    // all available in the document already.
// pass the pointer to the grid view

SetParam(GetDocument()->m_pParam, FALSE);
    // ^-- indicates that document
    // is responsible for deleting
    // the object.

// Call base class version of this method
// (This will call OnGridInitialUpdate)
CGXGridView::OnInitialUpdate();

// check if data need to be initialized
if (bFirstView)
{
    // Lock any drawing
    BOOL bOldLock = LockUpdate();

    // turn off creation of Undo-information
    GetParam()->EnableUndo(FALSE);

    // initialize the grid
    SetRowCount(...);
    ...

    // reenable Undo mechanism
    GetParam()->EnableUndo(TRUE);

    // Unlock drawing
    LockUpdate(bOldLock);
}

// Just to be sure that everything is redrawn
Invalidate();

// Enable Objective Grid internal update hint mechanism

// You should put this line as last command into
// OnInitialUpdate, because as long as EnableHints is not
// called, the modified flag of the document will not be
// changed.
EnableHints();
}
2.8 Choosing Between Grid Types

Objective Grid differentiates between several types of grids:

- **Regular Grid** — You fill the grid with data at startup. The user can edit cells. Changes will be stored in the grid.

- **Virtual Grid** — The data to be displayed in the grid are maintained in a self-written data structure or in an external file. The grid does not hold any cells in memory. You don't fill the grid at startup. Instead, you override the `GetStyleRowCol()` function to supply data on demand. Before it draws the cells on the screen, Objective Grid calls `GetStyleRowCol()` only for the currently visible cells. If you want the user to be able to edit cells, you need to override `StoreStyleRowCol()` and store the changes back into your custom data source. Objective Grid calls `StoreStyleRowCol()` whenever a cell is changed.

- **Formula Grid** — You enable the formula engine and fill the grid with data at startup. Cells can contain formulas that reference other cells. When the user edits cells, changes will be stored in the formula engine. If a cell that is referenced by a formula is changed, the depending cell is automatically updated.

- **Browser Grid** — A Browser Grid is somewhat similar to a Virtual Grid. The main difference is that the Browser Grid has been designed with certain characteristics of record-based data sources in mind. Take an ODBC recordset, for example. You cannot directly access data in any record. You have to loop through the records in order to access the desired record. Then you can read data from this current record and make changes. When the user starts editing a record, changes should only be written back to the recordset when the user navigates to a new record. Therefore, Objective Grid holds changes to the current record in a buffer and flushes this buffer only when the user moves to a new record. Browser Grid also lets the user easily append rows. The appearance of Browser Grid is very similar to MS Access query views.

Objective Grid provides Browser Grid implementations for ODBC and ADO. Furthermore, you can easily add your own implementation of Browser Grid for any other record-based data source. For example, you could write your own Browser Grid implementation for SQL Server API, Sybase SQL Server, SourcePro DB from Rogue Wave, or CodeBase from Sequiter.
Here are some guidelines that should help you decide what kind of grid you want to use:

- **Do you need formula support?**
  If so, you should use the Formula Grid. Check out the Formula sample for implementation details.

- **Does your grid have a large number of rows?**
  If so, consider using a Virtual Grid.

- **Do you maintain data in an external data source or custom data structure?**
  If so, there are more issues to consider:
    - **Do you want to browse an ODBC or ADO recordset?**
      If so, you should use the ODBC or ADO Browser Grid implementation. Check out the GXQuery and AdoQuery samples for details on using these types of grids.
    - **Is the external data source record based? (Should changes in a cell be immediately written back to the data source or should they be held in a buffer and flushed only when the user navigates to another record?)**
      If the external data source is record based, then you should derive from CGXBrowserGrid.
    - **Do you want the grid to directly operate on your data?**
      Use Virtual Grid. This has the advantage that you don’t have to copy all the data at startup into the grid. The GridApp sample provides a sample for this type of grid. Check out the files browsevw.cpp and browsevw.h.
    - **Do you want to copy the data into your grid?**
      Use Regular Grid. At startup you fill the grid with your data. The 1stGrid tutorial demonstrates this kind of grid.
3.1 Introduction

This section presents a technical discussion of the major features of Objective Grid. Objective Grid provides Microsoft Foundation Class (MFC) developers with a feature-rich grid control implemented as a complete MFC class extension library.

Like MFC, MFC extensions are a group of C++ classes. They are built on and extend MFC classes. All Stingray MFC extension products come with complete Visual C++ compatible source code and can be built either as static libraries or as extension DLLs.

Since C++ was built with object oriented components in mind, MFC extensions are a very natural way to extend MFC. In fact, as you read the description of the Objective Grid classes, you will notice that we describe our extension classes using the familiar C++ class hierarchy charts. Our MFC extension classes fit so closely with MFC that you probably won’t even notice that you are using a different product.

Unlike other grid controls, Objective Grid has been built from the ground up with MFC compatibility as a major goal. The Objective Grid classes fit seamlessly with MFC and in many cases inherit from existing MFC classes such as CView or CWnd.
3.2 Class Summary

This section provides an overview of the following groups of MFC extension classes.

- **Drawing Classes** - The Objective Grid drawing classes perform all of the drawing and updating of the grid cells. Both `CWnd` and `CView` derived grid classes are available for maximum flexibility.

- **Control Classes** - Objective Grid cells can be a variety of control types. An MFC control extension class is provided for each type of control. An abstract base class is also included that helps the developer create new controls to supplement the existing controls.

- **Style Classes** - A style refers to the attributes of a cell or group of cells. The Objective Grid style classes manage these attributes and provide a pre-built dialog that enables end users to modify them.

- **Formula Engine** - The Objective Grid formula engine implements all of the Objective Grid formula support. All formula engine code is separate from the base Objective Grid code and only gets linked when you specify formula support for your grid.

- **Browser Classes** - The Objective Grid Browser classes let you easily browse any external data sources by overriding some virtual methods. Objective Grid already provides special implementations of the browser classes for displaying ODBC `CRecordset` and ADO results.

- **Data Structures** - Objective Grid maintains grid properties and attributes in separate classes, so that several grid views can share the same data. Objective Grid also provides pre-built dialogs to modify these properties.

- **Utility Classes** - Objective Grid uses a variety of MFC extension classes for internal utility classes. These classes may be helpful in other parts of your applications. Examples of the utility classes are a long operation class that lets you monitor and cancel long operations, a tabbed window, and an Undo/Redo architecture.

- **Plug-In Component Classes** - Plug-in components allow you to implement reusable window functionality that can be attached to various window classes. IntelliMouse support is an example of a plug-in component.
3.3 Drawing Classes

Figure 5 – Drawing class hierarchy

This class hierarchy diagram may be a little different from most that you’ve seen. That’s because Objective Grid uses C++ multiple inheritance and MFC does not. Objective Grid uses multiple inheritance in order to support both a CWnd and a CView derivative via the CGXGridCore class. The sections below describe each class in detail and explain why Objective Grid uses this approach.

3.3.1 CGXGridCore

The CGXGridCore class encapsulates all of the drawing logic for the grid and its components. CGXGridCore responsibilities include:

- Drawing and scrolling the grid.
- Executing commands such as formatting cells, inserting rows and columns, and moving rows and columns.
- Handling Undo and Redo commands.
- Managing the position of the current cell.
- Interpreting user interactions such as mouse clicks and keyboard strokes.
- Freezing rows and columns.
- Inverting selected cells.
The **CGXGridView** and **CGXGridWnd** classes inherit from both an MFC class and **CGXGridCore**. This use of multiple inheritance is necessary to avoid duplication of the drawing logic in both a **CWnd** and a **CView** derivative. By placing all of the drawing logic in **CGXGridCore**, Objective Grid centralizes the drawing logic in one class.

### 3.3.2 **CGXView**

The **CGXView** class serves as base class for a variety of views. It adds support for storing printer settings in a view or document and provides interfaces that make it easier to place the grid in a tabbed window.

### 3.3.3 **CGXGridView**

This class presents the grid component as a view. **CGXGridView** supports **CView**-like features for the grid including splitter window support, printing, and print-preview. **CGXGridView** is designed as a base class and is typically not instantiated.

### 3.3.4 **CGXGridWnd**

**CGXGridWnd** gives the grid a **CWnd** interface so that it can be used like a control. In other words, **CGXGridWnd** can be placed in a dialog and manipulated with the Visual C++ dialog editor. Like **CWnd**, **CGXGridWnd** is rarely instantiated. Instead, the developer derives from **CGXGridWnd**, overrides the virtual functions to obtain the desired behavior and then instantiates the derivative.

### 3.3.5 **CGXGridHandleView**

**CGXGridHandleView** lets you dynamically attach a **CGXGridCore**-derived class to a **CView** derivative. The advantage of this technique is that **CGXGridHandleView** can be bound with any **CGXGridCore** derivative at run time. It is not statically bound to **CGXGridCore** as **CGXGridView** is. **CGXGridHandleView** delegates all mouse actions and user input to the **CGXGridCore** object.

### 3.3.6 **CGXGridHandleWnd**

**CGXGridHandleWnd** lets you dynamically attach a **CGXGridCore**-derived class to a **CWnd** derivative so that it can be used like a control.

### 3.3.7 **CGXGridHint**

Objective Grid uses **CGXGridHint** internally to perform optimized updating of all views.
3.4 Control Classes

All MFC control classes are derived from \textit{CWnd}, but this base class does not provide some of the control interfaces needed by the grid object such as mouse operations, find and replace, etc. Objective Grid uses the same multiple inheritance technique to create control classes with an appropriate interface.

Controls can be combined with children. For example, a spin control has two arrow children: an up-arrow child and a down-arrow child. Objective Grid provides several control child classes, mainly to be used as small buttons in a cell.

You need to register new controls in the grid object (with CGXGridCore::RegisterControl) before using them as cell controls. See Section 14.7, “Quick-Reference Guide to Built-in Functions,” for more information.

Figure 6 shows the class hierarchy for the control classes.
Figure 6 – Control classes hierarchy

CGXControl
  ├ CGXCheckBox
  │   ├ CGXCheckBoxEx
  │   │   └ CGXButton
  │   │       └ CGXBitmapButtonChild
  │   └ CGXPushButton
  │       └ CGXRadioButton
  │       └ CGXPushButtonEx
  │          └ CGXListBox
  │                 └ CGXStatic
  │                         └ CGXEditControl
  │                               └ CGXChild
  │                                   └ CGXButton
  │                                       └ CGXBitmapButtonChild
  │                                           └ CObject
  │                                               └ CCmdTarget
  │                                                   └ CWnd
  │                                                       └ CListBox
  │                                                           └ CEdition
  │                                                               └ CGXBDropEdit
  │                                                                  └ CGXBCurrencyEdit
  │                                                                      └ CRichEditCtrl
  │                                                                             └ CGXProgressCtrl
  │                                                                                   └ CGXRichEditCtrl
  │                                                                                       └ CGXBitmapButton
  │                                                                                               └ CGXWndWrapper
  │                                                                                                               └ CGXStatic
  │                                                                                                                  └ CGXDateTimeCtrl
  │                                                                                                                      └ CGXBDropEdit
  │                                                                                                                  └ CGXStatic
  │                                                                                                                      └ CGXCurrencyEdit
  │                                                                                                                      └ CGXDateTimeCtrl
  │                                                                                                                   └ CGXDateTimeCtrl
3.4.1 CGXControl

The CGXControl class is an abstract base class. CGXControl establishes a default grid-to-control interface that derived control classes must implement. By deriving from CGXControl, new controls can be implemented simply by overriding a handful of methods. New controls can also be derived from CGXControl and an existing MFC control class to implement a standard MFC control derivative. The resulting MFC control derivative derives its message map from the MFC inheritance and its grid properties from the CGXControl inheritance. For example, classes CGXEditControl, CGXComboBoxWnd and CGXListBox are MFC control derivatives that use this approach.

Developers can easily modify the behavior of the CGXControl derived classes. For example, you could create an owner-drawn combo box of colors or a drop-down calendar.

3.4.2 CGXBitmapButton

The CGXBitmapButton class implements a bitmap button control that can be used in grid cells.

Figure 7 – CGXBitmapButton in action

3.4.3 CGXCheckBox

The CGXCheckBox class implements a check box control that can be used in grid cells. This class realizes the GX_IDS_CTRL_CHECKBOX and GX_IDS_CTRL_CHECKBOX3D control IDs.

Figure 8 – CGXCheckBox in action

Figure 9 – 3D CGXCheckBox in action

3.4.4 CGXCheckBoxEx

The CGXCheckBoxEx class implements a check box control that can be used in grid cells. It also supports an active state for the check box control. This class realizes the GX_IDS_CTRL_CHECKBOX_EX and GX_IDS_CTRL_CHECKBOX3D_EX control IDs.

Figure 10 – CGXCheckBoxEx in action

Figure 11 – 3D CGXCheckBoxEx in action
3.4.5 CGXCheckListComboBox

The CGXCheckListComboBox class implements a drop down list that allows the user make multiple selections. This class realizes the GX_IDS_CTRL_CHECKLIST_COMBOBOX control ID.

Figure 12 – CGXCheckListComboBox in action

<table>
<thead>
<tr>
<th>Click Me</th>
<th>One</th>
<th>Two</th>
<th>Three</th>
<th>Four</th>
<th>Five</th>
<th>Six</th>
<th>Seven</th>
</tr>
</thead>
</table>

3.4.6 CGXComboBox

The CGXComboBox class implements a text input control that can be used to display and edit text in cells. The user can modify this text in place or select an item from a drop down list box. CGXComboBox realizes the GX_IDS_CTRL_COMBOBOX, GX_IDS_CTRL_TEXTFIT, GX_IDS_CTRL_ZEROBASED, GX_IDS_CTRL_ONEBASED, GX_IDS_CTRL_ZEROBASED_EX, and GX_IDS_CTRL_ONEBASED_EX control IDs.

Figure 13 – CGXComboBox in action

| two | one | two | three | four |

3.4.7 CGXComboBoxWnd

The CGXComboBoxWnd class implements a combo box control that can be used to display and select items from a list. The user can select items in the combo box. The difference between CGXComboBoxWnd and CGXComboBox is that CGXComboBoxWnd is derived from the Windows standard CComboBox control, while CGXComboBox is a CGXEditControl with a small button at the right-hand side of the input area. CGXComboBoxWnd realizes the GX_IDS_CTRL_CBS_DROPDOWN and GX_IDS_CTRL_CBS_DROPDOWNLIST control IDs.
3.4.8 CGXCurrencyEdit

The CGXCurrencyEdit class implements a currency control cell type that can be used to display and enter currency values. CGXCurrencyEdit realizes the GX_IDS_CTRL_CURRENCY control ID.

3.4.9 CGXDateTimeCtrl

The CGXDateTimeCtrl class implements a date control cell type that can be used to display and enter dates. CGXDateTimeCtrl realizes the GX_IDS_CTRL_DATETIME and GX_IDS_CTRL_DATETIMENOCAL control IDs.
3.4.10 CGXEditControl

The CGXEditControl class implements a text input control that can be used to display and edit text in cells. CGXEditControl realizes the GX_IDS_CTRL_EDIT control ID.

3.4.11 CGXHeader

The CGXHeader class implements a static text control which is used to display column and row headers. Headers typically have a 3d-button-look and a special feature is that they are drawn in a pressed state if the current cell is at the same row or column. CGXHeader realizes the GX_IDS_CTRL_HEADER control ID.

3.4.12 CGXHotSpotEdit

The CGXHotSpotEdit class implements a text input control that can be used to display and edit text in cells. The user can modify this text in place. The difference between CGXHotSpotEdit and CGXEditControl is that CGXHotSpotEdit displays a small button at the right side of the input area. CGXHotSpotEdit realizes the GX_IDS_CTRL_HOTSPOT control ID.
3.4.13 CGXListBox

The CGXListBox class implements a list box control that can be used to display and select items from a list. The user can select items in the list box. CGXListBox realizes the GX_IDS_CTRL_LISTBOX control ID.

Figure 20 – CGXListBox in action

3.4.14 CGXMaskControl

CGXMaskControl lets you add formatted input capabilities for cells. CGXMaskControl realizes the GX_IDS_CTRL_MASKEDIT control ID.

Figure 21 – CGXMaskControl in action

3.4.15 CGXPasswordControl

The CGXPasswordControl class implements a text input control for passwords. When the user modifies the text, a password character (e.g. an asterisk, *) will be shown for each entered character. CGXPasswordControl realizes the GX_IDS_CTRL_PASSWORD control ID.

Figure 22 – CGXPasswordControl in action

3.4.16 CGXProgressCtrl

The CGXProgressCtrl class lets you display a progress bar in cells. It also lets you display text (e.g. percentage value) in the cell. CGXProgressCtrl realizes the GX_IDS_CTRL_PROGRESS control ID.

Figure 23 – CGXProgressCtrl in action

3.4.17 CGXPushbutton

The CGXPushbutton class implements a pushbutton control that can be used in grid cells. CGXPushButton realizes the GX_IDS_CTRL_PUSHBTN control ID.

Figure 24 – CGXPushbutton in action
3.4.18 CGXRadioButton

The CGXRadioButton class implements a control with a group of radio buttons that can be used in grid cells. CGXRadioButton realizes the GX_IDS_CTRL_RADIOBTN and GX_IDS_CTRL_RADIOBTN3D control IDs.

Figure 25 – CGXRadioButton in action

- Radio 1
- Radio 2
- Radio 3

Figure 26 – 3D CGXRadioButton in action

- one
- two
- three

3.4.19 CGXRadioButtonEx

The CGXRadioButtonEx class implements a control with a group of radio buttons that can be used in grid cells. It supports an active state for the control as well as horizontal or vertical orientation for the radio buttons. CGXRadioButtonEx realizes the GX_IDS_CTRL_RADIOBTNEX and GX_IDS_CTRL_RADIOBTN3DEX control IDs.

Figure 27 – CGXRadioButtonEx in action

- Radio 1
- Radio 2
- Radio 3

Figure 28 – 3D CGXRadioButtonEx in action

- one
- two
- three

This control has the ability to align vertically or horizontally via the GX_IDS_UA_RADIOBUTTON_ALIGN user attribute.

In version 7.0 and earlier, text was always located to the right of the radio button. An option in more recent versions allows you to place the text to either the left or right of the radio button.

The GX_IDS_UA_RADIOBUTTONTEXTALIGN user attribute is similar to GX_IDS_UA_RADIOBUTTON_ALIGN, but indicates whether to place the text to the left or right of the button. To get the look you had in earlier versions (text on the right side), try the following code:

SetStyleRange(CGXRange(1,5),
    CGXStyle()
        .SetControl(GX_IDS_CTRL_RADIOBTN3DEX)
        .SetChoiceList("choice one\nchoice two\nchoice three\n")
        .SetUserAttribute(GX_IDS_UA_RADIOBUTTON_ALIGN, _T("1"))
    //test to the right
3.4.20 CGXRichEditCtrl

The **CGXRichEditCtrl** lets you display and edit Rich Text in cells. Rich Text allows you to format individual characters or paragraphs with different fonts, colors or horizontal alignment. **CGXRichEditCtrl** realizes the **GX_IDS_CTRL_RICHEDIT** control ID.

![Image of CGXRichEditCtrl in action]

The quick brown fox jumped over the lazy dog.

3.4.21 CGXSpinEdit

The **CGXSpinEdit** class implements a text input control with spin buttons. The user can modify this text in place, increase or decrease the value by clicking on the spin buttons. **CGXSpinEdit** realizes the **GX_IDS_CTRL_SPINEDIT** control ID.

![Image of CGXSpinEdit in action]

3.4.22 CGXStatic

The **CGXStatic** class implements a static text control that can be used to display text in cells. The user cannot modify this text in place. **CGXStatic** realizes the **GX_IDS_CTRL_STATIC** control ID.

![Image of CGXStatic in action]

3.4.23 CGXTabbedComboBox

The **CGXTabbedComboBox** class implements a tabbed combo box based on the existing **CGXComboBox** implementation. **CGXTabbedComboBox** realizes the **GX_IDS_CTRL_TABBED_COMBOBOX** control ID.
3.4.24 CGXTabbedComboBoxWnd

The CGXTabbedComboBoxWnd class implements a tabbed combo box based on the existing CGXComboBoxWnd implementation. CGXTabbedComboBoxWnd realizes the GX_IDS_CTRL_CBS_TABBED_DROPDOWN and GX_IDS_CTRL_CBS_TABBED_DROPDOWNLIST control IDs.

3.4.25 CGXVScrollEdit

The CGXVScrollEdit class implements a text input control that can be used to display and edit text in cells. The user can modify this text in place. The difference between CGXVScrollEdit and CGXEditControl is that CGXVScrollEdit displays a vertical scrollbar for multiline text cells by default. CGXVScrollEdit realizes the GX_IDS_CTRL_SCROLLEDIT control ID.

3.4.26 CGXWndWrapper

The CGXWndWrapper class is an adapter-like class for CWnd-derived objects. You can use any CWnd-derived object as a cell in the grid when you wrap it with CGXWndWrapper.
3.4.27 CGXChild

The CGXChild class is an abstract base class. CGXChild establishes a default control to child interface that derived control child classes must implement. CGXChild objects can be added to a CGXControl object and can be used for small buttons (for example the up- and down-arrow button in the CGXSpinEdit control).

3.4.28 CGXButton

The CGXButton class implements a pushbutton that can be used as a child in a control.

3.4.29 CGXBitmapButtonChild

The CGXBitmapButtonChild class implements a bitmap button that can be used as a child in a control.
3.5 Style Classes

Objective Grid implements an attribute architecture, or styles, that lets the developer manipulate grid attributes in a variety of ways. This section introduces the style classes.

Figure 35 – Objective Grid style class hierarchy

3.5.1 CGXStyle

The CGXStyle class contains all the information necessary for formatting a cell. A style consists of several attributes such as the text color, borders, control type and font attributes. All of these styles can be modified by the end user via the CGXStyleSheet dialog.

The CGXStyle class lets the developer extend the class at run time with specific user attributes that provide additional information about specific cells. An example of a user attribute would be an additional “Expression” attribute. The end user could modify the value for the “Expression” attribute with CGXStyleSheet, and the developer could interpret the “Expression” attribute in a subclassed control and so extend Objective Grid with some simple formula capabilities.

An advanced attribute of the CGXStyle class is the base style so that you can easily group specific kinds of cells and make them have similar attributes.
3.5.2 CGXStylesMap

Each grid owns one `CGXStylesMap` object that maintains grid-wide base styles. Base styles make it possible to group specific kinds of cells and make them have similar attributes. The predefined base styles are: row-header-style, column-header-style and standard-style. Row header cells inherit their attributes from row-header-style. Column headers inherit from column-header-style. Standard-style is the base style for all cells in the grid.

3.5.3 CGXStylesDialog

Class `CGXStylesDialog` provides a dialog that lets end users change the values of base styles found in the `CGXStylesMap`. The user can add new or change existing base styles through the `CGXStylesDialog`. For example, if the user changes the background color of the standard-style, all cells in the grid that do not have a specific background color will inherit the new background color. `CGXStylesDialog` also provides an option that allows the user to save/load the base styles using the registry/profile.

3.5.4 CGXStyleSheet

Class `CGXStyleSheet` encapsulates a property sheet that lets the end user specify a style. A style sheet can either apply changes to a range of selected cells or to one of the base styles. Please refer to Section 5.13 for a detailed description of this class and how to use it.

3.5.5 CGXFont, CGXBrush, CGXPen

These classes encapsulate Windows structures and add style features to them. For example, `CGXFont` encapsulates a `LOGFONT` structure and also adds serialization support and the ability to archive the font information to the registry/profile.

3.5.6 CGXAbstractUserAttribute

By deriving from `CGXAbstractUserAttribute`, you can store binary data into any `CGXStyle` object. Using a `CGXAbstractUserAttribute` will be much more convenient for you than using `SetItemDataPtr`, because you don’t have to worry about deleting the objects any more.

3.5.7 CGXUserAttribute

`CGXUserAttribute` provides support for number and string user attributes in the `CGXStyle` class. `CGXStyle::SetUserAttribute()` instantiates this default `CGXUserAttribute` class for strings and numbers automatically.
3.6 Data Structures

Objective Grid maintains grid properties and attributes in separate classes. This allows several grid views to share the same data. Objective Grid provides several pre-built dialogs that allow the end user to modify properties.

Figure 36 – Objective Grid data structure class hierarchy

- CObject
  - CPtrList
  - CGXRangeList
    - CGXGridParam
    - CGXProperties
    - CGXData
    - CGXPrintDevice
    - CGXCommand
    - CGXRange
  - CGXAbstractControlFactory
    - CGXControlFactory
  - CGXNoOleDataObjectProxy
  - CGXNoOleDataSourceProxy
  - CGXCommand
  - CGXDisplayPropertiesDialog
  - CGXPrintPropertiesDialog
  - CGXHeaderFooterDialog
  - CWnd
  - CDlg
3.6.1 CGXGridParam

CGXGridParam contains shared attributes such as the row heights, the cell contents, the styles map, the printer settings and the selected ranges. By default, an instance of CGXGridParam is created in CGXGridCore::OnGridInitialUpdate(), but can also be embedded in the document and linked to the grid.

3.6.2 CGXData

The CGXData class maintains the cell contents of the grid and the row and column base styles. The grid component creates an instance of this class in CGXGridCore::OnGridInitialUpdate() and attaches it to the parameter object.

3.6.3 CGXProperties

The CGXProperties class maintains the display and print properties of a grid, such as margins, header/footer and colors of grid lines. Objective Grid offers some pre-built dialog classes that allow the end user to modify these settings at run time. All settings except the header/footer text can be written to and loaded from the profile.

3.6.4 CGXCommand

The CGXCommand class is an abstract base class for storing Undo information for commands. If you want to support Undo/Redo for your commands, you should derive a class from CGXCommand and store all information necessary for undoing the operation into the command object. A detailed discussion of this process is included in the Programmer’s Guide section of this manual.

3.6.5 CGXAbstractControlFactory

CGXAbstractControlFactory is an abstract base class for registering new cell types together with user attributes. This class simplifies the registration of new cell types in the grid.

3.6.6 CGXControlFactory

CGXControlFactory is the default implementation of the Objective Grid control factory. CGXControlFactory registers all cell types, user attributes and window classes provided by Objective Grid and will force them to be linked in your application.
3.6.7 CGXGridDropTarget

A CGXGridDropTarget object provides the communication mechanism between a grid and the OLE libraries. Creating an object of this class allows a window to accept data through the OLE drag-and-drop mechanism.

3.6.8 CGXNoOleDataObjectProxy

CGXNoOleDataObjectProxy objects are used to pass COleDataObject objects to the OLE drag & drop overridables in CGXGridCore.

3.6.9 CGXNoOleDataSourceProxy

CGXNoOleDataSourceProxy objects used to pass COleDataSource objects to the OLE drag & drop overridables in CGXGridCore.

3.6.10 CGXDisplayPropertiesDialog, CGXHeaderFooterDialog, CGXPrintPropertiesDialog

These classes provide dialogs that allow the end user to change the settings maintained by the CGXProperties class. All dialogs provide an option that allows the user to save/load the settings using the registry/profile.

3.6.10.1 Saving, Restoring, and Customizing Colors

The CGXDisplayPropertiesDialog class uses CGXGridLineColorBox to display background colors. Grid gets its colors by calling GXGetVGAColor(), which has a hard-coded array of colors. An API has been added that allows you to customize this color array:

```c
// in gxext.h
GRID_API void AFXAPI GXSetVGAColor(int index, COLORREF rgbColor)
```

```c
// in gxdrawx.cpp
void AFXAPI GXSetVGAColor(int i, COLORREF rgbColor)
```

In addition, the display properties dialog did not used to have a way to back out color changes. The buttons RevertAll and Default have been added, allowing you to revert changes you have made before saving them and to restore the default colors, respectively.

3.6.11 CGXRange

CGXRange objects define a rectangular range of cells in the grid. A range is specified through a top and bottom row and a left and right columns. Ranges can represent a selection of cells, columns, rows or all cells.
3.6.12 CGXRangeList

CGXRangeList maintains a list of range objects. For example, ranges of cells selected by the user are stored in a CGXRangeList.

3.6.13 CGXPrintDevice

The CGXPrintDevice class maintains printer settings. These printer settings can be modified through the CPrintDialog dialog or by the programmer.
3.7 Browser Classes

The Objective Grid Browser classes allow you to easily browse any external data source by overriding a few virtual methods. Objective Grid already provides special implementations of the browser classes for displaying ODBC CRecordset and ADO results.

The MFC ODBC classes do not directly support the ability to determine database schema information at run time. Objective Grid provides a full set of ODBC classes that allow the end user to specify and view the results of an SQL query at run-time (where schema is unknown). Objective Grid also supports the MFC paradigm where the developer uses ClassWizard to create record sets with a known schema.

Figure 37 shows the browser class hierarchy.

Figure 37 – Objective Grid database browser class hierarchy
3.7.1  **CGXBrowseGrid**  
The *CGXBrowseGrid* encapsulates all the functionality necessary for browsing external data sources. By deriving your own class from *CGXBrowseGrid*, you can create Microsoft Query-like browsers for proprietary or unsupported database tool kits.

3.7.2  **CGXBrowseView**  
The *CGXBrowseView* class represents the *CGXBrowseGrid* class as view.

3.7.3  **CGXBrowseWnd**  
The *CGXBrowseWnd* class represents the grid component as child control. *CGXBrowseWnd* can be easily used as dialog control or in a workbook window.

3.7.4  **CGXBrowseParam**  
*CGXBrowseParam* holds specific data for the *CGXBrowseGrid*. It can be shared among several views. You can embed it in a document if you want to support several browse views of the same data as in a dynamic splitter window.

3.7.5  **CGXDbParam**  
*CGXDbParam* holds specific data for the *CGXODBCGrid*. It can be shared among several views. You can embed it in a document if you want to support several browse views on the same data as in a dynamic splitter window.

3.7.6  **CGXDynamicRecordset**  
This class provides the functionality to specify SQL query statements at run time for ODBC recordsets. Instead of creating a *CRecordset* derived class with ClassWizard, you can create a *CGXDynamicRecordset* object and specify the SQL query statements by passing a string to the object.

3.7.7  **CGXODBCGrid**  
The *CGXODBCGrid* class provides functionality for browsing ODBC data sources. This class lets you create Microsoft Query-like views. It uses the field exchange mechanism to exchange data between the grid and the recordset and supports adding, editing, and deleting records. *CGXODBCGrid* also supports find and replace, and freezing and dragging columns.
3.7.8 CGXRecordInfoBeam

This class provides the record status beam functionality used by CGXRecordInfoWnd. CGXRecordInfoBeam displays the current record in an edit field along with some navigation buttons. The end user can either click on the navigation buttons to scroll through the recordset or directly enter a record number in the edit field.

3.7.9 CGXRecordInfoWnd

This class displays a record status beam (similar to Microsoft Access or Microsoft Query) in the scrollbar.

3.7.10 CGXRecordInfoSplitterWnd

The CGXRecordInfoSplitterWnd embeds a non-resizable CGXRecordInfoBeam on the left side of the horizontal scrollbar of a CGXSplitterWnd. The user can use this CGXRecordInfoBeam to navigate through a grid just like you know from applications like Microsoft Access or Microsoft Query.

3.7.11 CGXRecordView

The CGXRecordView represents the grid-component as a view for browsing and changing recordsets.

3.7.12 CGXRecordWnd

The CGXRecordWnd represents the grid component as a CWnd or dialog-control for browsing and changing recordsets.

3.7.13 CGXRecSetInfo

CGXRecSetInfo is used by CGXBrowseParam to store information about the data source, the query string and the filename.

3.7.14 CGXAdoGrid

The CGXAdoGrid class provides functionality for browsing databases that can be accessed through any Microsoft OLE DB provider. This uses ADODB, an implementation of ADO optimized for use with Microsoft OLE DB providers. As for any CGXBrowseGrid derived class, this class lets you create Microsoft Query-like views where you can browse these databases.
3.7.15 CGXAdoParam

CGXAdoParam holds specific data for the CGXAdoGrid.

3.7.16 CGXAdoRecordView

The CGXAdoRecordView represents the CGXAdoGrid as a view for browsing and changing recordsets.

3.7.17 CGXAdoRecordWnd

The CGXAdoRecordWnd class represents the ADO grid browser as child control. CGXAdoRecordWnd can be easily used as a dialog control or in a workbook window.
3.8 **Utility Classes**

Objective Grid uses a variety of MFC extension classes for internal utility classes. These classes may be helpful in other parts of your applications.

Figure 38 – Utility class hierarchy

3.8.1 **CGXDrawingAndFormatting**

*CGXDrawingAndFormatting* encapsulates some general methods used for drawing and formatting text in the grid.

3.8.2 **CGXLongOperation**

The *CGXLongOperation* class implements a mechanism for executing large commands that take a long time. It will notify the user about the progress of the operation and give the user the option to abort and even rollback the operation.
3.8.3 CGXScrollTip

The CGXScrollTip class lets you display scroll tips when the user drags a scrollbar thumb.

3.8.4 CGXSplitterWnd

The CGXSplitterWnd class subclasses and enhances the MFC CSplitterWnd class. It gives you more options to customize the behavior of the CSplitterWnd class. CGXSplitterWnd allows you to specify the alignment of the splitter box. With MFC CSplitterWnd the splitter box is always positioned on top of the vertical scrollbar and on the left of the horizontal scrollbar. CGXSplitterWnd gives you the option to position the splitter box on the other sides of the scrollbars.

3.8.5 CGXTabBeam

The CGXTabBeam class provides the functionality of a sheet tab control. A tab beam control looks like a Microsoft Excel tab control. Normally, the control is used by the CGXTabWnd class to switch between several views or windows within a frame window.

3.8.6 CGXTabInfo

The CGXTabInfo class contains information for an associated tab of the tab-beam control. Each tab has its own CGXTabInfo object.

3.8.7 CGXTabWnd

The CGXTabWnd class provides the functionality to easily switch between several views in a frame window. The CGXTabWnd class displays a tab-beam control on the bottom-left side of the frame window. The functionality of the CGXTabWnd class is similar to the functionality of a workbook in Microsoft Excel.
3.9 **Formula Engine Classes**

Objective Grid treats the formula engine as a separate entity. You might consider Objective Grid as a front end to this engine that loads values from the engine when cells need to be drawn and that stores values or formulas back into the engine whenever cells are changed.

The formula engine is discussed in detail in Chapter 14, “The Formula Engine.”

Figure 39 – Objective Grid formula class hierarchy

3.9.1 **CGXFormulaSheet**

The *CGXFormulaSheet* class provides an interface to the Objective Grid formula engine. *CGXFormulaSheet* can be attached to a *CGXGridCore* object and replace the *CGXData* object that holds all the cell data. *CGXFormulaSheet* has special overrides for *GetStyleRowCol()* and *StoreStyleRowCol()* so that whenever cells are changed in the grid the actual change will be directly stored in the formula engine. Also other operations like moving, removing and inserting cells will be forwarded to the engine by overrides in *CGXFormulaSheet*.

3.9.2 **CGXWorksheetFunctions**

The *CGXWorksheetFunctions* class provides an extensible interface to the formula engine. You can specify what worksheet functions should get linked into your application. You can also add your own custom worksheet functions to the engine.

3.9.3 **CGXDefaultWorksheetFunctions**

The *CGXDefaultWorksheetFunctions* class has a lookup table with all built-in worksheet functions.
3.9.4 CGXFormula

*CGXFormula* is used by the `CGXFormulaSheet::ParseExpression()` method to store a compiled formula expression in binary form. You can later pass this *CGXFormula* object to `EvaluateExpression()`.
3.10 Plug-in Component Classes

Plug-in components are CWnd-derived classes that can share a window handle with another CWnd and process window messages for that window. This allows you to implement reusable window functionality that can be attached to various window classes. There is no need to inherit from an existing window class only to add a specific functionality.

Figure 40 – Objective Grid plug-in component class hierarchy

3.10.1 CGXPluginComponent

CGXPluginComponent is the base class for all plug-in components.

3.10.2 CGXMouseWheelPlugin

CGXMouseWheelPlugin provides support for mouse-wheel scrolling and zooming with the IntelliMouse.

3.10.3 CGXIntelliMousePlugin

CGXIntelliMousePlugin provides support for mouse-wheel scrolling, zooming and panning (auto-scrolling) with the IntelliMouse.
3.10.4 CGXGridMouseWheelPlugin

This class customizes some behavior of the CGXMouseWheelPlugin to make it work correctly with CGXGridCore.

3.10.5 CGXGridIntelliMousePlugin

This class customizes some behavior of the CGXIntelliMousePlugin to make it work correctly with CGXGridCore.

3.10.6 CGXScrollTipPlugin

The CGXScrollTipPlugin simplifies the adding of scroll tip support to your grid or other windows using the plug-in approach. CGXScrollTipPlugin provides default implementations of the WM_VSCROLL and WM_HSCROLL message that display the current scroll thumb position in the scroll tip window.

3.10.7 CGXCellTipPlugin

The CGXCellTipPlugin class simplifies adding cell tip support to your grid, using the plug-in approach. It manages showing and hiding the tip for the supported cell types.
4.1 Creating a Starter Application

This section discusses creating a simple application, using either the Objective Grid static libraries or the Objective Grid DLL.

4.1.1 Using the Objective Grid Static Libraries

The following steps describe how to build a starter application using the Objective Grid static libraries:

1. Create a new MFC Application MDI project with AppWizard. You may choose to use the MFC library As a shared DLL or As a statically linked library.

2. Include the resources needed for the grid component. To do this, select Resource View from the View menu. Right-click the root of the resource tree and choose Resource Includes… from the pop-up menu. Add the line:
   
   ```
   #include "grid\gxresrc.h"
   ```
   
   to the Read-only symbols directives: section and the line:
   
   ```
   #include "grid\gxres.rc"
   ```
   
   to the Compile-time directives: section. A message box warns that "Directive text will be written verbatim into your resource script and may render it un compilable." You can safely click OK.

3. Open the declaration (.h) and implementation (.cpp) files for the view class. Change the derivation for the view class to CGXGridView. All references to CView should be replaced by CGXGridView. The OnBeginPrinting(), OnEndPrinting(), OnPreparePrinting() and OnDraw() member functions created by AppWizard should either be removed from your view class or you should insert a call to the associated CGXGridView version of this method.

4. In the stdafx.h file, you should add the line:

   ```
   #include <grid\gxall.h>
   ```

   Also add the header files that are common to all Stingray Studio products:
4.1.2 Using the Objective Grid Extension DLLs

The following steps describe how to build a starter application using the Objective Grid Extension DLL:

1. Create a new MFC Application MDI project with AppWizard. You must choose to use the MFC library As a shared DLL.
2. Perform Steps 2 through 5 from Section 4.1.1, “Using the Objective Grid Static Libraries.”
3. In the configuration settings, add the preprocessor directive _GXDLL to your solution’s C++ compiler settings. (Make sure you add it to all configurations.) Alternatively, add the following to the file stdafx.h:

```cpp
#define _GXDLL
```
4. Compile and run the project.

4.1.3 Converting an Application to Use the Objective Grid Extension DLL

If you are using the internal makefile with most of the standard defaults, you can easily change the project to build the DLL version. Here are the steps:

1. Open your project in Visual Studio.
2. Open the Configuration Settings dialog box by selecting Project Properties | Settings...
3. On the General tab, select Use MFC in a Shared DLL from the Microsoft Foundation Classes combo box.
4. On the C/C++ tab, add _GXDLL to the Preprocessor definitions.
5. Click OK.
6. Do a Rebuild All for your application.
4.2 Using Objective Grid in a Dialog Box

This section discusses integrating Objective Grid into a dialog box.

4.2.1 Objective Grid vs. ActiveX Controls

Objective Grid is a custom grid component that is MFC based and does not implement ActiveX control interfaces. This does not allow the grid to be directly displayed in the dialog editor as ActiveX controls can be. (With ActiveX controls you will choose Insert Component and point to the control and Microsoft Visual Studio would generate the wrapper files as well as ensure that the ActiveX appeared in the dialog editor.)

The grid object is typically created and displayed using the custom control placeholder in the dialog editor. You can create a window on any dialog.

The next section discusses the steps that are required to accomplish this.

4.2.2 Using Objective Grid in a Dialog Template

The following steps are necessary to embed a grid in a dialog.

1. Create a new class with the View | Add Class menu. Use “CWnd” as the class type.

2. Next, using a text editor, change the derivation in your window class, in both the .h and .cpp files. All references to CWnd should be replaced with CGXGridWnd. A simple find and replace operation will work here.

3. Create or open your dialog template. Choose the user control icon, drag it into the dialog and open its property page. Enter GXWND as class name into the class box. Please note that this is a registered WNDCLASS and not the C++ class name. CGXGridWnd will not work here.

4. Specify the style bits for the grid:
   - 0x50b10000 to display both scrollbars and a border
   - 0x50810000 to display no scrollbars but a border

   For other style bits see the definitions for Windows style bits in your windows.h header file.

5. If you created a new dialog, you should now create your dialog class with ClassWizard.

6. Embed an object of your derived grid class in the dialog.

   // dergriddlg.h : header file
   //

   ///////////////////////////////////////////////////////////////////////////
   // CDerGridDialog dialog
   //

   #include "dergrid.h"
class CDerGridDialog : public CDialog
{
// Construction
public:
    CDerGridDialog(CWnd* pParent = NULL); // standard constructor

// Dialog Data
    enum { IDD = IDD_GRIDDIALOG }; // NOTE: the ClassWizard will add data members here
//}}AFX_DATA

// You do not have to derive from CGXGridWnd unless you want
// to handle some events, etc. You can also declare m_wndGrid
// to be an instance of CGXGridWnd if you do not wish to
// handle any of the grid virtuals.

CDerivedGridWnd m_wndGrid;
...
};

7. Add the OnInitDialog member to the dialog class. In OnInitDialog, you can initialize
   the grid:

    //////////////////////////////////////////////////////////////////////////
    // CDerGridDialog message handlers
    // CDerGridDialog::OnInitDialog
    BOOL CDerGridDialog::OnInitDialog()
    {
        CDialog::OnInitDialog();

        // Please refer to the MFC documentation on
        // SubclassDlgItem for information on this call. This
        // makes sure that our C++ grid
        // window class subclasses the window that is created
        // with the User Control.
        m_wndGrid.SubclassDlgItem(IDC_GRIDSAMPLE, this);

        // Initialize the grid. For CWnd based grids this call is
        // essential. For view based grids this initialization is
        // done in OnInitialUpdate.
        m_wndGrid.Initialize();

        m_wndGrid.GetParam() -> EnableUndo(FALSE);

        m_wndGrid.SetRowCount(100);
        m_wndGrid.SetColCount(20);

        m_wndGrid.GetParam() -> EnableUndo(TRUE);

        return TRUE; // return TRUE unless you set the
        // focus to a control
        // EXCEPTION: OCX Property Pages should return FALSE
    }
If you want to use the grid in a formview, you should override OnInitialUpdate instead of OnInitDialog. Otherwise, the steps are identical.

8. Now, you can compile the class. The grid should display when you bring up the dialog. If the grid does not show up as expected, please refer to the next section before contacting Rogue Wave Support Services.

4.2.3 Troubleshooting

If the grid does not show up when you compile the application:

1. Check whether you have called GXInit() in your application's InitInstance. GXInit() is required in all applications that make use of the grid library.

2. Check whether you have included the grid resources if you are linking statically with the grid. For more information on this, see Section 4.1.1.

3. If you have OLE controls on your dialog, check whether you have called AfxEnableControlContainer().

4. Verify that the class name in the user control properties box is GXWND. This refers to the registered WNDCLASS name and not to the C++ class name.
## 4.3 Miscellaneous Tips

### 4.3.1 Checklist for Build Configurations

Table 3 – Build Configurations

<table>
<thead>
<tr>
<th>Type</th>
<th>Objective Grid build</th>
<th>MFC build</th>
<th>Compile Time Directives for resources*</th>
<th>Pre-processor definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static</td>
<td>Static</td>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static</td>
<td>DLL</td>
<td>Yes</td>
<td>_AFXDLL</td>
<td></td>
</tr>
<tr>
<td>DLL</td>
<td>DLL</td>
<td>No</td>
<td>_AFXDLL, _GXDLL</td>
<td></td>
</tr>
<tr>
<td>DLL</td>
<td>Static</td>
<td>Not supported</td>
<td>Not supported</td>
<td></td>
</tr>
</tbody>
</table>

| Custom Objective Grid Extension DLL: |                      |           |                                        |                           |
| Static              | DLL                  | Yes       | _AFXEXT                                |                           |
| DLL                 | DLL                  | No        | _AFXEXT, _GXEXT                        |                           |

| Custom Objective Grid regular DLL: |                      |           |                                        |                           |
| Static              | DLL                  | Yes       | _AFXDLL                                |                           |
| DLL                 | DLL                  | No        | _AFXDLL, _GXDLL                        |                           |

*Include "grid\gxres.rc" in Compile-time directives.*

You should always include "grid\gxresrc.h" in Read-only symbols so you can access the grid resources.

Custom macro definitions should be used in custom Objective Grid Extension DLL; these definitions are provided in the file CustExtDefs.h.

Add "_CUSTEXTDLL" to your C++ Preprocessor Definitions.
// Add "_AFXDLL" to your C++ Preprocessor Definitions.
// Make sure the project Configuration Type is set to "Dynamic Library (.dll)."
// Make sure the project Use of MFC is set to "MFC in a Shared Dll."
// For Debug, set the Code Generation Runtime Library to /MDd.
// For Release, set the Code Generation Runtime Library to /MD.
//Code:
// Include this header file in your project.
// Add "#define _CUSTEXTIMPORT_IMPL" to the top of stdafx.cpp.

4.3.1.1 Miscellaneous Configurations

- Application that links to a custom Objective Grid Extension DLL as well as the Objective Grid Extension DLL;
- Application that links to a custom Objective Grid Extension DLL as well as the Objective Grid static lib;
- Application that links to a custom Objective Grid Regular DLL as well as the Objective Grid Extension DLL;
- Application that links to a custom Objective Grid Regular DLL as well as the Objective Grid static lib;

The settings for all the above miscellaneous configurations can be resolved from the settings for the basic three types of configurations.

Take, for example, an application that links to a custom Objective Grid Extension DLL as well as the Objective Grid Extension DLL. Its settings are similar to an application that links to an Objective Grid Extension DLL: it should include _GXDLL in its preprocessor settings and should not include the resources in its compile time directives.

4.3.2 Objective Grid and VC++

4.3.2.1 Building and Using VC++ Libraries

The solution files required for building OG and the samples under all supported versions of VC++ are included in the appropriate directories as Grid<ver>.sln, where <ver> represents the VC++ version. After rebuilding the libraries, the applications linking to those libraries should also be recompiled in order for them to link and work properly.

4.3.2.2 Make Files and Building Directly with nmake

When you build the Stingray libraries in Visual Studio, Visual Studio invokes make files that ship with the product. For information on these underlying make files, and how to build the libraries by invoking nmake on these files directly, see Section 2.2, “Building from the Command Line with nmake,” in the Stingray Studio Getting Started Guide.

This section also discusses the issue of building the libraries with 1-byte structure alignment rather than the default 8-byte structure alignment.
4.3.2.3 Known Issues in VC++ Unicode Libraries

- Unicode Libraries return read-only recordsets in database applications using ODBC if the default snapshot option is used to open the recordset.

The \texttt{CRecordset} class in VC++ does not return writable recordsets in Unicode. Because of this, even though ODBC grid libraries under Unicode display recordsets properly, the recordsets cannot be updated. When trying to update grids such as these, the following error may be displayed: "Error: Invalid Option/identifier".

As a workaround, \texttt{CRecordset::dynaset} or \texttt{CRecordset::dynamic} options should be used to open recordsets.

- Unicode is not supported in this version of the Excel read\write classes, so the option \texttt{ExcelReadWrite} must be \textit{unchecked} in the Grid BuildWizard to allow building Grid libraries with Unicode.

4.3.2.4 Objective Grid AppWizard

All of the OG AppWizards under VC++ change the default charset to Unicode. The MFC Application Wizard under VC++ by default includes the manifest file that points the application to link to \texttt{comctrl32.dll}.

4.3.2.5 Objective Grid Designer Utility

The source for the Objective Grid Designer Utility is available under the \texttt{StingrayInstallation\utils} directory, and can be compiled with any of the supported compiler versions.

4.3.2.6 Setting Paths in Visual Studio

If you are using Visual Studio 2010 or higher, the Stingray Studio solution and \texttt{vcxproj} files now use Microsoft’s Property Sheets to set these paths. Please refer to Section 2.6.3, “Stingray Studio Paths in Property Sheets,” in the \textit{Stingray Studio Getting Started Guide} for information on how to add property sheet(s) with Stingray Studio paths.

4.3.3 Objective Grid and Windows Look and Feel

Objective Grid can take advantage of a number of Windows visual styles, including Windows Vista, Windows XP, Office 2003, and Windows Classic. The next section describes a general mechanism for enabling these styles.

4.3.3.1 Vista Look and Feel

The Vista Classic style feature allows your applications to take on a look and feel similar to Windows Vista Classic. This visual style is theme-enabled, and requires Windows Vista. The style applies to MFC-derived and Stingray custom controls.
Applications and samples enable a particular drawing style with the function call `RWSetVisualStyle()`. Using this function requires inclusion of the header file `RWUXTheme.h`, which is found in the `<stringray-installdir>\RWUXTheme\Include` directory.

`RWSetVisualStyle()` should be placed in the application or sample’s main `InitInstance()` or `InitDialog()` function call. The call should look like this:

```c
RWSetVisualStyle(RW_VISUALSTYLE_VISTACLASSIC);
```

The following styles are available through this call:

- `RW_VISUALSTYLE_VISTACLASSIC`
- `RW_VISUALSTYLE_DOTNET`
- `RW_VISUALSTYLE_OFFICE2003`
- `RW_VISUALSTYLE_WINDOWSCLASSIC`

### 4.3.3.2 XP Look and Feel

On Windows XP, Objective Grid can take advantage of XP visual styles. All the samples in Objective Grid are by default visual styles-enabled. The visual style applied to controls depends on the Windows theme enabled in the system. You can use the same mechanism described in the previous section to enable the XP look and feel.

If the visual styles feature must be turned off (although the manifest file is still included as a resource), call

```c
RWSetVisualStyle( RW_VISUALSTYLE_WINDOWSCLASSIC );RWSetThemeAppProperties(4);
```

in the `InitInstance()` of the application. This will ensure that the theme (even though active) is not applied to the application in question.

For details on how Objective Grid behaves differently under Windows XP, see Section 13.7, “Objective Grid and Windows XP Visual Styles.”
4.4 Using Objective Grid in a WinForm

One straightforward way to integrate MFC-based projects with .NET libraries is to set the /clr compiler option and use C++/CLI. Stingray Objective Grid samples are provided with a /clr build configuration to demonstrate compatibility with this approach.

This section discusses two possible ways for integrating Objective Grid into a WinForm:

- Use C++/CLI projects to employ Windows Forms and apply CGXGridWnd-based classes directly.
- Create .NET controls wrapping classes from Objective Grid libraries and use these controls with any .NET language, particularly C#, VB, C++/CLI.

Simple samples that demonstrate these approaches are shipped with Stingray Studio:

- `<install_dir>\Samples\Grid\Integration with .NET\GridForm`
- `<install_dir>\Samples\Grid\Integration with .NET\GridControl`

More advanced samples are available in the Rogue Wave Knowledge Base kb.roguewave.com/kb/). Specifically, see Using Objective Grid with C++/CLI at kb.roguewave.com/kb/index.php?View=entry&EntryID=1400.

The procedures below reference Section 4.1.1, “Using the Objective Grid Static Libraries,” in this chapter for more detailed information on how to perform particular steps.

4.4.1 Direct Use of Objective Grid in a WinForm

To embed a grid in a WinForm:

1. Use the Visual Studio App Wizard to create a Visual C++\CLR\Windows Form Application project GridForm.
2. Add grid resources, as described in Step 2 of Section 4.1.1.
3. Add #include to the stdafx.h file, as described in Step 4 of Section 4.1.1.
4. Add GXInit() to the InitInstance() method, as described in Step 5 of Section 4.1.1.
5. Add the following immediately after class CmyApp (to initialize grid libraries):

   ```cpp
   CMyApp theApp;
   ```
6. Add a member to class Form1:

   ```cpp
   private: CGXGridWnd *m_pGX;
   ```
7. Initialize the grid, using this code:

   ```cpp
   private: System::Void Form1_Load(System::Object^ sender, System::EventArgs^ e)
   {
   m_pGX = new CGXGridWnd();
   if (!m_pGX) return;
   DWORD dwFlags = WS_TABSTOP | WS_BORDER
   ```
CRect rect(panel1->ClientRect.Left,
    panel1->ClientRect.Top,
    panel1->ClientRect.Right,
    panel1->ClientRect.Bottom);

BOOL bOk = m_pGX->Create(dwFlags, rect,
    CWnd::FromHandle((HWND)panel1->Handle.ToInt32()), 32000);

m_pGX->Initialize();

m_pGX->GetParam()->SetSortRowsOnDblClk(TRUE);

m_pGX->GetParam()->EnableMoveCols(TRUE);

m_pGX->GetParam()->EnableMoveRows(TRUE);

m_pGX->GetParam()->EnableSelection(TRUE);

m_pGX->LockUpdate(TRUE);

m_pGX->SetRowCount(15);

m_pGX->SetColCount(5);

m_pGX->SetStyleRange(CGXRange(1,1),
    CGXStyle().SetControl(GX_IDS_CTRL_PUSHBTN));

m_pGX->LockUpdate(FALSE);
}

8. Clean up memory in the destructor:

~Form1()
{
    if (m_pGX)
        delete m_pGX;
    if (components)
    {
        delete components;
    }
}

9. Optionally, add _GXDLL to the Preprocessor definitions, as described in Step 4 of Section 4.1.3.

10. Build and run the project.

### 4.4.2 Use of Objective Grid in a .NET Control

To embed a grid in a WinForm:

1. Use the Visual Studio App Wizard to create a Visual C++\CLR\Windows Form Control Library project Grid.

2. Add grid resources, as described in Step 2 of Section 4.1.1.

3. Add #include to the stdafx.h file, as described in Step 4 of Section 4.1.1.
4. Add the following immediately after `namespace Grid {…}:

    CMyApp theApp;

5. Add this member to class `GridControl`:

    private: CGXGridWnd *m_pGX;

6. Initialize the grid, using this code:

    

```cpp
private: System::Void GridControl_Load(System::Object^ sender,
                                         System::EventArgs^ e)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    GXInit();
    m_pGX = new CGXGridWnd();
    if (!m_pGX) return;
    DWORD dwFlags = WS_TABSTOP | WS_BORDER
                     | WS_VISIBLE | WS_VSCROLL | WS_HSCROLL;
    CRect rect(ClientRectangle.Left, ClientRectangle.Top,
               ClientRectangle.Right, ClientRectangle.Bottom);
    BOOL bOk = m_pGX->Create(dwFlags, rect,
                              CWnd::FromHandle((HWND)this->Handle.ToInt32()),32000);
    m_pGX->Initialize();
    m_pGX->GetParam()->SetSortRowsOnDblClk(TRUE);
    m_pGX->GetParam()->EnableMoveCols(TRUE);
    m_pGX->GetParam()->EnableMoveRows(TRUE);
    m_pGX->GetParam()->EnableSelection(TRUE);
    m_pGX->LockUpdate(TRUE);
    m_pGX->SetRowCount(15);
    m_pGX->SetColCount(5);
    m_pGX->LockUpdate(FALSE);
}
```

7. Expose the APIs to call from the parent program:

```cpp
public: System::Void SetValueRange(int t, int l, int b, int r,
                                    String^ value)
{
    m_pGX->SetValueRange(CGXRange(t, l, b, r), value);
}
public: System::Void SetControl(int t, int l, int b, int r, WORD nControl)
{
    m_pGX->SetStyleRange(CGXRange(t, l, b, r),
                          CGXStyle().SetControl(nControl));
}
```

8. Clean up memory, as described in Step 8 of Section 4.4.1.

9. Build the control.

10. Add a new project to the solution in any .NET language program; here, we’ll use Other Languages | Visual C# | Windows Forms Application.

11. Open `Form1`. 
12. Select **Tools | Choose Toolbox Items**, then click **Browse**. Navigate to and select **Grid.dll** built in Step 9.

**For x64 Build Configurations:**

For x64 build configurations (or “AnyCPU” on an x64 machine), any control built with `/clr` (or even `/clr:pure`) throws an error when you try to add it to the Toolbox; controls built only with `/clr:safe` are valid for the Toolbox. This is a current Visual Studio limitation. As a workaround, you can use a control built with x64 and `/clr` by adding code to WinForms manually, rather than using the Toolbox.

To do so, set the configuration to Win32, use the Toolbox to add the control built with Win32, and **then** set the x64 build configuration and build it. The reference should be set to the GridControl project, rather than to a particular assembly.

Alternatively, save the Windows Forms Designer generated code and use it for initialization of an x64 control in a project with an x64 build configuration, like so:

```csharp
// Add a member to class Form1
private Grid.GridControl gridControl1;

// Use following code to initialize the control
this.gridControl1 = new Grid.GridControl();
this.gridControl1.Location = new System.Drawing.Point(77, 57);
this.gridControl1.Name = "gridControl1";
this.gridControl1.Size = new System.Drawing.Size(350, 235);
this.gridControl1.TabIndex = 0;
```

13. Add a reference to the Grid project in the Windows Forms Application.

14. Initialize GridControl with the following code:

```csharp
private void Form1_Load(object sender, EventArgs e)
{
    this.gridControl1.SetValueRange(1, 1, 1, 1, "Value");
    this.gridControl1.SetControl(2, 1, 2, 1,
        GX_IDS_CTRL_PUSHBTN);
}
```

To use an embedded pushbutton, you can add the following to class **Form1**:

```csharp
private ushort GX_IDS_CTRL_PUSHBTN = 52514;
```

15. Build and run the solution.
5.1 Cell Value

This section discusses the manipulation of grid values in a regular grid. The Objective Grid class methods presented in this section are the most commonly used subset of available methods. Other sections will discuss relevant methods for manipulating cell values as necessary. For more information about these or other methods, please consult the Objective Grid Class Reference.

5.1.1 Filling the Grid with Values

There are two common methods for putting values into a grid:

1. CGXGridCore::SetValueRange() — This is a good method to use when only the cell value needs to be changed.

2. CGXGridCore::SetStyleRange() — This is a good method to use when the cell attributes need to be changed in addition to the value. This usually occurs during (but is not limited to) initialization. The method accepts a style object. You can set the cell value in the style object using CGXStyle::SetValue().

If you are making changes to multiple cells, you should lock the updating mechanism in the grid object using CGXGridCore::LockUpdate(). This will prevent unnecessary redrawing and flickering during multiple updates.

```cpp
// Lock the grid update mechanism
BOOL bOldLock = LockUpdate(TRUE);

// Fill cell values with SetValueRange
SetValueRange(CGXRange(nRow, 1), str); // string
SetValueRange(CGXRange(nRow, 2), d);    // number

// Fill cell values with SetStyleRange
SetStyleRange(CGXRange(nRow, 3),
              CGXStyle()
              .SetValue(str)
              );
```
How Objective Grid stores the cell values is very important. There are APIs supplied to set cell values based on numerous data types ranging from `double` and `float` to `CString` and `LPCTSTR`. Objective Grid always stores those values as strings. When data is passed as anything but a string, Objective Grid converts it using the appropriate precision and then stores it as a string.

### 5.1.2 Getting Values from the Grid

There are two methods for getting values from the grid:

- `CGXGridCore::GetValueRowCol()`
- `CGXStyle::GetStyle()`

Both methods return a `CString` representation of the cell’s value. If you need the value as a different data type, you should convert it.

```cpp
// String
CString strValue = GetValueRowCol(nRow, nCol);

// Number
double d = atof(GetValueRowCol(nRow, nCol));

// Via style
CString strStyleVal = _T("" component=""nRow," component=""nCol," component=""&style;" component=""style.GetValue();"
if (style.IncludeValue())
{
  strStyleVal = style.GetValue();
}
```

Changes in cell values made by the end user interactively are stored when the cell becomes inactive. This is important to know, because the above two methods return the stored cell value. Calling these methods on an active cell will not reflect a pending value change. With this in mind, there are two ways to get the cell value regardless of the cell’s current state.

- Call `CGXGridCore::TransferCurrentCell()` before getting the cell value. This solution is appropriate if maintaining the current state of the cell is not necessary. `TransferCurrentCell()` stores and deactivates the current cell. Calling this method is very useful if you want to ensure that the current cell’s contents are transferred to the grid before performing specific operations with grid data.
Implement the following `GetCellValue()` method:

```cpp
CString CMyGrid::GetCellValue(ROWCOL nRow, ROWCOL nCol)
{
    if (IsCurrentCell(nRow, nCol) && IsActiveCurrentCell())
    {
        CString s;
        CGXControl* pControl = GetControl(nRow, nCol);
        pControl->GetValue(s);
        return s;
    }
    else
    {
        return GetValueRowCol(nRow, nCol);
    }
}
```

This method will return the current value for a given cell, regardless of the active state of the cell. It will not change the active state either. This method is useful when you need to preserve the current state of the cell.

### 5.1.3 Clearing Cell Values

To empty a cell or range of cells, call:

```cpp
ClearCells(CGXRange(1, 1, nRows, nCols), FALSE);
```

Note that the second parameter is `FALSE`. This indicates that only the cell value should be cleared. Passing `TRUE` for this parameter will cause all cell style attributes to be cleared.
5.2 Cell Attributes

5.2.1 Reusing Style Objects

You should never create style objects on the heap yourself. While it is not necessarily harmful to your code to do so, it is considered bad practice in the Objective Grid world. If you need to create and destroy style objects dynamically, you should use `CGXGridCore::CreateStyle()` and `CGXGridCore::RecycleStyle()`.

`CreateStyle()` returns a pointer to a style object created on the heap. `RecycleStyle()` returns the style object back to Objective Grid. The grid object maintains a list of style objects created on the heap. There are two advantages to using this approach:

1. It is more efficient because the list of style objects enables them to be reused. There are fewer allocations and deallocations of memory.
2. There is no explicit type specified. This is helpful when you want to subclass `CGXStyle`. You can be sure that Objective Grid will always create the subclassed style object instead of a `CGXStyle` object by overriding `CreateStyle()`.

5.2.2 Disabled and Read-Only Cells

You can enable and disable individual cells with

`CGXStyle::SetEnable()`

Enables or disables a cell.

You can change the read-only state using

`CGXStyle::SetReadOnly()`

Sets a cell to be read-only.

`CGXGridCore::SetReadOnly()`

Sets entire grid to be read-only.

`CGXGridCore::SetLockReadOnly()`

Enables or disables the read-only state of cells.

For a detailed explanation, please refer to the Objective Grid Class Reference.
5.2.2.1 What Are Disabled Cells?

Disabled cells are cells that cannot become current. It is not possible to position the current cell onto a disabled cell. When the user navigates with arrow keys through the grid, disabled cells will be skipped. When the user clicks the mouse on a disabled cell, nothing will happen. It is recommended that you also set disabled cells to be read-only. Otherwise, the user can select a range of cells and press DELETE to clear all cells in the selected range.

5.2.2.2 What Are Read-Only Cells?

Read-only cells are cells that cannot be changed. Any subsequent calls to SetStyleRange() or SetValueRange() are ignored and FALSE is returned to indicate the operation failed.

Please note that although an edit cell cannot be changed when you set it read-only, the user can still activate the cell and make the text caret visible. If you want the cell to stay static even when the user clicks into the edit cell, you should change the cell type to static by calling SetControl(GX_IDS_CTRL_STATIC).

When you have read-only cells in the grid, these cells are protected against any changes from the end user and from the programmer. After setting cells to read-only, you cannot apply any changes programmatically to the cell, nor can the user make changes from the user interface. Although this makes sense, it is often necessary for the programmer to change the contents of read-only cells. SetLockReadOnly() enables the programmer to do this.

SetLockReadOnly() toggles the protection of read-only cells in the grid object. You can change read-only cells by calling:

GetParam()->SetLockReadOnly(FALSE)

This call disables the read-only mode for all cells so you can apply your changes. After changing the cells, you should call GetParam()->SetLockReadOnly(TRUE). This will enable the protection for read-only cells again.

GetParam()->SetLockReadOnly(FALSE);
// make changes to read-only cells
GetParam()->SetLockReadOnly(TRUE);

Note that SetLockReadOnly() enables and disables read-only protection for all cells. This means the user can also make changes. For this reason, you should always use SetLockReadOnly(FALSE)/SetLockReadOnly(TRUE) in a tight code stack where it is not possible for the user to make changes.

To make read-only cells writable again, you have to turn read-only protection off, make the style change, then turn read-only protection back on.

GetParam()->SetLockReadOnly(FALSE);
SetStyleRange(range, CGXStyle().SetReadOnly(FALSE));
GetParam()->SetLockReadOnly(TRUE);
5.2.3 Covered Cells

Objective Grid lets you cover cells. To cover cells means that one cell can span several other cells. This is very useful for headings in reports.

CGXGridCore::SetCoveredCellsRowCol() is an overridable command for setting the covered cells-range for a cell. Covered cells are stored with a call to StoreCoveredCellsRowCol(). The display is updated with UpdateCoveredCellsRowCol().

To uncover a covered range, call this method with only the top-left cell of the original covered range:

StoreCoveredCellsRowCol(nRow, nCol, nRow, nCol, FALSE);

The parameter-object maintains a list of ranges with the covered cells. It makes no sense to override StoreCoveredCellsRowCol() and store the covered cells anywhere else.

5.2.4 Floating Cells

To enable floating cells, add the following line in OnInitialUpdate() or OnInitDialog():

SetFloatCellsMode(gxnFloatDelayEval);

With floating cells enabled for the grid, you can enable and disable floating for individual cells with

- CGXStyle::SetFloatCell() — allow a cell to float over others
- CGXStyle::SetFloodCell() — prevent a cell from being flooded (hidden) by another floating cell

For a detailed explanation of the various modes, please refer to the Objective Grid Class Reference.

5.2.4.1 What Are Floating Cells?

Floating cells work very much like covered cells with the difference that you can still access and change the individual flooded cells. For example, if cells A5 to E5 are floated, you can still click and change cells in the range B5 to E5. If you change cell D5, cells D5 and E5 will automatically become unflooded.
Floating cells have to be calculated depending on the size of the text in a cell. When you type text into a cell, the size of the text will always be calculated. In the case that the text does not fit into a cell, the cell will automatically be expanded (similar to Microsoft Excel).

### 5.2.5 Merged Cells

To enable cell merging, add the following line in `OnInitialUpdate()` or `OnInitDialog()`:

```c
SetMergeCellsMode(nMode);
```

`nMode` can be one of:

- `gxnMergeDisable` - Disable merge cells.
- `gxnMergeDelayEval` - Delay evaluation of merge cells.
- `gxnMergeEvalOnDisplay` - Always reevaluate merge cells before they are displayed.

`nMode` may be combined with any of the following optional flags:

- `gxnMergeHorzOnly` - Cells can only be merged horizontally.
- `gxnMergeVertOnly` - Cells can only be merged vertically.
5.2.5.1 What Are Merged Cells?

Merged cells are cells that automatically join neighboring cells when both cells have the same value or style settings.

A very popular use for merge cells is the display of cross tables or other reports where the information is grouped by different categories.

For example, if you have a report like:

<table>
<thead>
<tr>
<th>Year</th>
<th>Month</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1997</td>
<td>Jan</td>
<td>5</td>
</tr>
<tr>
<td>1997</td>
<td>Feb</td>
<td>4</td>
</tr>
<tr>
<td>1997</td>
<td>Mar</td>
<td>4</td>
</tr>
<tr>
<td>1997</td>
<td>Apr</td>
<td>2</td>
</tr>
<tr>
<td>1997</td>
<td>Jun</td>
<td>8</td>
</tr>
</tbody>
</table>

with merge cells, the year will only be displayed as one big cell and not repeated for every row.

Figure 42 – Merged Cells
5.2.6 Registration and Usage of Base Styles

The `CGXStylesMap` class maintains grid-wide styles that make it possible to group specific kinds of cells and make them have similar attributes. The predefined base styles are: row-header-style, column-header-style, and standard-style. Row header cells inherit their attributes from row-header-style. Column headers inherit their attributes from column-header-style. Standard-style is the base style for all cells in the grid.

You can modify these predefined base styles by calling the `CGXGridCore` member functions `ChangeStandardStyle()`, `ChangeRowHeaderStyle()`, and `ChangeColHeaderStyle()` to change their state.

```cpp
// change the standard style. This changes the default settings
// for all cells in the grid.
ChangeStandardStyle(CGXStyle()
   .SetFont(CGXFont().SetFaceName("Courier").SetSize(10))
   .SetWrapText(FALSE));
```

If you want to extend the grid with additional base styles, you need to register them in the `CGXStylesMap` object.

The following example shows you how to:

1. Register a base style.
2. Change the base style with the `BaseStyle()` method.
3. Apply the base style to cells.

The base style settings can be loaded from or written to the `My base styles` profile/registry section.

```cpp
class CMyGrid: public CGXGridView
{
   ...

   // Attributes
   public:
      WORD m_wStyleError;
   ...
};

void CMyGrid::OnInitialUpdate()
{
   // Attach parameter object to the grid
   SetParam(&GetDocument()->m_param, FALSE);

   // check for existence of stylesmap object
   if (GetParam()->GetStylesMap() == NULL)
   {
      // Create one
      CGXStylesMap* stylesmap = new CGXStylesMap;

      // standard styles
      stylesmap->CreateStandardStyles();
   }
}```
// Register additional base styles
stylesmap->RegisterStyle(szErrorStyle,
    CGXStyle()
        .SetTextColor(RGB(255,255,0)) // yellow
        .SetInterior(RGB(255,0,0))     // red
        .SetFont(CGXFont()
            .SetBold(TRUE)
        ),
    TRUE // system-style (non removable)
); // load profile/registry settings
stylesmap->SetSection("My base styles");
stylesmap->ReadProfile();

// attach styles-map object to parameter-object
GetParam()->SetStylesMap(stylesmap);

CGXGridView::OnInitialUpdate();

// determine the identifiers for the base styles
m_wStyleError = GetParam()->GetStylesMap()->
    GetBaseStyleId(szErrorStyle);

// Now you can use this identifier to change the style
// e.g.
BaseStyle(m_wStyleCombo)
    .SetChoiceList("one\n\ntwo\n\nthree");

// where BaseStyle(id) is a member function
// of CGXGridCore which returns a
// reference to the specified base-style.
//
// - Or -
//
// you can to apply this base-style to cells
// e.g.
SetStyleRange(CGXRange(5,2,8,3),
    CGXStyle()
        .SetBaseStyle(m_wStyleError)
);}

You can also apply base style to cells at run time (depending on the context of the cell) by overriding the GetStyleRowCol() method. The following example assigns the Error Style to the cell if the cell’s value is negative. The appearance of the Error Style can be modified by the end user with the pre-built CGXStylesDialog dialog.

BOOL CMyGrid::GetStyleRowCol(ROWCOL nRow, ROWCOL nCol,
    CGXStyle& style, GXModifyType mt, int nType)
{
    if (CGXGridView::GetStyleRowCol(nRow, nCol,
        style, mt, nType))
    {
        if (mt == gxRemove || nType == -1)
            return TRUE;
        if (mt == gxSet)
            style = m_wStyleError;
        else
            style = m_wStyleError;
    }
    return FALSE;
}
if (style.GetIncludeValue() &&
    style.GetDoubleValue() < 0)
{
    // Negative value should be outlined
    // with "Error Style"
    style.SetBaseStyle(m_wErrorStyle);
}

return TRUE;
}
return FALSE;

### 5.2.7 Registration and Usage of User Attributes

You can store binary information into any `CGXStyle` object by using a derived `CGXAbstractUserAttribute` object. This technique is much more convenient than using `SetItemDataPtr()`, because you don’t have to worry about deleting objects.

Objective Grid offers a default `CGXUserAttribute` class that provides support for numbers and strings. Using `CGXUserAttribute` decreases the memory usage of user attributes. Numbers will only be stored as doubles and no string needs to be allocated. If you pass a string to `CGXStyle::SetUserAttribute()`, the value will be parsed, and if it is a plain number, it will be stored as a `double`.

In addition, user attributes increase the flexibility of the styles architecture. It is much easier now to add user attributes than to subclass `CGXStyle` to add built-in attributes. `CGXAbstractUserAttribute` comes with full support for serialization and reading and writing to the profile or registry. User attributes can also be modified by the user via the `CGXUserAttributePage`.

If you want to create a user attribute class for your own binary objects, you have to override `Clone()`. This method is called in order to copy your binary object from one style object to another.

```cpp
// Copying attribute (e.g. from one style object to another)
CGXAbstractUserAttribute* CGXUserAttribute::Clone() const
{
    return new CGXUserAttribute(*this);
}
```

If you want to add support for OLE Drag & Drop, cutting to and pasting from the clipboard, and Serialization you should also override `Serialize()`.

```cpp
void CGXAbstractUserAttribute::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
        ar << GetValue();
    else
    {
        CString s;
        ar >> s;
        SetValue(s);
    }
}
```
If you want to add support for the user attribute page in the grid object and/or registry, you should also override the `GetValue()` and `SetValue()` methods. They convert your binary object into a string and back. See the Objective Grid Class Reference for `CGXAbstractUserAttribute` if you want to do this. There are additional overrides that allow you to fine-tune your derived user attribute. For example, `isEqual()` and `isEmpty()`.

To control the behavior of text that is too long to be easily visible in a cell, you can set an ellipsis ("...") using the attribute `CGXEllipseUserAttribute` with `GX_IDS_UA_ELLIPSISTYPE`.

For example, the following code sets a dotted ellipsis:

```cpp
SetStyleRange(CGXRange().SetRows(5, GetRowCount()),
              CGXStyle().SetUserAttributePtr(GX_IDS_UA_ELLIPSISTYPE,
                                          new CGXEllipseUserAttribute(gxDotEllipse)));
```

Three possible options are available:

- `gxNoEllipse`: No ellipsis, the default.
- `gxDotEllipse`: Dotted ellipsis ("...")
- `gxPoundEllipse`: Pound ellipsis ("####")

**Note:** Inserting an ellipsis at the beginning of text, i.e. “...text” is not supported.

### 5.2.7.1 Creating Custom Attributes

The `CGXStylesMap` class maintains a list of user attributes. If you want to extend the `CGXStyle` class with user defined attributes, you need to register them in the `CGXStylesMap` object.

To register user defined attributes, follow these steps:

1. Create a string resource (e.g. `IDS_MY_USERATTR`) using Visual Studio Resource View by opening the string table in the resource file and adding your custom string resource. Your `RC` file will change accordingly:

   ```cpp
   STRINGTABLE DISCARDABLE
   BEGIN
     IDS_UA_MYUSERATTR                "My user attribute"
   END
   // "My user attribute" will be displayed in the
   // "User"-page of the pre-built CGXStyleSheet dialog.
   ```

2. Call `AddUserAttribute()` with the resource ID and a style. This style is used for displaying the attribute’s value in the **User-defined attributes** grid in the `CGXStyleSheet` dialog. Each user-defined attribute is displayed as a row with the name and the attribute’s value. The end user can change the value through the `CGXStyleSheet` dialog. This step is not necessary if you do not want the user attribute exposed to the end user via the `CGXStyleSheet` dialog.

```cpp
void CMyGrid::OnInitialUpdate()
{
    // Attach parameter object to the grid
    SetParam(&GetDocument()->m_param, FALSE);
    // check for existence of stylesmap object
```
if (GetParam()->GetStylesMap() == NULL)
{
    // Create one
    CGXStylesMap* stylesmap = new CGXStylesMap;

    // standard styles
    stylesmap->CreateStandardStyles();

    // Additional base styles
    ...

    // user attributes
    // style for the cell in "User"-page
    // (user should be able to increase/decrease the value
    // with a spin control).
    CGXStyle styleSpin;
    
    styleSpin
        .SetControl(GX_IDS_CTRL_SPINEDIT)
        .SetWrapText(FALSE);

    // "My user attribute"
    stylesmap->AddUserAttribute(IDS_UA_MYUSERATTR, 
                                CGXStyle().SetWrapText(TRUE).SetAutoSize(TRUE));

    // load profile/registry settings
    stylesmap->SetSection("My base styles");
    stylesmap->ReadProfile();

    // attach object to parameter-object
    GetParam()->SetStylesMap(stylesmap);
}

3. A control can now interpret this attribute.

void CMyControl::OnClickedButton(CGXChild* pChild)
{
    NeedStyle();

    CString sValue = m_pStyle->
        GetUserAttribute(IDS_UA_MYUSERATTR);

    AfxMessageBox("My user attribute = " + sValue);

    CGXControl::OnClickedButton(pChild);
}

4. And you can use the user attribute with your cells:

   SetStyleRange(CGXRange(1,1),
                 CGXStyle()
                 .SetControl(IDS_CTRL_MYCONTROL);
                 .SetUserAttribute(IDS_UA_MYUSERATTR, "20");

   SetStyleRange(CGXRange(1,2),
                 CGXStyle()
                 .SetControl(IDS_CTRL_MYCONTROL);
                 .SetUserAttribute(IDS_UA_MYUSERATTR, "40");
5.3 Controls

5.3.1 Registration and Usage of Controls

You need to register new controls in the grid object (with \texttt{CGXGridCore::RegisterControl}) before using them as cell controls. You do this using a string resource identifier at run time. The name of the registered control will be displayed in the list box of the \texttt{Controls} page in the pre-built \texttt{CGXStyleSheet} dialog. It is not necessary to register existing Objective Grid controls. They are registered by the control factory as needed.

Follow these steps to register a control:

1. Add a string resource using Visual Studio Resource View by opening the string table in the resource file and adding your custom string resource. Your \texttt{.RC} - file will change accordingly:

   
   ```
   STRINGTABLE DISCARDABLE
   BEGIN
   IDS_CTRL_BITMAP        "Welcome bitmap"
   END
   ```

2. In \texttt{OnInitialUpdate()}, pass a new instance of the object as well as the string resource ID to \texttt{RegisterControl()}.

   ```
   void CGridSampleView::OnInitialUpdate()
   {
   ...

   RegisterControl(IDS_CTRL_BITMAP,
   new CGXBitmapButton(this, IDB_WELCOME));
   }
   ```

If you are registering a control derived from an existing Objective Grid control, you should check the control factory code for creation information. Some built-in controls simply need to be constructed while others require additional steps. The control creation code is in the \texttt{CGXControlFactory::CreateControl()} method in \texttt{gxfactry.cpp}.

// Snippet from gxfactry.cpp. This code demonstrates // the steps necessary when creating an instance of // CGXMaskControl

```
case GX_IDS_CTRL_MASKEDIT:
    CGXMaskControl* pEditCtrl =
        new CGXMaskControl(pGrid, GX_IDS_CTRL_MASKEDIT);
pEditCtrl->CreateControl();
pControl = pEditCtrl;
break;
```}

// In contrast, here's the code necessary to create an // instance of CGXEditControl

```
case GX_IDS_CTRL_EDIT:
    // default edit box
    pControl = new CGXEditControl(pGrid, GX_IDS_CTRL_EDIT);
    break;
```
3. Now, apply the control to any cell in the grid with `CGXStyle::SetControl()`.

```cpp
// Display the welcome bitmap
SetStyleRange(CGXRange(3, 2), CGXStyle().SetControl(IDS_CTRL_BITMAP));
```

## 5.4 Columns and Rows

### 5.4.1 Column and Row Count

The row and column count can be changed at run time using the following APIs:

- **CGXGridCore::SetColCount()**
  
  Specifies the number of columns in the grid. If this number is greater than the current number of columns, new columns will be added to the right-hand side of the grid. If the new count is less the current count, the extra columns will be removed from the right-hand side of the grid.

- **CGXGridCore::SetRowCount()**

  Specifies the number of rows in the grid. If this number is greater than the current number of rows, new rows will be added to the bottom of the grid. If the new count is less the current count, the extra rows will be removed from the bottom of the grid.

- **CGXGridCore::InsertCols()**

  Inserts columns before a specified anchor column.

  ```cpp
  InsertCols(8, 5);  // Insert 5 columns before column 8
  ```

  To add columns at the bottom of the grid, use the current column count + 1 for the column ID in the above call.

  ```cpp
  // Add 3 columns to right-hand side of grid
  InsertCols(GetColCount() + 1, 3);
  ```

- **CGXGridCore::InsertRows()**

  Inserts rows before a specified anchor row.

  ```cpp
  InsertRows(2, 4);  // Insert 4 rows before row 2
  ```

  To add rows to the bottom of the grid, use the current row count + 1 for the row ID in the above call.

  ```cpp
  // Add 2 rows to bottom of grid
  InsertRows(GetRowCount() + 1, 2);
  ```
CGXGridCore::RemoveCols()

Removes the specified range of columns.

RemoveCols(1, 4); // Remove columns 1 through 4
RemoveCols(6, 6); // Remove column 6

CGXGridCore::RemoveRows()

Removes the specified range of rows.

RemoveRows(5, 10); // Remove rows 5 through 10
RemoveRows(3, 3); // Remove row 3

5.4.2 Column and Row Sizing

5.4.2.1 Using Code to Size

Columns and rows may be sized programmatically using `CGXGridCore::SetColWidth()` or `CGXGridCore::SetRowHeight()`.

5.4.2.2 End User Resizing

The end user can resize a column or row by dragging the grid line separating two headers. The column to the left or the row above the grid line being dragged is resized. You can change the behavior for end user resizing using:

`CGXGridParam::EnableTrackColWidth()`

Specifies how columns can be resized.

`CGXGridParam::EnableTrackRowHeight()`

Specifies how rows can be resized.

Or turn user resizing off by calling

`CGXGridParam::EnableTrackColWidth(FALSE)`

or

`CGXGridParam::EnableTrackRowHeight(FALSE)`.

The end user can also size a column or row by double-clicking the adjacent grid line. Again, the column to the left or the row above will be resized. Double-clicking the grid line resets the column or row to the default size. The programmer can specify the default size using:

`CGXGridCore::SetDefaultColWidth()`

Specifies the default column width.
CGXGridCore::SetDefaultRowHeight()

Specifies the default row height.

5.4.3 Hidden Columns and Rows

5.4.3.1 Using Code to Hide

Hidden columns have zero width. Hidden rows have zero height. The programmer can hide a column by calling CGXGridCore::HideCols() or hide a row by calling CGXGridCore::HideRows().

Bit visibility flags indicating which columns or rows are hidden are set in the grid object. Visibility flags are stored with a call to StoreHideRow() for each row or StoreHideCol() for each column. The display is updated with UpdateChangedRowHeights() or UpdateChangedColWidths().

CGXGridCore::HideCols()

An overridable command for specifying the visible-flag for specific columns. This function can also be used to unhide a column.

CGXGridCore::HideRows()

An overridable command for specifying the visible-flag for specific rows. This function can also be used to unhide a row.

5.4.3.2 End User Hiding and Unhiding

The end user can hide a column or row by resizing the column or row to zero. (User resizing of rows and columns is discussed in Section 5.4.2.2.) Users can unhide hidden columns and rows by double-clicking the grid line where columns or rows are hidden. For example, if column 3 is hidden, the user could double-click the grid line between the column 2 header and the column 4 header. This will cause column 3 to be resized to the default size. Users can also resize the column or row (using the special cursors shown in Figure 43 and Figure 44) until the column or row is restored.

Figure 43 – Resizing a Hidden Column

<table>
<thead>
<tr>
<th>I</th>
<th>K</th>
</tr>
</thead>
</table>

Figure 44 – Resizing a Hidden Row

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
To prevent users from hiding columns, override `CGXGridCore::GetColWidth()` as in the following code:

```cpp
int CMyGrid::GetColWidth(ROWCOL nCol)
{
    int nRet = CBaseGrid::GetColWidth(nCol);

    // Check if stored col width is less than min col width
    if (nRet < MIN_COL_WIDTH)
    {
        nRet = MIN_COL_WIDTH;
    }

    return nRet;
}
```

With the same logic, you can use `GetRowHeight()` to prevent users from hiding rows.

To prevent the user from resizing a hidden row, override `OnTrackRowHeight()`. Check for `IsRowHidden()==TRUE` and return `FALSE` when this is the case.

## 5.4.4 Frozen Columns and Rows

**Frozen columns** are columns that will never scroll out of the window; they are always fixed at the left side of the grid. **Frozen rows** are rows that will never scroll out of the window; they are always fixed at the top of the grid.

`CGXGridCore::SetFrozenCols()`

An overridable command for specifying the number of frozen columns and columns to be used as row headers. **Row headers** are columns that will be used to display titles for rows. They typically have a 3D-look.

`CGXGridCore::GetFrozenCols()`

An overridable method that returns the number of frozen columns stored in the parameter-object.

`CGXGridCore::StoreFrozenCols()`

An overridable method that stores the number of frozen columns and columns to be used as row headers to the corresponding attribute of the parameter-object. You can override this method if you want to maintain the data in your own classes. The library accesses the data through `GetFrozenCols()` and `GetHeaderCols()`.

`CGXGridCore::UpdateFrozenCols()`

An overridable method that updates the window display after freezing columns. The method simply calls `Redraw()`.

The method creates the following hint:
CGXGridHint hint(gxHintUpdateFrozenCols, m_nViewID);
hint.nCol1 = nOldFrozenCols;
hint.nCol2 = nOldHeaderCols;
hint.flags = flags;

You can override this method if you need to change the hint.

CGXGridCore::SetFrozenRows()

An overridable command for specifying the number of frozen rows and rows to be used as column headers. **Column headers** are rows that will be used to display titles for columns. They typically have a 3D-look.

Numbering begins with zero. This means that if you set nFrozenRows = 0 and nHeaderRows = 0 there will still be one column header.

If you want to hide the first column header, you should call HideRows(0, 0).

This example shows you how to use two rows for column headers:

```cpp
// use 1 extra column as header (+ standard header at row 0)
SetFrozenRows(1, 1);

// Don’t draw column headers pressed when moving current cell
GetParam()->GetProperties()->SetMarkColHeader(FALSE);

// Some column headers shall span over several columns
SetCoveredCellsRowCol(0,2,0,4);

// This looks like this:
// |----|----------------|-----|-----|
// | A  | Group of Cols  | E   | F   |
// | 1  | 2   | 3   | |
// |----|----------------|-----|-----|
```

CGXGridCore::GetFrozenRows()

An overridable method that returns the number of frozen rows stored in the parameter-object.

CGXGridCore::StoreFrozenRows()

An overridable method that stores the number of frozen rows and rows to be used for column headers to the corresponding attribute of the parameter-object. The library accesses the data through `GetFrozenRows()` and `GetHeaderRows()`.

You can override this method if you want to maintain the data in your own classes.

CGXGridCore::UpdateFrozenRows()

An overridable method that updates the display window after freezing rows. The method simply calls `Redraw()`.
UpdateFrozenRows() creates the following hint:

```cpp
CGXGridHint hint(gxHintUpdateFrozenRows, m_nViewID);
hint.nRow1 = nOldFrozenRows;
hint.nRow2 = nOldHeaderRows;
hint.flags = flags;
```

You can override this method if you need to change the hint.

## 5.5 Cell and Window Coordinates

Cell and window coordinates are related to each other. It is possible to compute the cell coordinate from a given window point or to compute the window rectangle for a given cell. **Hit-Testing** is used for determining what is located under a given window coordinate.

### 5.5.1 Cell Coordinates

The user can scroll the data rows in the view. Column and row headers are fixed at the top and left margins of the window. Optionally, several rows or columns can be frozen so that they never scroll out of the view.

The grid maintains the number of frozen rows and columns and the cell coordinate of the top-left cell. The top-left cell is always located in the first scrollable row and column. All other cell coordinates can be derived from these parameters.

The grid component distinguishes between two coordinate systems:

- **Relative cell coordinates** are numbered from 0 to n in the current visible view.
- **Absolute cell coordinates** are unique identifiers for rows and columns, independent of the current visible view.

A characteristic of headers and frozen rows and columns is that both relative and absolute coordinates are the same value:

- Row headers are at column 0.
- Column headers are at row 0.
- Frozen rows are at row 1 to s, where s equals the number of frozen rows.
- Frozen columns are at column 1 to z, where z equals the number of frozen columns.
The following `CGXGridCore` methods can be used to determine or convert coordinates:

- `GetClientRow()`, `GetClientCol()`, `GetRow()`, `GetCol()`, `GetTopRow()`, `GetLeftCol()`, `GetFrozenRows()`, and `GetFrozenCols()`.

### 5.5.2 Window Coordinates

All column widths, row heights, and font heights are specified in pixels. A typical VGA screen has a resolution of 800x600 or 1024x768 pixels. A printer normally has a considerably higher resolution. To avoid generating much smaller output when printing versus drawing to the screen, Windows uses a technique called coordinate mapping. Objective Grid uses coordinate mapping extensively with its device context.

#### 5.5.2.1 Device Context

All drawing operations in Microsoft Windows are done through a device context that contains information about the coordinate mapping and drawing attributes, such as font and color. The grid component initializes the mapping mode with 72 pixels per inch for printer output. This is the same value used for screen output. Thus, the developer does not have to consider whether the grid is being displayed to a printer or to the screen when drawing. The coordinate mapping mode for the device context is initialized in `CGXGridCore::OnPrepareDC()`.

#### 5.5.2.2 Row Heights and Column Widths

Row heights are determined with the overridable method `CGXGridCore::GetRowHeight()`. Column widths are determined with `CGXGridCore::GetColWidth()`.

Row and column widths are stored internally as logical values. This makes it possible to adapt these values to a change of the zooming factor or standard font size. The following methods in the grid component perform these conversions: `CGXGridCore::Height_LPtoDP()` and `CGXGridCore::Width_LPtoDP()` convert logical values to pixel values. `CGXGridCore::Height_DPtoLP()` and `CGXGridCore::Width_DPtoLP()` convert pixel values to logical values.
Furthermore, `GetRowHeight()` and `GetColWidth()` call the overridable method `CGXGridCore::IsRowVisible()` or `CGXGridCore::IsColVisible()` to determine if the specified row or column is visible. In a splitter window, for example, the row headers of the right pane are not visible.

### 5.5.2.3 Converting Between Cell and Window Coordinates

It is possible to compute the window rectangle for a given cell because the height and width for each row and column is known. Conversely, it is also possible to determine the cell displayed at a given point.

The following `CGXGridCore` methods perform the necessary calculations:

- `CalcClientColFromPt()`
  
  Computes the column index for the column displayed at a given window point.

- `CalcClientRowFromPt()`
  
  Computes the row index for the row displayed at a given window point.

- `CalcRectFromRowCol()`
  
  Computes the window rectangle for the given range of cells.

- `CalcRectFromRowColEx()`
  
  Computes the window rectangle for the given range of cells, making sure that covered cells are not cut off.

- `CalcSumOfColWidths()`
  `CalcSumOfClientColWidths()`
  
  Add the width of several columns.

- `CalcSumOfRowHeights()`
  `CalcSumOfClientRowHeights()`
  
  Add the height of several rows.

- `GetGridRect()`
  
  Returns the window area where the grid is drawn. Normally, `GetGridRect()` determines drawing area by calling the MFC-method `GetClientRect()`. If the drawing area has been explicitly restricted with `SetGridRect()` or if the grid is in printing-mode, `GetGridRect()` returns an internally stored rectangle.

- `SetGridRect()`
  
  Restricts the drawing area to the specified rectangle.
5.5.3 Hit-Testing

Hit-testing is used to determine what is located under a given window coordinate. The method `CGXGridCore::HitTest()` computes a hit value for a given point.

The following hit values are defined:

- **GX_HORZLINE** - The point is on the gridline of a row header.
- **GX_VERTLINE** - The point is on the gridline of a column header.
- **GX_HEADERHIT** - The point is on the interior rectangle of a row or column header.
- **GX_CELLHIT** - The point is on the interior rectangle of a cell.
- **GX_SELEDGEHIT** - The point is on the edge of a selection. This hit value will only be generated if `m_bHitTestSelEdge` is TRUE.
- **GX_NOHIT** - No information is displayed at the point.

One use of `HitTest()` is for a context-sensitive mouse cursor. Whenever the mouse is moved, the grid calls `HitTest()` and changes the mouse cursor as necessary.

GetRowCount() and GetColCount() are used extensively to validate values within routines. In addition, because `HitTest()` is called repeatedly on every mouse move and because this method touches a lot of grid code, including `GetRowCount()`, it is not unusual for `GetRowCount()` to be called thousands of times in a short period of time. Because `GetRowCount()` and `GetColCount()` are called quite often, your overrides of these methods are not an appropriate place to do extensive calculation.
5.6 Grid Lines

5.6.1 Grid Line Drawing

Objective Grid provides two different ways for drawing grid lines. The new grid line drawing mode enables you to give the grid a more professional look. This new drawing mode lets you easily switch between solid, dotted, and dashed grid lines and allows you to remove the borders between the column and row headers.

If you want to use the new drawing mode as the default for all grids in your application, you should call GXSetNewGridLineMode() in the InitInstance() method of your application. If you want to use the new drawing mode only for individual grids, call GetParam()->SetGridLineStyle() during initialization for the grid.

BOOL CGridSampleApp::InitInstance()
{
    ...

    // Initialize the grid library and resources
    GXInit();

    // Use smarter grid lines (and dotted)
    GXSetNewGridLineMode(TRUE);

    ...
}

or:

BOOL CMyGridView::OnInitialUpdate()
{
    ...

    // Use smarter grid lines (and dotted)
    GetParam()->SetGridLineStyle();

    ...
}

Check out the description of CGXGridParam::SetNewGridLineMode in the Objective Grid Class Reference for a sophisticated discussion about this topic.
5.7  Events

5.7.1  Keyboard Events

Microsoft Windows sends keyboard events to the active window. MFC will call the message method `OnKeyDown()` for the active window. Objective Grid will call the method `CGXGridCore::ProcessKeys()`.

`ProcessKeys()` is called both from the current cell control and the grid itself, depending on active state of the current cell. `ProcessKeys()` needs to know which window is calling it. For this reason, the calling window passes the `this` pointer to `ProcessKeys()`.

If you want to process keyboard events, you should override `ProcessKeys()` in a derived grid class. The return value of `ProcessKeys()` specifies if the event has been processed.

The following example illustrates how to override `ProcessKeys()` to process the `F1` and `TAB` keys. If the user presses `TAB` at the bottom-right cell, a new row will be added to the table.

It is important to realize that `ProcessKeys()` is called from both `OnKeyDown` and `OnKeyUp` message handlers. You should keep this in mind when overriding `ProcessKeys()`, as it may be necessary to distinguish between `WM_KEYDOWN` and `WM_KEYUP` messages.

```cpp
BOOL CGSampleView::ProcessKeys(CWnd* pSender, UINT nMessage, UINT nChar, UINT nRepCnt, UINT flags)
{
    // check if user pressed <CTRL>-Key
    BOOL bCtl = GetKeyState(VK_CONTROL) & 0x8000;

    // check if user pressed <SHIFT>-Key
    BOOL bShift = GetKeyState(VK_SHIFT) & 0x8000;

    if (nMessage == WM_KEYDOWN)
    {
        switch (nChar)
        {
        case VK_HELP:            // user pressed <F1>-Key
            // display a message box with help
            if (bShift)
                { MessageBox("Help!");
                  return TRUE;
                }
        case VK_TAB:            // user pressed <TAB>-Key
            { ROWCOL nRow = 1, nCol = 1;
              GetCurrentCell(nRow, nCol);
              if (!bShift)
                  {
                    // Jump to the right cell
                    if (nCol < GetColCount())
                        MoveCurrentCell(GX_RIGHT);
                    else if (nRow < GetRowCount())
                        {

```
// if current cell is at last column,
// move down a row and to the first column.
if (MoveCurrentCell(GX_MOSTLEFT))
    MoveCurrentCell(GX_DOWN);
else
{
    // if current cell is at last row,
    // add a new row
    SetRowCount(GetRowCount()+1);
    // move to the first column in the new row
    if (MoveCurrentCell(GX_MOSTLEFT))
        MoveCurrentCell(GX_DOWN);
}
else
{
    // Jump to the left cell or move up a row
    if (nCol > 1)
        MoveCurrentCell(GX_LEFT);
    else if (nRow > 1)
    {
        if (MoveCurrentCell(GX_MOSTRIGHT))
            MoveCurrentCell(GX_UP);
    }
    else
    {
        MoveCurrentCell(GX_BOTTOMRIGHT);
    }
    return TRUE;
}

// call base class version
return CGXGridView::ProcessKeys(pSender,nMessage,nChar,nRepCnt,flags);

5.7.2 Keyboard Shortcuts

Table 4 lists keyboard shortcuts available in Objective Grid.

Table 4 – Keyboard shortcuts

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL+Z</td>
<td>Undo</td>
</tr>
<tr>
<td>CTRL+R</td>
<td>Redo</td>
</tr>
<tr>
<td>CTRL+C</td>
<td>Copy</td>
</tr>
<tr>
<td>CTRL+V</td>
<td>Paste</td>
</tr>
<tr>
<td>CTRL+X</td>
<td>Cut</td>
</tr>
</tbody>
</table>
5.8 Deriving a Grid

You should derive from the Objective Grid classes under the following circumstances:

- When you need to handle user interface events
- When you need to override some functionality

Say that you are using Objective Grid in a dialog. You would use **SubclassDlgItem** to place a grid window on the dialog. You could then make API calls such as

```c
m_WndGrid.SetValueRange(CGXRange().SetCols(1), _T("Test"));
```

to initialize the grid object. For this case, there is no need to derive from the grid classes.

On the other hand, say that you want to display a message box when the user clicks on a certain cell. There are no APIs that enable the programmer to do this. In order to display the message box, you have to override a grid virtual function **OnLButtonHitRowCol()**. This requires that you have a derived **CGXGridWnd** control.

You cannot use ClassWizard to add these virtual functions. The methods are defined in the Objective Grid library, but ClassWizard will not list them as overridable methods. They are nevertheless available. You can usually copy and paste the declaration for the overrides of your choice from the Objective Grid help files.

Another situation that warrants derivation is when you want to handle standard Windows messages.

Two steps are necessary to derive a class from a **CGXGridWnd**:

1. Create a new class with ClassWizard. Use “generic **CWnd**” as the Class Type.
2. Next, using a text editor, change the derivation for your window class, in both the **.h** and **.cpp** files. All references to **CWnd** should be replaced by **CGXGridWnd**.

If you want to derive a class from **CGXGridView**, use the following steps instead:

1. Create a new class with ClassWizard. Use "**CView**" as Class Type.
2. Next, using a text editor, change the derivation in your window class, in both the **.h** and **.cpp** files. All references to **CView** should be replaced by **CGXGridView**.
3. Change the generated **OnDraw()**, **OnBeginPrinting()**, and **OnEndPrinting()** methods to call the base-class **CGXGridView** version.
5.9  Virtual Grids

There are two ways to populate the grid.

1. Store the data inside the grid, using `CGXGridCore::SetStyleRange()` or `CGXGridCore::SetValueRange()`.

2. Dynamically bind or tie the grid to data such as a database, a live data feed, or an external data structure in your application.

The second method is known as Virtual Binding. When operating in virtual mode, the programmer can supply both the data and the grid dimensions dynamically. This section discusses in detail the architecture that enables virtual mode in the grid.

5.9.1  Supplying Data

Before the grid object can prepare itself for drawing, the dimensions of the grid must be known. In Objective Grid, those dimensions are supplied through `CGXGridCore::GetColCount()` and `CGXGridCore::GetRowCount()`. In a virtual grid, you will need to override one or both of these, depending on your implementation. The base class versions of these methods simply return the values stored in the grid object.

GetRowCount() and GetColCount() are used extensively to validate values within routines. In addition, because HitTest() is called repeatedly on every mouse move and because this method touches a lot of grid code, including GetRowCount(), it is not unusual for GetRowCount() to be called thousands of times in a short period of time. Because GetRowCount() and GetColCount() are called quite often, your overrides of these methods are not an appropriate place to do extensive calculation.

Once the dimensions are known, the grid object determines which cells need to be redrawn. For each cell that needs to be redrawn, the grid object composes its cell style. This is done by calling `CGXGridCore::ComposeStyleRowCol()`.

Recall from the styles architecture discussion that `ComposeStyleRowCol()` calls `GetStyleRowCol()` multiple times. The `CGXGridCore` implementation for `GetStyleRowCol()` simply returns the internally stored styles. To supply data or attributes at run time, the programmer can override `GetStyleRowCol()`.

```cpp
// Supply data (and other formats) from external // data source
BOOL CVirtGridView::GetStyleRowCol(ROWCOL nRow, ROWCOL nCol, CGXStyle& style, GXModifyType mt, int nType)
{
    BOOL bRet = CGXGridView::GetStyleRowCol(nRow, nCol, mt, nType);
    if(nType >= 0)
    {
        if(nCol != 0)
        {
            // Get value from external data source
            style.SetValue(GetValueFromExternalDataSource(nRow, nCol));
        }
    }
    return bRet;
}
```
// If negative no. then return
if(gxttof(GetValueFromExternalDataSource(nRow, nCol)) < 0)
    style.SetTextColor(RGB(255,0,0));

return bRet;

The first thing this method does is call the base class version. This is to populate the style parameter with any internally stored attributes or data. You should always give the grid object first crack at filling the style. If you set attributes in the style object and then call the base class, you run the risk of losing your original settings.

Next the method checks to see if nType is greater than or equal to zero. This is an important step. Remember that ComposeStyleRowCol() makes several calls to GetStyleRowCol(). One is for the cell-specific style (nType >= 0), and subsequent calls are for base styles (nType = -1). Using this check, you can supply cell-specific or base style attributes dynamically.

Inside the if statement, the value is set using:
style.SetValue(GetValueFromExternalDataSource(nRow, nCol));

GetValueFromExternalDataSource() is some arbitrary method that returns the value for a given row and column. It is not an Objective Grid method, only a function name made up for this discussion. Notice that SetValue() is called on the style object passed in the parameter list. This call sets the value in that object and not to any styles stored internally in the grid. This means that every time the style is composed, this override will fetch and set the data. No data is stored in the grid object.

On a side note, the style parameter is an instance of CGXStyle created externally to the GetStyleRowCol() method. It is passed by reference to GetStyleRowCol(). GetStyleRowCol() then populates the object using internally stored styles or virtually as we have done here. It is important to realize that this style object is temporary. Any changes made to it are not stored in the grid object, so it is not possible to “set-and-forget” style properties for a grid cell like you can for a non-virtual grid.

The second chunk of code in the if statement:

// If negative no. then return
if(gxttof(GetValueFromExternalDataSource(nRow, nCol)) < 0)
    style.SetTextColor(RGB(255,0,0));

sets the text color if the cell’s value is negative. This is an example of setting attributes virtually. It follows the same guidelines as virtual data.

The discussion thus far assumes that you only have plain data in your external data source and that attributes need to be set on a per-cell basis. You can also supply base style formatting by checking for nType = -1. For example:

if(nType >= 0)
{
    ...
}
else if(nType == -1)
{
    if(nCol != 0)
        // Get font for the column from external data source
5.9.2 Storing Data

Just as you can set attributes or data for a grid dynamically, you can retrieve attributes or data from a grid dynamically. The key method for this process is `StoreStyleRowCol()`. `StoreStyleRowCol()` will be called every time an attribute or value needs to be stored in the grid object. This includes user changes to the cell data. You can override `StoreStyleRowCol()` to divert those changes to your external data source. Here again, you should call the base class to allow the grid object to store those attributes that you are not storing externally.

```cpp
// Store the changes made in the grid into your external data source
BOOL CVirtGridView::StoreStyleRowCol(ROWCOL nRow, ROWCOL nCol, const CGXStyle * pStyle, GXModifyType mt, int nType)
{
    if(nType == -1)
    {
        return CGXGridView::StoreStyleRowCol(nRow, nCol, pStyle, mt, nType);
    }
    else if(nType >= 0)
    {
        if(nCol != 0 && pStyle->GetIncludeValue())
        {
            StoreValueIntoExternalDataSource(nRow, nCol, pStyle->GetValue);
            return TRUE;
        }
    }
    return FALSE;
}
```

Usually, the value is the only modifiable style attribute that you supply in `GetStyleRowCol()`. As demonstrated above, it is possible to supply other attributes dynamically. If you would like to store those attribute changes, you can do so in the `StoreStyleRowCol()` override.

```cpp
if(nCol != 0 && pStyle->GetIncludeValue())
{
    ...
}
else if (nCol != 0 && pStyle->GetIncludeFont())
{
    StoreFontIntoExternalDataSource(nRow, nCol, pStyle->GetFont());
    return TRUE;
}
```
5.10 Formula Support

Formula support will be enabled if you call the function `EnableFormulaEngine()` at the beginning of your `OnInitialUpdate()` routine or before you call `CGXGridWnd::Initialize()`. The `CGXGridCore::EnableFormulaEngine()` call will force the grid to replace the default `CGXData` object with a `CGXFormulaSheet` object. The `CGXFormulaSheet` object establishes a connection to the formula engine.

Furthermore, you might want to enable all the built-in worksheet functions that come with the formula engine. To do this, call `GXEnableWorksheetFunctions()` from your `InitInstance()` method.

See Chapter 14, “The Formula Engine,” for detailed information about the Objective Grid formula engine.

5.11 Undo and Redo

Objective Grid provides a powerful mechanism that lets the end user undo and redo a series of commands. Objective Grid maintains Undo information for all previous commands in an Undo list and Redo information for all previously undone commands in a Redo list. The number of commands or Undo stack size can be specified in the parameter object.

The end user can undo and redo commands with the Edit | Undo and Edit | Redo menu commands and their associated shortcuts. This mechanism is automatically provided by Objective Grid. No additional programming is required to enable this feature.

As you populate your grid, Objective Grid may be generating and maintaining Undo information which allows these actions to be undone by the user. Generating and maintaining this Undo information can be time consuming. If you do not want your user to be able to undo the work you are doing to populate the grid, then you should disable the grid Undo mechanism. To do so, call `GetParam()->EnableUndo(FALSE);` before the code you use to populate the grid. If you want the enable the grid’s Undo support after you grid has been populated, call `GetParam()->EnableUndo(TRUE);` after your grid has been populated.

5.11.1 CGXGridCore - Undo/Redo interface

The `CGXGridCore` class provides the following methods for undoing and redoing commands.

`BeginTrans()`

Marks the beginning of a transaction-like block of commands. This block will be undone, redone, and rolled back in one step.

`CommitTrans()`

Marks the end of a transaction

`Redo()`

Redoes the previously undone command.
Rollback()

Undoes all commands since the start of the transaction without generating Redo information.

Undo()

Undoes the previous command.

### 5.11.2 Transaction-Like Blocks of Commands

There are situations where you would like the user to be able to abort, undo or redo several changes in the grid with one Undo or Redo command. For example, if the user executed a “Replace All” command and later wants to undo this command, all resulting changes should be undone at once.

The key is calling `BeginTrans()` before the series of commands and `CommitTrans()` when the series is finished.

```cpp
void CGridSampleView::CommandBlock()
{
    BeginTrans( "Block of commands" );

    // These commands can only be undone all at once
    SetStyleRange(CGXRange(1,1), style1);
    SetStyleRange(CGXRange(1,2), style2);
    SetStyleRange(CGXRange(1,3), style3);

    CommitTrans();
}
```

### 5.11.3 Long Operations

In Windows programming, it is good practice to update the end user with the progress of particularly long commands. Additionally, you should give the end user the option to abort or even rollback the entire operation.

The `CGXLongOperation` class implements a mechanism exactly for this purpose and goes one step further. `CGXLongOperation` allows you to implement this mechanism in any method. It also ensures that when an operation is executed in a short amount of time, no further overhead is added. `CGXLongOperation` starts only after a delay after a specified amount of time has passed. If the operation completes before the wait time is over, no indication will be given to the end user.

`CGXLongOperation` also maintains an operation level as a static class variable. Every time you create a `CGXLongOperation` object in your code, the operation level will be increased. If the destructor for the object is called, the operation level will be decreased. This makes it possible to nest operations.

The following example explains how you can use `CGXLongOperation` in your application:

```cpp
void operation1()
{
```
// At first, create a CGXLongOperation object
CGXLongOperation theOp;

// Assign a status text. This text will be displayed only
// if the operation takes longer than a specific amount of
// ticks.
theOp.SetStatusText("Operation1...", FALSE);

// In many operations you have a loop. In this loop, you
// should call NeedMessages frequently. NeedMessages will
// return TRUE if the operation runs for a specific amount
// of time. Furthermore, you could add TRY/CATCH statements.
// They make it easy to cleanup when the user wants to abort
// the operation.
TRY
{
    BOOL bAbort = FALSE;

    while (bStatementsToProcess)
    {
        // Statements to be processed
        // ...

        // check, if user pressed ESC to cancel
        if (theOp.NeedMessages())
        {
            // if theOp.NeedMessages is TRUE the first time,
            // theOp.DoMessages will display a wait cursor
            theOp.SetPercentDone(nPercentDone);
            theOp.DoMessages(bAbort);

            if (bAbort)
                AfxThrowUserException();
        }
    }

    // operation executed successfully
    // cleanup
}
CATCH(CUserException, e)
{
    if (theOp.GetRollbackConfirmedState())
    {
        // user did select "Retry" in Abort-dialog box
        // So, try to undo already done changes and
        // cleanup
    }
    if (theOp.GetAbortConfirmedState())
    {
        // user did select "Abort" in the Abort-dialog box
        // So, abort the operation and
        // cleanup
    }
}
You can call this method from another method:

```cpp
void Execute()
{
    CGXLongOperation theOp;
    theOp.SetStatusText("Executing ...", FALSE);

    // The following call will lock the current operation level.
    // This means that when calling operation1(), this method
    // cannot change the status text. The call to
    // theOp.SetStatusText("Operation1...", FALSE);
    // will have no effect.

    theOp.SetLockedState(TRUE);

    TRY
    {
        BOOL bAbort = FALSE;

        while (bStatementsToProcess)
        {
            Operation1();
            // if user aborted Operation1(), also
            // this method will be aborted.

            // check, if user pressed ESC to cancel
            if (theOp.NeedMessages())
            {
                theOp.SetPercentDone(nPercentDone);
                theOp.DoMessages(bAbort);

                if (bAbort)
                    AfxThrowUserException();
            }
        }

        // operation executed successfully
        // cleanup
    }
    CATCH(CUserException, e)
    {
        if (theOp.GetRollbackConfirmedState())
        {
            // user did select "Retry" in Abort-dialog box
            // So, try to undo already done changes and
            // cleanup
        }
        if (theOp.GetAbortConfirmedState())
        {
            // user did select "Abort" in the Abort-dialog box
            // So, abort the operation and
            // cleanup
        }
    }
    END_CATCH
}
```
You can change the number of ticks (amount of time) necessary before `NeedMessages()` will return `TRUE` by calling the static `CGXLongOperation` member functions `SetTicksFirstTime()` and `SetTicksContinued()`.

If you pass `LONG_MAX` to `SetTicksFirstTime()`, the `CGXLongOperation` mechanism will be completely disabled.

## 5.12 Sorting

Objective Grid supports two types of sorting: a basic sort, and a more sensitive “natural” sort.

### 5.12.1 Basic Sorting

To support basic sorting when the user double-clicks the row header (col=0) or column header (row=0) use the following calls:

```cpp
GetParam()->SetSortColsOnDblClk(TRUE);  //sorting by ROWS
GetParam()->SetSortRowsOnDblClk(TRUE);  //sorting by COLUMNS
```

By default, enabling sorting sets `autodetect` as the sorting method. Autodetect examines the data in the row or column and sets the sort type to `numeric`, `alphanumeric`, or `datetime`.

For more information on this type of sorting, see the following entries in the *Objective Grid Class Reference*:

- `CGXGridCore::SortRows`
- `CGXGridCore::SortCols`
- `CGXGridCore::ImplementGridSort`

### 5.12.2 Natural Sorting

Natural sorting attempts to better reflect what a person would naturally expect the sort to do with respect to numeric values. For example, 10 miles, 1 kilometer, 2 feet, 100 inches, and 25 centimeters would be sorted as follows:

1 kilometer, 2 feet, 10 miles, 25 centimeters, 100 inches

Natural sorting takes place in your application’s event handler, or when double-clicking the column’s header if sorting of columns has been previously turned on and natural sorting specified. Natural sorting must be specified explicitly; that is, it is never set by autodetect.

```cpp
// Add a sort info array in your CGXGridView-derived view class
CGXSortInfoArray m_sortInfo;

// Specify the natural sort type in your view class or event handler
m_sortInfo[0].sortType = CGXSortInfo::natural;
SortCols(CGXRange().SetTable(), m_sortInfo); // sort columns
SortRows(CGXRange().SetTable(), m_sortInfo); // sort rows
```
5.12.3 Sorting Example

Examples of both the basic and natural sorting implementations are demonstrated on the Sorting tab in the sample \<stingray_installdir>\Samples\Grid\General\GridApp. To test natural sorting in the GridApp sample:

1. Start the sample and go to the Sorting tab.
2. Click on any cell in columns B or C of the grid.
3. To use natural sorting on the selected column, from the Format | Sort Columns menu item click on the Natural option.

   The column with focus should sort “naturally”.

Double-clicking either the column B or C header row also sorts the column, but autodetect sets a different sort type (numeric, alphanumeric, or datetime) rather than natural.

5.13 Pre-Built Dialogs for Accessing Objective Grid Objects

Each section below contains a brief description of the pre-built dialogs contained in Objective Grid, a detailed breakdown of each component on the dialog, and a list of associated APIs. If you would like more information about their use, please consult the Objective Grid Class Reference. Dialog item resource IDs can obtained by opening the Objective Grid resource file gxres.rc and browsing the dialog resources.

5.13.1 Display Settings Dialog

This pre-built dialog allows the end user to modify certain attributes of the CGXProperties object associated with the grid object at run time.
The Display Settings Dialog contains the following features:

- **Titles and Grid Lines.** This is a set of check boxes that allow the user to customize the column/row headers and grid lines.

Table 5 – Customizing Column/Row Headers and Grid Lines

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D-Buttons</td>
<td>Specifies whether the column/row headers are displayed as buttons (3D with raised and lowered effects).</td>
</tr>
<tr>
<td>Vertical Lines</td>
<td>Specifies whether the vertical grid lines are displayed.</td>
</tr>
<tr>
<td>Horizontal Lines</td>
<td>Specifies whether the horizontal grid lines are displayed.</td>
</tr>
<tr>
<td>Mark Current Row</td>
<td>Specifies whether the current cell's row should be marked visually in the row header. (Has no effect if 3D-Buttons is not checked.)</td>
</tr>
<tr>
<td>Mark Current Column</td>
<td>Specifies whether the current cell's column should be marked visually in the column header. (Has no effect if 3D-Buttons is not checked.)</td>
</tr>
</tbody>
</table>

The following APIs are relevant for titles and grid lines:

- `CGXProperties::GetDisplay3dButtons()`
- `CGXProperties::SetDisplay3dButtons()`
- `CGXProperties::GetDisplayVertLines()`
Preview. This is a preview grid that allows the user to see the changes immediately, before actually applying them to the grid.

Color. This is a pair of controls that allow the user to customize the grid’s colors. On the left is a list box containing several color properties and on the right is a color box containing color choices.

Table 6 – Customizing the Grid’s Colors

<table>
<thead>
<tr>
<th>Grid Lines</th>
<th>Specifies the grid lines’ color.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Lines</td>
<td>Specifies the color of the line that separates frozen columns/rows and non-frozen columns/rows.</td>
</tr>
<tr>
<td>Tracking Line</td>
<td>Specifies the color of the outline when columns/rows are being resized.</td>
</tr>
<tr>
<td>Dragging Line</td>
<td>Specifies the color of the line where a dragged column/row will be dropped.</td>
</tr>
<tr>
<td>Background</td>
<td>Specifies the color used to draw the “gray area” outside of the grid.</td>
</tr>
</tbody>
</table>

New color properties may be added using `CGXProperties::AddColorProperty()`.

Use the following APIs for color:

- `CGXProperties::GetColor()`
- `CGXProperties::SetColor()`
- `CGXProperties::AddColorProperty()`

User Properties. This is a grid control that allows the user to change user properties.

New user properties may be added using `CGXProperties::AddUserProperty()`.

Use the following APIs for User Properties:

- `CGXProperties::GetUserProperty()`
- `CGXProperties::SetUserProperty()`
- `CGXProperties::AddUserProperty()`
*Settings to Profile.* This is a check box that allows the user to save the current dialog settings as the default values.

When the user has checked this check box, a boolean variable, whose address was passed during dialog class construction, will be set to `true`. The developer should check this variable to determine if the profile should be saved.

### 5.13.2 Header / Footer Dialog

This pre-built dialog allows the end user to specify page header and page footer text.

**Figure 47 – Header and Footer Dialog**

- **Header / Footer.** This is a tabbed grid control that contains 3 columns and 10 rows on each of 2 tabs. Its purpose is to provide an interface for entering page header and page footer text.

  The column headers, labeled Left Aligned, Centered, and Right Aligned, indicate the header alignment of the text in the cells below them. All of the text contained in the **Left Aligned** column (with the exception of the column header) will be left justified in the header/footer. The **Centered** column will be centered horizontally and **Right Aligned** will be right justified.

  There are 10 rows used for entering header/footer text, providing the user with a mechanism to mix fonts within the header/footer. The actual text printed to the page is pieced together by appending each of the cells to the previous. In other words, the **Left Aligned** header text is determined as follows: \((1,1) + (2,1) + (3,1) + (4,1) + (5,1) + (6,1) + (7,1) + (8,1) + (9,1) + (10,1)\), where \((x,y)\) indicates the text contained in that cell and \(+\) indicates string concatenation.
The font can be changed by selecting a cell or range of cells, and then clicking the font button. A standard font dialog will be displayed and the user can make the necessary changes. When the font dialog is closed via its OK button, the changes will be applied back to the selected cells.

In addition to text, the grid will also accept the following predefined tokens (or escape sequences):

- **$F** – Document file name
- **$A** – Application name
- **$P** – Current page number
- **$N** – Total number of pages
- **$D** – Current date (See `GetDataHeader()` in the Objective Grid Class Reference for additional date formatting information.)
- **$R** – Register / Tabsheet name (if grid is used in a workbook)

Existing tokens may be modified or new tokens may be defined using the following `CGXProperties` methods: `AddToken()`, `SubstTokenText()`, and `GetTokenArgs()`.

Use the following APIs for Header / Footer:

- `CGXProperties::GetDataHeader()`
- `CGXProperties::GetDataFooter()`
- `CGXProperties::AddToken()`
- `CGXProperties::SubstTokenText()`
- `CGXProperties::GetTokenArgs()`
- `CGXGridCore::OnPrintHeaderAndFooter()`

**Distance to Frame.** This is a set of edit controls that specify the vertical space between the header/footer and the top/bottom margin. The value may be entered as inches or centimeters by appending **in** or **cm** to the end of the value. The default, if no units are entered or the unit is not recognized, is inches.

Use the following APIs for Distance to Frame:

- `CGXProperties::GetDistances()`
- `CGXProperties::SetDistances()`

**Page Numbering.** This is an edit control that allows the user to specify the starting page number. All subsequent pages are numbered according to this value.

Use the following APIs for Page Numbering:

- `CGXProperties::GetFirstPage()`
- `CGXProperties::SetFirstPage()`

**Save Settings to Profile.** This is a check box that allows the user to save the current dialog settings as the default values.

The check box’s resource ID is `GX_IDC_PCHECK_SAVEDEFAULT`.
When the user has checked this check box, a boolean variable, whose address was passed during dialog class construction, will be set to \texttt{TRUE}. The developer should check this variable to determine if the profile should be saved.

### 5.13.3 Page Setup Dialog

This pre-built dialog allows the end user to specify printing specific settings for Objective Grid.

**Figure 48 – Page Setup Dialog**

![Page Setup Dialog](image)

See Table 7 for a description of the features in the Page Setup dialog.

**Table 7 – Page setup dialog**

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Relevant APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Margins</strong></td>
<td>Editing controls used to specify the print margins.</td>
<td>CGXProperties::GetMargins()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CGXProperties::SetMargins()</td>
</tr>
<tr>
<td><strong>Preview</strong></td>
<td>Preview grid that allows the user to see the changes in real time, before applying them back to the grid.</td>
<td>N/A</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
<td>Relevant APIs</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Titles and Grid Lines</td>
<td>Check boxes that allow the user to customize the column/row headers and</td>
<td>CGXProperties::GetPrintRowHeaders()</td>
</tr>
<tr>
<td></td>
<td>gridlines <strong>Row Headers</strong>. Specifies if row headers should be printed. **Column</td>
<td>CGXProperties::SetPrintRowHeaders()</td>
</tr>
<tr>
<td></td>
<td>Headers**. Specifies if column headers should be printed. <strong>Print Frame</strong>.</td>
<td>CGXProperties::GetPrintColHeaders()</td>
</tr>
<tr>
<td></td>
<td>Specifies if a frame should be drawn around the grid when printed. **Vertical</td>
<td>CGXProperties::SetPrintColHeaders()</td>
</tr>
<tr>
<td></td>
<td>Lines**. Specifies if vertical grid lines should be printed. **Horizontal</td>
<td>CGXProperties::GetPrintFrame()</td>
</tr>
<tr>
<td></td>
<td>Lines**. Specifies if horizontal grid lines should be printed. **Only Black</td>
<td>CGXProperties::SetPrintVertLines()</td>
</tr>
<tr>
<td></td>
<td>and White**. Specifies if the grid should be printed only with black and</td>
<td>CGXProperties::GetPrintHorzLines()</td>
</tr>
<tr>
<td></td>
<td>white colors.</td>
<td>CGXProperties::SetPrintBlackWhite()</td>
</tr>
<tr>
<td>Page Order</td>
<td>Radio buttons used to specify the page order when printing a grid across</td>
<td>CGXProperties::GetPageOrder()</td>
</tr>
<tr>
<td></td>
<td>multiple pages.</td>
<td>CGXProperties::SetPageOrder()</td>
</tr>
<tr>
<td>Center on Page</td>
<td>Check boxes used to specify vertical and horizontal centering when printing</td>
<td>CGXProperties::GetCenterVertical()</td>
</tr>
<tr>
<td></td>
<td>a grid.</td>
<td>CGXProperties::SetCenterVertical()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CGXProperties::GetCenterHorizontal()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CGXProperties::SetCenterHorizontal()</td>
</tr>
<tr>
<td>Save Settings to Profile</td>
<td>Check box that allows the user to save the current dialog settings as the</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>default values. When the user has checked this check box, a boolean variable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>whose address was passed during dialog class construction, will be set to</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>TRUE</strong>. The developer should check this variable to determine if the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>profile should be saved.</td>
<td></td>
</tr>
</tbody>
</table>

Table 7 – Page setup dialog (Continued)
5.14 Number Formatting

5.14.1 Value Types

CGXStyle::SetValueType() allows you to specify the value type in a cell. Possible types are:

- GX_VT_STRING
- GX_VT_NUMERIC
- GX_VT_EXPRESSION
- GX_VT_ERROR

CGXStyle::GetValueType() returns the value type of the cell.

SetValueType() does not affect the way values are stored in CGXStyle. All values are stored as a string. SetValueType() only offers additional information about the value that is stored in the cell. For example, when you apply numeric formatting to a cell, it will only affect the cell if its value type is GX_VT_NUMERIC.

The value type information is very important when you pass style objects to the formula engine. The formula engine parses values to determine if they should be stored as strings or as numbers. It is very important for the formula engine to distinguish between numbers and strings.

```
// Sets the value type to GX_VT_NUMERIC
SetValueRange(CGXRange(nRow, nCol), 122.22);

// Sets the value type to GX_VT_STRING
SetValueRange(CGXRange(nRow, nCol), "122.22");

// Sets the value type to GX_VT_STRING
SetStyleRange(CGXRange(nRow, nCol), CGXStyle().SetValue("122.22");

// Sets the value type to GX_VT_NUMERIC
SetStyleRange(CGXRange(nRow, nCol), CGXStyle().SetValue(929.2));

// SetExpressionRowCol parses the string and sets the value type to GX_VT_NUMERIC
SetExpressionRowCol(nRow, nCol, "1222.22");

// SetExpressionRowCol parses the string and sets the value type to GX_VT_STRING
SetExpressionRowCol(nRow, nCol, "ABDGDDG1222.22");
```

5.14.2 Formatting Number Values for Static Cells

The following methods pertain to number formatting:

- CGXStyle::SetFormat()
- CGXStyle::SetPlaces()
These methods let you format number values for static cells and specify the precision of the value (the number of significant digits).

Objective Grid lets you format number values for inactive cells and specify the precision when appropriate. The formatted value is only displayed for static (inactive) cells. If you start editing a cell, the original value is displayed.

For example, if you store the value 122.2345 in a cell and then set the format to be fixed with two decimals, 122.23 is displayed in the cell. When you click the cell make the cell active, the cell value changes to 122.2345. Once you leave the cell the cell is made inactive again, the display text reverts to 122.23.

When you apply numeric formatting to a cell, it only affects the cell if the value type is `GX_VT_NUMERIC`.

Objective Grid supports the following formats. For certain format types you can also specify the number of places (referred to as ‘N’ in Table 8).

**Table 8 – Format Types**

<table>
<thead>
<tr>
<th>Format ID</th>
<th>Description</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>GX_FMT_FLOAT</td>
<td>Scientific</td>
<td>Displays the number in scientific notation (exponent form) with N significant digits.</td>
</tr>
<tr>
<td>GX_FMT_FIXED</td>
<td>Fixed</td>
<td>Displays the number using a fixed number of decimal places, specified by N.</td>
</tr>
<tr>
<td>GX_FMT_FIXED_PARENS</td>
<td>Fixed</td>
<td>Displays the number using a fixed number of decimal places, specified by N. Uses parentheses to indicate negative numbers.</td>
</tr>
<tr>
<td>GX_FMT_NUMLOCALE</td>
<td>Locale</td>
<td>Displays number using locale from regional settings (, instead of . for example).</td>
</tr>
<tr>
<td>GX_FMT_GEN</td>
<td>General</td>
<td>Displays the number in fixed format or scientific notation, whichever fits. Trailing zeros are not displayed.</td>
</tr>
<tr>
<td>GX_FMT_DOLLARS</td>
<td>Dollars</td>
<td>Displays the number with a leading dollar sign ($) and with comma delimiters, as in $1,000,000. Negative values are displayed in parentheses.</td>
</tr>
<tr>
<td>GX_FMT_COMMA</td>
<td>Comma</td>
<td>Displays the number with comma delimiters, as in 1,000,000. Negative values are displayed in parentheses.</td>
</tr>
<tr>
<td>GX_FMT_HEX</td>
<td>Hex</td>
<td>Display the number in hexadecimal notation. For example, the value 31 is displayed as 1F.</td>
</tr>
<tr>
<td>GX_FMT_LOGIC</td>
<td>Logic</td>
<td>Displays the value 0 as 0, the value 1 as 1, and all other values as ?.</td>
</tr>
<tr>
<td>GX_FMT_DD-MMM-YY</td>
<td>DD-MMM-YY</td>
<td>Displays the integer portion of a date/time value as a Gregorian date, in the format 01-Aug-99.</td>
</tr>
<tr>
<td>GX_FMT_DD-MMM</td>
<td>DD-MMM</td>
<td>Displays the integer portion of a date/time value in the format 01-Aug.</td>
</tr>
</tbody>
</table>
Use CGXStyle::SetFormat() to apply number formatting to cells.

```cpp
SetExpressionRowCol(nRow, 1, _T("General"));
SetExpressionRowCol(nRow+1, 1, _T("7777.77"));
SetStyleRange(CGXRange(nRow+1, 1),
    CGXStyle().SetFormat(GX_FMT_GEN).SetPlaces(15));

SetExpressionRowCol(nRow, 2, _T("Fixed"));
SetExpressionRowCol(nRow+1, 2, _T("7777.77"));
SetStyleRange(CGXRange(nRow+1, 2),
    CGXStyle().SetFormat(GX_FMT_FIXED).SetPlaces(4));

SetExpressionRowCol(nRow, 3, _T("Scientific"));
SetExpressionRowCol(nRow+1, 3, _T("7777.77"));
SetStyleRange(CGXRange(nRow+1, 3),
    CGXStyle().SetFormat(GX_FMT_FLOAT).SetPlaces(4));

SetExpressionRowCol(nRow, 4, _T("Dollars"));
SetExpressionRowCol(nRow+1, 4, _T("7777.77"));
SetStyleRange(CGXRange(nRow+1, 4),
    CGXStyle().SetFormat(GX_FMT_DOLLARS).SetPlaces(2));
```

---

**Table 8 – Format Types (Continued)**

<table>
<thead>
<tr>
<th>Format ID</th>
<th>Description</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>GX_FMT_MONTH_YEAR</td>
<td>MMM-YY</td>
<td>Displays the integer portion of a date/time value in the format Aug-99.</td>
</tr>
<tr>
<td>GX_FMT_DATE</td>
<td>MM/DD/YY</td>
<td>Displays the integer portion of a date/time value in the format 08/01/99.</td>
</tr>
<tr>
<td>GX_FMT_HIDDEN</td>
<td>Hidden</td>
<td>The cell contents are not displayed.</td>
</tr>
<tr>
<td>GX_FMT_TIME</td>
<td>HH:MM:SS</td>
<td>Displays the fractional portion of a date/time value in the format 06:15:30.</td>
</tr>
<tr>
<td>GX_FMT_PERCENT</td>
<td>Percent</td>
<td>Displays the number as a percentage, multiplying it by 100. For example, the value 0.1 is displayed as 10.00%.</td>
</tr>
<tr>
<td>GX_FMT_TEXT</td>
<td>Text</td>
<td>For cells that contain formulas, the formula itself is displayed rather than the computed value of the cell.</td>
</tr>
<tr>
<td>GX_FMT_INTL_DATE</td>
<td>DD.MM.YYYY</td>
<td>Displays the integer portion of a date/time value in the format 01.08.1999.</td>
</tr>
<tr>
<td>GX_FMT_ISO8061_DATE</td>
<td>YYYY-MM-DD</td>
<td>Displays the integer portion of a date/time value in the ISO 8061 date format 1999-08-01.</td>
</tr>
<tr>
<td>GX_FMT_DATETIME</td>
<td>MM/DD/YY HH:MM</td>
<td>Displays the date and time according to the system settings.</td>
</tr>
<tr>
<td>GX_FMT_USER1</td>
<td>User1</td>
<td>If you override GXFormatText you might use these format codes:</td>
</tr>
<tr>
<td>GX_FMT_USER2</td>
<td>User2</td>
<td>(GX_FMT_USER1, GX_FMT_USER2)</td>
</tr>
<tr>
<td>GX_FMT_USER3</td>
<td>User3</td>
<td>GX_FMT_USER3, GX_FMT_USER4)</td>
</tr>
<tr>
<td>GX_FMT_USER4</td>
<td>User4</td>
<td>for your own currency formatting (e.g. DM, Lira, ...).</td>
</tr>
</tbody>
</table>
5.14.3 Recognizing Locale Settings in Static Formatting

This applies only to non formula grids.

By default, Objective Grid does not use regional settings when you choose to format numbers. So, it cannot understand that 1.345,13 is a number, and format it properly, even if your system’s regional settings are set so that period is the thousands separator and comma is the decimal separator. To get Objective Grid to use your regional settings, add a call to CGXGridCore::EnableLocaleNumbers() in your grid’s Initialize() or OnInitUpdate() call. After that, any cell using a CGXEditControl that has been formatted using one of the codes listed below will use the numeric locale settings. In addition, with this setting enabled, Objective Grid will recognize values typed into numeric formatted cells as being numerical values.

No matter what the locale settings, the actual numerical values stored in the grid’s data object use a period as the decimal separator and do not contain any thousands separator. This convention makes the grid’s serialization locale independent. Numbers serialized from a grid using Italian settings can be read into a grid using English settings without losing the numerical values.

Also, for historical reasons, Objective Grid catches system changes through handling the WM_WININICHANGE message. If your code implements a handler for the newer WM_SETTINGCHANGE message, make sure you return a FALSE in your message handler so the grid’s handler will also be called.

To customize this behavior, derive your control from CGXEditControl and override one of these three virtual CGXEditControl methods.

virtual BOOL StripSeparatorFromValue(CString& sValue, const CGXStyle* pStyle, double& d);
// called from Store & ConvertControlTextToValue (handles Paste)
// to strip group separator from the value before it is stored

virtual BOOL PutDecInValue(CString& sValue, const CGXStyle* pStyle);
//called from Init to put correct dec sep in windowtext

virtual BOOL AcceptCommasInNumbers(const CGXStyle* pStyle);
// returns whether strings with thousands separator should be
// treated as a number. Default behavior checks for one of these
// number formats:
  // GX_FMT_FIXED
  // GX_FMT_COMMA
  // GX_FMT_DOLLARS
  // GX_FMT_FLOAT
By default in a non-formula grid, the code below will display 1234.5555 as 1,234.56 no matter what your regional settings are. But if you add a call to `EnableLocaleNumbers()` in your initialization code, the formatting will reflect your regional settings. For example, using Italian locale settings would display the value 1234.5555 as 1.234,56.

```c
SetStyleRange(CGXRange(1,1,1,2), CGXStyle()
    .SetFormat(GX_FMT_COMMA)
    .SetValueType(GX_VT_NUMERIC)
    .SetPlaces(2)
    .SetValue((double) 1234.5555)
);```

5.15 Miscellaneous

5.15.1 Using CGXDrawingAndFormatting

`CGXDrawingAndFormatting` encapsulates some general methods used for drawing and formatting text in Objective Grid. In earlier versions, these methods were global methods. The disadvantage was that there was no possibility for you to customize these methods without modifying the source code.

The `CGXDrawingAndFormatting` class gives you the ability to override the default behavior of these methods.

If you have derived a class from `CGXDrawingAndFormatting` you can force Objective Grid to use your customized methods by calling `SetDaFTools()`.

```cpp
class CMyDrawingTools: public CGXDrawingAndFormatting
{
public:
    CMyDrawingTools();

    virtual CString GXFormatText(const CGXStyle& style, LPCTSTR pszValue = NULL, unsigned nValueType = GX_VT_STRING, CGXGridCore* pGrid = NULL, CGXControl* pControl = NULL);
};
```

// in your InitInstance method

```cpp
GXInit();
GXSetDaFTools(new CMyDrawingTools);
```

where `CMyDrawingTools` must be derived from `CGXDrawingAndFormatting`.

// Text routines
```cpp
virtual int GXDrawTextLikeMultiLineEdit(CDC* pDC, LPCTSTR lpszString, int nCount, CRect rect, UINT nFormat, LPRECT lpRectClip = NULL);

virtual int GXDrawRotatedText(CDC* pDC, LPCTSTR lpszString, int nCount, CRect rect, UINT nFormat, LONG lfOrientation, LPRECT lpRectClip = NULL);

virtual int GXGetMultiLineTextPosUnderPt(CDC* pDC, LPCTSTR lpszString, int nCount, CRect rect, UINT nFormat, CPoint pt);

virtual int GXGetMultiLineTextBreakCount(CDC* pDC, LPCTSTR lpszString, int nCount, CRect rc, UINT nFormat);
```
virtual int GXComputeLineBreaks(CDC* pDC, LPCTSTR sz, int nCount,
    LPRECT rc, UINT nFormat,
    int* LineBreaks, int* LineLengths,
    int nMaxBreaks);

// Text formatting and parsing
virtual CString GXFormatText(const CGXStyle& style,
    LPCTSTR pszValue = NULL,
    unsigned nValueType = GX_VT_STRING,
    CGXGridCore* pGrid = NULL,
    CGXControl* pControl = NULL);

virtual BOOL GXDeval(LPCTSTR s, double *d,
    unsigned* pImportFormat = NULL,
    unsigned* pImportPlaces = NULL);

See “Macros And Globals” in the Objective Grid Class Reference for a description of these methods.

## 5.15.2 IntelliMouse Support

To enable IntelliMouse Support for Objective Grid, you should add a call to EnableIntelliMouse() before calling OnGridInitialUpdate() or Initialize().

The IntelliMouse support in Objective Grid is very similar to the IntelliMouse support in Microsoft Excel and Microsoft Internet Explorer. Objective Grid supports the following features:

- **Scrolling by rolling the mouse wheel**
- **Scroll horizontally by clicking SHIFT and rolling the mouse wheel**
- **Zoom in and out by clicking CTRL and rolling the mouse wheel**
- **Auto scrolling by clicking the mouse wheel button and dragging the mouse up, down, to the left or right.**
- **ClickLock for the mouse wheel button:** Just click and hold down the mouse wheel button for a moment and your click is locked. With ClickLock you can scroll the grid very easy by simply dragging the mouse (just as with Autoscroll but you don’t have to hold the mouse wheel button down). Click again to release ClickLock.

```cpp
cvoid CMyGridView::OnInitialUpdate()
{
    CGXGridView::OnInitialUpdate();
    EnableIntelliMouse();
}

BOOL CSample4Dialog::OnInitDialog()
{
    CDialog::OnInitDialog();
    GetGridWnd()->EnableFormulaEngine();
    GetGridWnd()->Initialize();
    GetGridWnd()->SetRowCount(25);
    GetGridWnd()->SetColCount(52);
}
GetGridWnd()->SetCurrentCell(1,1);
GetGridWnd()->EnableIntelliMouse();
GetGridWnd()->SetFocus();
return FALSE;  // return TRUE unless you set the
    // focus to a control

5.15.3 Rotated Text in Cells

To display rotated text in a cell, use CGXFont::SetOrientation() and specify the angle.

SetCoveredCellsRowCol(400, 1, 402, 1);
SetStyleRange(CGXRange(400, 1),
    CGXStyle()
       .SetFont(CGXFont()
           .SetSize(12)
           .SetBold(TRUE)
           .SetOrientation(-900)
        )
       .SetValue(_T("TEST"))
    );

Please note that only single lines are support for rotated text. Objective Grid does not support word-break with rotated text.

---

It is possible to add support for rotated text in the font page by adding a combo box with style set to “Simple” to the IDD_DIALOG_FONTPAGE template. When you specify IDC_GXCBORIENTATION as the resource ID for this combo box, the font page will allow the user to specify the font orientation via the combo box.

5.15.4 Character Sets

Objective Grid has support for different character sets. You can assign a specific character set to a cell via SetFaceName().

SetFaceName("Arial");            // Default charset will be used
SetFaceName("Arial (Western)");  // Western charset will be used
SetFaceName("Arial (Cyrillic)"); // Cyrillic charset will be used

You can also specify if the character set should be displayed in the font page of the stylesheet by setting

        GXGetAppData()->m_bDisplayFontCharset = TRUE;

The default is FALSE (character sets will not be displayed in the font page).

Enabling Japanese in ANSI-DBCS Grid build types using the Japanese locale may require following workaround:

        LOGFONT logFont;
        SystemParametersInfo(SPI_GETICONTITLELOGFONT,sizeof(logFont),&logFont,0);
        CGXFont gxFont(logFont);
5.15.5 Direct Copying and Moving of Cells

The CGXGridCore: :CopyCells() and CGXGridCore: :MoveCells() APIs lets you easily copy or move a given range of cells to a new destination.

You can also force CGXGridCore to use these new commands for doing Copy/Paste and OLE drag & drop commands within a worksheet by setting the m_bDirectCopyPaste and m_bDirectCutPaste attributes in the CGXGridParam object.

- m_bDirectCopyPaste — Set this to TRUE if data should be copied within the grid with the CopyCells() method, and not using the clipboard. Set it to FALSE if conventional clipboard copy/paste should be used.

- m_bDirectCutPaste — Set this to TRUE if data should be moved within the grid with the MoveCells() method, and not using the clipboard. When cutting cells, cells will only be marked as cut, and are cleared only when the user executes a Paste command. If m_bDirectCutPaste is FALSE, the conventional clipboard cut/paste will be used.

- m_bDirectDragDrop — Set this to TRUE if data should be copied or moved with CopyCells() or MoveCells(), or to FALSE if conventional clipboard cut/copy/paste should be used.

When using direct cut and paste, cell values will not be cleared until you paste data into the same worksheet. When you select cells and then select the menu item Edit|Cut, the selected cells will be marked with a gray background. This indicates that the cells will be cleared once the paste operation is completed.

5.15.6 AutoScrolling

When you click on a scrollbar or if you press an arrow key, scrolling will accelerate incrementally. Also, when dragging the mouse out of the grid, autoscrolling will accelerate incrementally. The user can slow down the scrolling speed by moving the mouse nearer to the grid or speed up the scrolling by moving the mouse farther away from the grid.

- Scrolling when the user drags the mouse out of the grid (with the left button pressed) is controlled with SetAutoScroll().

- Acceleration of the scrolling when the user holds down an arrow key is controlled with SetAccelArrowKey().

- Acceleration of the scrolling when the user presses a scrollbar button is controlled with SetAccelScrollbars().
5.15.7 OLE Drag and Drop Integration

To add OLE Drag and Drop support to your grid, simply:

1. Add a call to `EnableOleDropTarget()` and a call to `EnableOleDataSource()` during initialization for your grid object (usually in your override of `CDialog::OnInitDialog()` or `CView::OnInitialUpdate()`).

2. If you want to customize the OLE Drag and Drop behavior, you can pass a combination of flags as the parameter.

   ```cpp
   EnableOleDropTarget(GX_DNDEGDESCROLL | GX_DNDAUTOSCROLL |
                      GX_DNDTEXT    | GX_DNDSTYLES);
   ```

3. Add a call to `AfxOleInit()` in your project’s override of `CWinApp::InitInstance()`.

You can still continue to use the traditional MFC approach by embedding a `CGXGridDropTarget` as member variable into your grid or parent dialog. The MFC approach is still supported (for the time being), but we recommend that you use the newer approach when you write new grids.

5.15.7.1 Enabling “Drag and Copy”

A popular Excel feature allows users to copy (but not move) values by dragging with the mouse while holding down the Ctrl key. To enable this functionality in your grid:

1. Enable OLE Drag and Drop.

2. Add the `OnGridDragOver()` override shown below.

   ```cpp
   // in h file
   virtual DROPEFFECT OnGridDragOver(CGXNoOleDataObjectProxy* pDataObject,
                                    DWORD dwKeyState, CPoint point);
   
   // in cpp file
   DROPEFFECT C1stGridView::OnGridDragOver(CGXNoOleDataObjectProxy* pDataObject,
                                           DWORD dwKeyState, CPoint point)
   { return CGXGridView::OnGridDragOver(pDataObject, MK_CONTROL, point); }
   ```

This technique is shown in the sample located at `<stringray-installdir>\Samples\Grid\Tutorial\1stGrid\Step3`.

For more information on the DROPEFFECT type, see the Visual C++ online help.
5.15.8 Scroll Tip Support

Before Objective Grid 6.1, the programmer had to override the WM_VSCROLL and WM_HSCROLL messages. With Objective Grid 6.1 and later, you only need to call:

EnableScrollTips();

during the initialization for your grid object. This will enable the display of the row and column number when the user is dragging the scrollbar thumb.

However, if you want to customize the text displayed in the scroll tips, you need to use the old approach instead. Embed a CGXScrollTip object into your derived class and forward WM_VSCROLL and WM_HSCROLL messages to the CGXScrollTip object.

5.15.9 CellTip Support

You can enable cell tips by calling EnableCellTips() during initialization for the grid object. Cell tips are implemented using Objective Grid’s plug-in classes. They are essentially fly-by windows that appear when the following conditions are met:

- The cell’s text is truncated (either horizontally or vertically)
- The mouse pointer has hovered over the cell for a specified amount of time
- The cell’s WrapText (Word break) setting is FALSE (disabled)

A similar feature can be seen in Visual Studio Solution Explorer window or in the Windows Explorer tree view.

5.16 Tools

5.16.1 Objective Grid Designer

This section of the User’s and Programmer’s Guide will demonstrate the steps involved in creating a sample layout file. It will also show how the layout file can be integrated into your application. For more information about relating particular Objective Grid Designer functions to Objective Grid APIs, please refer to the Objective Grid Designer help file (ogdesign.chm).

5.16.1.1 What is the Objective Grid Designer?

Objective Grid Designer is a tool that allows you to design the layout of a grid and save this layout into a file. This layout can be loaded at run time in order to initialize grids in your application.
For example, if you are designing a dialog with a grid where the user will enter data, you can specify the initial cell contents, parameter object settings, row heights, and column widths with Objective Grid Designer. This layout can be loaded into your grid with one API call. There is no need to specify the grid formatting with code.

You can also use a mixed approach where you load a layout and still apply certain formatting with code.

### 5.16.1.2 The Grid Designer Components

Figure 49 – The Objective Grid Designer

![Objective Grid Designer](image)

There are three essential components in the Objective Grid Designer interface.

1. **Grid View** - This is the main component, which uses a WYSIWYG model. This grid reflects the grid that will be created when you use the layout file in your application. The Objective Grid Designer does not make a distinction between run time and design time when designing a layout. If you specify a block of cells as read-only, those cells will be read-only in the Grid View. In order to make changes, you will have to turn the read-only flag back off. This important to keep in mind as you are designing your layout.

2. **General Settings** - The second component is the property page for general settings. General settings are settings that are applied to the whole grid.

   The property page has the following categories:

   - **Data.** Use the data property page to set the row count (for example, to 20 rows) and the column count (for example, to 40). You can also set the number of frozen rows and frozen columns with this property page. Note that the grid view
updates immediately to reflect the changed grid interface. On this page you can also set header-specific details and whether the row and column headers should be numbered.

- **User Actions.** The user actions page is very interesting since it controls many aspects of how the user will interact with the grid. This interface can be very useful in usability testing of your grid interface. You can put the user right there in front of the grid and configure the grid to suit their needs. You can specify whether the grid should support rearranging columns, allow rows and columns to be resized, and set the minimum size of row and columns, among other things. Try changing the settings and interacting with the grid. Note that you can also specify whether the grid should add support for sorting. The setting is located at the bottom of the page.

- **Selection.** It may be difficult to decide which of the various selection modes that your grid interface should support. You should take the time to select each and every option on this page to see how the grid interface interacts with this setting. This will give you a good feel for the various selection interfaces that the grid supports. It will also give you a good feel for how best to customize this to suit your needs. Also, there is support for some specific Excel emulation features that have been added. You should try all of these to see if there is some setting that your application must have! Another important aspect of this page is that it supports setting the grid into list box mode. You can choose between multiple and single selection list box modes.

  It is an interesting exercise to check how the grid list box modes interact with the selection modes. There is no end to the amount of prototyping that you can do with Objective Grid Designer!

- **Sheet.** Once you are content with the settings for the selection page, move to the next page, the sheet page. The sheet page also has several interesting features, including support for merged cells and floating cells. There is another important member in this page that you should check out and that is the Undo/Redo buffer size. You should look at optimizing this value. If you have memory constraints then you should look at the size of the Undo/Redo stack that is correct for your applications. And what better way to test this than with the Objective Grid Designer?

  You can also try out other interesting attributes, such as ‘Eat First Mouse Click’, and the default *CGXComboBox* height.

- **Display.** The display property page can be used to change the color of the grid lines, the tracking lines, and dragging lines, among other aspects of the grid. You can also specify whether these lines should be displayed or not.

- **Printing.** The printing section specifies all settings related to the printer. You can change the settings and then choose print preview from the main grid frame to get instant feedback on how the layout will appear when printed.

- **Formula Engine.** The formula engine lets you change formula engine-specific settings like calc mode, or toggling the display of formulas in cells.

3. **Cell Attributes** - The cell attributes property page lets you change the attributes of the current cell or the currently selected group of cells. These settings allow you to customize each aspect of the grid.
Most of the attributes that are on this page are self-explanatory. Some of the more arcane aspects for a beginner include the role of user attributes and the architecture of how they fit into the grid framework.

### 5.16.1.3 Tutorial for Creating a Simple Sample Layout File

**Step 1:** Start the Objective Grid Designer. You can find the Designer in the root directory of your Objective Grid installation. The Objective Grid setup program also adds a Designer icon to the program group.

After launching the Designer, you need to select the type of layout that you want to create from the **New** dialog. For purposes of this tutorial, choose the first option, **Layout**.

You will see the Objective Grid Designer and the property pages window.

Select a couple of cells and choose the cell type to be a spin edit control. You can use the cell type attribute on this page to set the type of control.

Once you have done this, scroll down the page a bit and you will see two user attributes: **spin-bound max value** and **spin-bound min value**. Set these to 100 and 10, respectively.

You will observe that the spin edit control now validates the input in the cell between these two boundaries. It ‘knows’ about these user attributes and always checks to see if they have any meaningful values and acts accordingly.

Similarly, several other controls have user attributes. These user attributes are presented on the page and can be used to control several aspects of the related controls. Remember user attributes pertain only to one control. Spin bound max value will validate only a spin control and not say, an edit control.

You are now ready to create a fully functional grid program that uses the MFC Document/View architecture without writing much code!

**Step 2:** Open an existing grid project or create a new grid project (you can use the Stingray AppWizard that ships with Objective Grid to generate a generic **CGXGridView**-based MFC application. Otherwise, take a look at the 1stGrid tutorial for creating a starter application).

**Step 3:** The layout file should be copied into the **RES** subdirectory in your project directory. Next, you have to manually edit the `.RC2` file in your application and insert a reference to your layout file.

```cpp
//
// 1STGRID.RC2 - resources Microsoft Visual C++ does not edit directly
//
#endif APSTUDIO_INVOKED

///////////////////////////////////////////////////////////////////
// Add manually edited resources here...
LAYOUT1 GXLAYOUT DISCARDABLE "RES\Layout1.OGL"
```
Binary resources must have the resource type “GXLAYOUT”. Otherwise, CreateFromResource will not find the resource in your resource file.

**Step 4:** Next, you have to manually edit the `OnInitialUpdate()` routine of your grid view class and load the parameter object from the binary resource with the following code:

```cpp
CGXGridParam* pParam = 
    CGXGridParam::CreateFromResource(IDR_GXLAYOUT1);
void C1stGridView::OnInitialUpdate()
{
    BOOL bFirstView = FALSE;
    if (GetDocument()->m_pParam == NULL)
    {
        bFirstView = TRUE;
        // bFirstView = TRUE indicates that
        // this is the first view connected
        // to the document and therefore the
        // data must be intialized.

        // construct parameter object
        // CGXGridParam* pParam =
            CGXGridParam::CreateFromFile(_T("RES\LAYOUT1.OGL"));
        // CGXGridParam* pParam = CGXGridParam::CreateFromResource(_T("LAYOUT1"));
        CGXGridParam* pParam = CGXGridParam::CreateFromResource(IDR_GXLAYOUT1);

        // if load failed, create empty object
        if (pParam == NULL)
            pParam = new CGXGridParam;
        GetDocument()->m_pParam = pParam;
    } // else
    // bFirstView = FALSE indicates that this is
    // only another view connected to the document
    // No data need be initialized. It is
    // available in the document already.

    // pass the pointer to the grid view
    SetParam(GetDocument()->m_pParam, FALSE);
        // ^-- indicates document is
        // responsible for deleting the object.

    // standard initialization, will create other objects
    // such as data object, stylesmap object, ...

    CGXGridView::OnInitialUpdate();

    // Just to be sure that everything is redrawn
    Invalidate();

    // Enable Objective Grid internal update-hint mechanism

    // You should make this line the last one
    // because as long as EnableHints is
    // not called, the modified flag
```
Step 5: Compile and run the application

There are some alternate techniques for loading the layout into the grid:

- Loading the layout into grid windows embedded in a dialog
  
  With `CMyGridWnd`, all you have to do is place a custom control in your dialog, specify `CMyGridWnd` as class name and enter the name of the binary resource (e.g. `LAYOUT1`) as caption text. `CMyGridWnd` then initializes itself automatically from the binary resource.

- Use the static method `CGXGridParam* CreateFromFile(LPCTSTR lpszFileName);`
  
  Give the name of the file and a new parameter object will be created from the layout file. The parameter object is roughly analogous to the document for the grid and the grid can initialize itself from the parameter object.

  Your `OnInitialUpdate()` code would look like this:

  ```cpp
  CGXGridParam* pParam = CGXGridParam::CreateFromFile(_T("Mylayout.ogl"));
  ```

---

5.16.1.4 Creating Formula Sheets

Objective Grid Designer supports formula sheets. When you start the Designer, you will be presented with a choice of layouts. You will have to choose Layout Formula, the second choice, in order to operate the Objective Grid Designer in Formula mode. Once the Designer has been started in Formula mode, you can then enter formulas, as you want them to appear in your completed interface.

You will also notice that the general settings tab window has a formula-specific sheet. This tab is created only when the Objective Grid Designer is operated in Formula mode. Once you have added all the data (including any formulas) that you would like to display in your view, you can save the file. The file extension for formula files is `.OGF`.

5.16.1.5 Converting .OGL (regular CGXData) Files to .OGF (CGXFormula Sheet) files

If you have any regular layout files (.OGL) that you would like to convert to .OGF (formula) files, you should start the Objective Grid Designer in ‘Formula Layout’ mode. After the Objective Grid Designer has been started in formula layout mode (a default view will be created for you), choose
Open OGL File from the File menu. Select the .OGL file that you want to open and the Objective Grid Designer will convert the data and display this file as an OGF view. (The original file will be left as is.) You can then choose save to save this file as a .OGF file.

5.16.1.6 Creating Layout Files for ODBC Queries

You can save queries and the ODBC connect string as part of the data that gets stored in the layout file. To do this select, Layout ODBC when the Objective Grid Designer is started. You will be presented with a dialog that lets you type in the required SQL query. Once you have typed in the query, the ODBC DSN dialog will be presented to you asking you to select the data source. When you have selected the data source, you will be presented with a view of this query.

You can then proceed to change the settings for the columns. Please remember that with the browser grid, it is not possible to change individual cell settings directly; only column settings can be changed. If you wish to change individual cell settings, you will have to override OnLoadCellStyle().

You can change the query by choosing re-query. Please note that you cannot change the DSN for any view. You will have to create a new view to use a different DSN.

You can also attach foreign tables with the Designer. Simply select a column and then choose Database...Attach Foreign Table from the menu. You can then specify the query for the foreign table as well as details on the key column, the text to be displayed etc. This information will be stored when you save the layout.

When you save the layout and reopen the file, you can see that everything is initialized automatically.

Please note that Objective Grid has the functionality to store the schema of a database, however it does not initialize itself from this schema by default. To implement this capability in your derived views, please look at the sample gxquery or the Objective Grid Designer source code.

Special note for ODBC layout files:

The ODBC connect string that is stored when Access files are viewed with ODBC has a parameter named DBQ. (Please refer to the MFC/ODBC documentation for further information on this parameter.) This parameter contains the path name of the Access file. For example, if your path name were c:\test\stdreg32.mdb, then your connect string would have this embedded:

...;DBQ=c:\test\stdreg32.mdb;...rest of the connect string.

If we were to use this connect string (from persistent storage), this would not work on another machine that has a different location for this file, even though this machine has the same data source registered. To work around this, we read the registry entry for this data source and get the DBQ string from there. We then replace this in the connect string. This only works for Access database sources. You will have to implement something similar to implement this support for other data sources. You can refer the gxquery sample or to the Objective Grid Designer source code for details on how this has been implemented for Access sources.
5.16.1.7 When Do I Use the Objective Grid Designer and When Do I Write Code?

To answer this question, you need to understand what the Objective Grid Designer does. The Objective Grid Designer takes advantage of Objective Grid’s support for serialization and writes all data that is entered at design-time into a binary file. This binary file is essentially CGXGridParam in a persistent state. Thereafter, the grid object in your application uses the serialization mechanism to read from this binary parameter object and initialize itself.

With the above in mind, it should be clear that the Objective Grid Designer can be used whenever a grid object needs to be initialized to some state. This same initialization can be achieved in code, but the Objective Grid Designer opens a path around the tedium of writing code for all the required calls.

What the Objective Grid Designer cannot do is to modify the behavior of Objective Grid (except through the properties that are already defined in a grid object). For example, the Objective Grid Designer can set the activation mode of cells to one of the predefined values. However, it cannot add code to pop up a dialog when the user performs a certain action. Whenever you need to enhance or override Objective Grid functionality, you go beyond the scope of the Objective Grid Designer. However, you can still use the Objective Grid Designer to lay out your grid object. The additional code needed to implement your special functionality should be added to your class.

5.16.2 Stingray AppWizard

The Stingray AppWizard is designed to help you create MFC applications that use the Objective Grid classes instead of the default MFC classes.

The interface of the Stingray AppWizard is identical to the Visual Studio AppWizard. You can use the Stingray AppWizard in the same manner as the regular AppWizard. We have added additional steps to customize your Objective Grid applications. The generated code will have a lot of sample code that may be used in ways that suit your needs.

5.16.2.1 Steps for Creating a Starter Application

You should see the Stingray Custom AppWizard appear when you choose the File | New | Projects menu from Microsoft Visual Studio.

Follow the normal steps when creating a new application, but choose the Stingray Objective Grid instead of the default MFC Application Wizard.
Select the options that you need as you would with any other AppWizard application. The interface is the same as the regular AppWizard.

When you come to the page Generated Classes, change the base class of the view from the default CView to CGXGridView. This step is very important in order to have Objective Grid code generated. Stingray AppWizard will warn you if you do not choose CGXGridView. You may choose to have CView-specific code in which case you can ignore the warning.

In the subsequent custom dialogs, select the Objective Grid options that you wish to include.
Figure 51 – Stingray AppWizard Options

Figure 52 – Choosing CGXGridView as base class
Once you click Finish, a complete project is generated that uses CGXGridView instead of CView. All Objective Grid-specific code needed for such an application is automatically added. We have also added extensive comments to help customize the application to your needs.

Simply compile the application and run it. You should see a complete Objective Grid application that is based on the document/view architecture.

The following points are important to keep in mind:

 The wizard cannot be used to generate OLE-enabled applications. It also cannot be used to generate dialog-based applications that use the grid.

 When data bound views are generated, formula support is disabled. Serialization support is enforced.

 When generating a data bound view, do not include support for the database headers or select binding to any datasource. Select 'None' for the database support. The Stingray AppWizard will take care of doing this for you.
Reducing the Size of Your Application

6.1 How Objective Grid Affects the Size of Your Application

Objective Grid consists of many thousand lines of code that implement a whole variety of different features and cell types. While some features and cell types might be essential for your application there will be many other features or cell types that are not needed in your application. Objective Grid gives you the option to choose the features and cell types you need in your application. Any unwanted features can be excluded. This allows you to minimize the distribution size of your application. The overhead added by Objective Grid to your application varies between 150 KB and 1,500 KB depending on how many cell types and features you link into your application.

Objective Grid offers two alternatives to be distributed with your application:

- Using Objective Grid as a shared DLL.
- Linking Objective Grid static into your application.

There are special considerations what solution is better for your specific project(s).

6.1.1 Using Objective Grid As a Shared DLL

Using Objective Grid as a shared DLL is recommended if your project consists of several executable files. The advantage of this solution is that all executables in your project can share the same DLL that implements the grid functionality.

Another advantage of this solution is a reduced linking time for your project. This is advantageous while doing development of your application. Once you build a release version of your project this linking time of your project should not be a criteria any more.
When you use Objective Grid as shared DLL all features that you want to use in any of your applications must be linked into the DLL. Objective Grid allows you to reduce the size of the DLL through its “Build Wizard” tool discussed later. The size of individual executables is not depending on the number of features and cell types linked into the Objective Grid DLL.

The main step for linking Objective Grid as shared DLL is to define \_GXDLL in your project’s C/C++ compiler settings.

### 6.1.2 Linking Objective Grid Statically

Linking Objective Grid statically into your application is recommended if your project consists only of one executable. This solution has the advantage that you don’t have to worry about redistributing the Objective Grid DLL with your application.

When you link Objective Grid statically into your application the Visual C++ Linker will link only to those Objective Grid modules that are referenced in your application. Independent features and cell types are implemented in different modules. In your application you can implement a “Control Factory Class”. This class establishes references to features and cell types you want to use in your application. Objective Grid helps you with creating such a Control Factory Class though the Build Wizard tool. If you don’t create an individual Control Factory Class for your application, Objective Grid will use a default Control Factory Class provided by the library that forces all features and cell types available in the static library to be linked into your application. Control factory classes and how they work with the Build Wizard are discussed in more detail in Section 6.2 and Section 6.3.

So, when you link Objective Grid statically, you can choose features and cell types individually for each application. Therefore it is recommended that you generate static Objective Grid libraries with all features and cell types linked into the library. You can then use the same static library for many different projects.

With DLLs you should create an individual DLL for each project that needs a different set of grid features and cell types.

---

**We recommend that you create Objective Grid libraries that provide all cell types and features. Use the Control Factory dialog screen of the Build Wizard instead to select those features you want to link into individual applications. The Visual C++ linker automatically binds only modules that are referenced in your project and in the project’s control factory. If you want to ship your application using Objective Grid as DLL you can later run the Objective Grid Build Wizard and create DLLs that are as small as possible.**

Let’s discuss the Control Factory Class, and Build Wizard in more detail.
6.2 The Control Factory Class

Objective Grid provides an abstract base class for Control Factory Classes, the `CGXAbstractControlFactory` class. This class defines the interface for the Control Factory Class. In order to create a Control Factory Class, derive from `CGXAbstractControlFactory` and override all pure virtual methods. While the Control Factory dialog of the Build Wizard fully automates the creation of the Control Factory Class, it is helpful to know how the Control Factory Class works:

- Registration of cell types and instantiation of cell type objects on demand.
- Registration of “concrete implementations” for various grid features like Undo/Redo, Serialization and more. The term “concrete implementation” means that a C++ object gets allocated that is derived from an abstract C++ base class.
- Registration of window classes for custom controls in dialog templates.

6.2.1 Registration of Cell Types

In Objective Grid each cell type is implemented through a class derived from `CGXControl`. `CGXControl` defines the necessary interface for the grid to be able to interact with cells. `CGXControl` also implements some default behavior for most functions. Objective Grid has no dependencies on any cell type object. You can derive custom cell type objects from `CGXControl` or use the default cell types. Only cell types that get registered will be linked into your application or DLL.

Registration of cell types is implemented through the following methods:

```cpp
virtual CGXControl* CreateControl(UINT nID, CGXGridCore* pGrid);
```

Each grid maintains a map of cell type (`CGXControl`) objects. A cell type is identified through the cell type id, an unsigned integer constant. Whenever a cell type is requested that is not yet registered in the grid, the grid calls `CreateControl()`. `CreateControl()` is responsible for instantiating the correct cell type object. The grid stores the cell type object in its map and will keep this cell type object alive until the grid gets out of scope. When the grid is destructed all cell type objects in the map will be destroyed.

```cpp
virtual void RegisterAllControls(CGXStylesMap* pStylesMap);
```

Registers the cell type names with the styles-map. This is necessary when you want the various cell types to appear in the Controls-page of the `CGXStyleSheet`.

```cpp
virtual void RegisterAllUserAttributes(CGXStylesMap* pStylesMap);
```

Registers user attribute names with the styles-map. This is necessary when you want the user attributes to appear in the User Attribute-page of the `CGXStyleSheet`.
6.2.2 Registration of Concrete Implementations for Grid Features

In Objective Grid features that are not essential for every grid application are implemented through abstract base classes. Find and Replace is an example of a feature that only gets linked into your application or DLL if a concrete implementation is registered with the grid. Objective Grid interacts with a concrete `FindReplace` object through an interface that is defined by an abstract base class. Whenever Objective Grid needs to call a `FindReplace`-specific method it first checks if a `FindReplace` object is available.

```cpp
BOOL CGXGridCore::FindText(BOOL bSetCell)
{
    if (m_pFindReplaceImp)
        return m_pFindReplaceImp->FindText(this, bSetCell);
    return FALSE;
}
```

In the above example `m_pFindReplaceImp` is a pointer to the abstract base class `CGXAbstractGridFindReplaceImp`. The `FindReplace` functionality is implemented in the `CGXGridFindReplaceImp` class that is derived from `CGXAbstractGridFindReplaceImp`. A concrete object is instantiated through the `ImplementFindReplace` method:

```cpp
void CGXGridCore::ImplementFindReplace()
{
    if (m_pFindReplaceImp == NULL)
        AutoDeletePtr(m_pFindReplaceImp = new CGXGridFindReplaceImp);
}
```

Using an abstract base class eliminates any dependencies within Objective Grid to the concrete `FindReplace` implementation. The abstract base has no implementation and therefore no code for `FindReplace` needs to be linked into the application or DLL if `ImplementFindReplace` is not called.

Registration of concrete implementations for grid features (by calling `ImplementFindReplace` for example) is done with the following methods:

```cpp
virtual void InitializeGridComponents(CGXGridCore* pGrid);
```

This method is called from each grid at initialization time (from `OnGridInitialUpdate()` to be more concrete). It instantiates concrete implementations for grid features and attaches the instantiated object to the grid. The grid will keep the object alive until the grid gets out of scope. When the grid is destructed all grid feature objects will be destroyed. This method instantiates objects that are used by particular grids (referred to by `pGrid`).

```cpp
virtual void InitializeApplicationComponents();
```

This method is called from Objective Grid at application initialization time (from `GXInit`). It instantiates concrete implementations for various features. Objective Grid will keep the objects alive until the application terminates or `GXForceTerminate()` is called. Objective Grid will then destroy the objects. This method instantiates objects that are useful to the application.
Previous versions of Objective Grid supported the function GXTerminate(), but this function is now obsolete. Please use GXForceTerminate() as stated above.

6.2.3 Registration of Window Classes

Registration of window classes is implemented through the following methods:

virtual void
RegisterWndClasses(HINSTANCE hInstance);

Registers custom window classes. If, for example, CGXGridWnd gets registered, you can use the CGXGridWnd class as user control in dialog templates in your application.

virtual void
UnregisterWndClasses(HINSTANCE hInstance);

Undoes the registration of previously registered window classes.

The following code shows a sample implementation of a Control Factory Class:

```
#include "stdafx.h"
#include "gridappf.h"

void CControlFactory::InitializeGridComponents(CGXGridCore* pGrid)
{
    pGrid->ImplementFindReplace();
    pGrid->ImplementHideCells();
    pGrid->ImplementAutoScroll();
}

void CControlFactory::InitializeApplicationComponents()
{
    CGXPrintDevice::ImplementPrintDevice();
    CGXStyle::ImplementCompareSubset();
    CGXStyle::ImplementSerialize();
    CGXStyle::ImplementSerializeOG5Compatible();
    GXImplementOleDateTime();
}

void CControlFactory::RegisterAllControls(CGXStylesMap* pStylesMap)
{
    // Basic cell types
    pStylesMap->AddControl(GX_IDS_CTRL_EDIT);
    pStylesMap->AddControl(GX_IDS_CTRL_HEADER);
    pStylesMap->AddControl(GX_IDS_CTRL_STATIC);
}

CGXControl* CControlFactory::CreateControl(UINT nID, CGXGridCore* pGrid)
{
    CGXControl* pControl = NULL;
    switch (nID)
    {
    case GX_IDS_CTRL_EDIT:
```
pControl = new CGXEditControl(pGrid, GX_IDS_CTRL_EDIT);
break;

case GX_IDS_CTRL_STATIC:
pControl = new CGXStatic(pGrid);
break;

case GX_IDS_CTRL_HEADER:
pControl = new CGXHeader(pGrid);
break;
}

return pControl;
}

void CControlFactory::RegisterAllUserAttributes(CGXStylesMap* pStylesMap)
{
    CGXControl::AddUserAttributes(pStylesMap);
}

void CControlFactory::RegisterWndClasses(HINSTANCE hInstance)
{
    CGXGridWnd::RegisterClass(hInstance);
}

void CControlFactory::UnregisterWndClasses(HINSTANCE hInstance)
{
    CGXGridWnd::UnregisterClass(hInstance);
}
6.3 The Build Wizard

The Objective Grid Build Wizard lets you customize the Objective Grid libraries to your needs. The Build Wizard allows you to link only those features you really want to use into the library and exclude any unwanted features and cell types. The Build Wizard automatically generates a makefile for building a custom library and DLL based on your settings. Other files generated by Build Wizard are header files with configuration information and a Control Factory Class file. The Control Factory Class will be linked into the library and serves as default Control Factory Class for all your applications. Applications can still provide an individual Control Factory Class or use the default Control Factory Class provided by the library.

You can find the Objective Grid Build Wizard in the `Utils\Grid` directory of your Objective Grid installation. The filename is `<stingray-installdir>\Utils\Grid\BuildWiz.exe`. The Build Wizard can be run from the Objective Grid program group under the Start menu. When run, the Build Wizard walks you through a series of pages, collecting information about which features from the product you use, what library names you wish to target etc. After you’ve answered these questions, the Build Wizard will generate the makefile that meets your specifications and write that makefile to `<stingray-installdir>\src\grid.mak`. You will be given the choice of starting Visual Studio on the resultant makefile or simply exiting from the Build Wizard. If you choose to exit, you are responsible for opening the makefile yourself in Developer Studio and building the Objective Grid library.

If you want to use the Build Wizard for multiple library configurations (for example, one configuration that has database support and one configuration that has no database support), you can assign each configuration a unique name in the first page when running the Build Wizard. The Build Wizard will generate a header file `config/<NAME>.h` in the “Include” directory of the Objective Grid installation and also store the set of chosen features in the file `<NAME>.BWC` in the `Utils` directory ( `<NAME>` is a place holder for the configuration name you specify). You can rerun the Build Wizard any time and adjust the selections you made for a specific configuration.

Be sure to use different library names for each configuration and re-link the Objective Grid library after running the Build Wizard for each configuration. You can’t build the libraries multiple different configurations all at once. You have to run Build Wizard for each configuration before building the libraries.

The Build Wizard seamlessly integrates with the existing Objective Grid library “autolink” function. This means linking to the appropriate Objective Grid library is as simple as including the header file with the configuration information in your application’s precompiled header. If you don’t include explicitly include a header file with configuration information Objective Grid will use `config/gxstndrd.h` as default. `GXSTNDRD` is the default configuration name for the Objective Grid libraries. (This generated file that is to be included has to be before any other grid header includes)

The Build Wizard loads its configuration information from the file `BuildWiz.ini` in the `Utils` directory and various `.BWI` files in the `BWInfo` subdirectory. Alteration of the file `BuildWiz.ini` and other `.BWI` files will allow you to further customize the build process, including adding your own source files, but please understand that this action is not supported by Rogue Wave!

Alteration of these files are done at your own risk. (We recommend that you keep a backup of the files before proceeding.)
6.3.1 DLL Naming

When building a custom configuration of Objective Grid, it is important that you specify a unique DLL target name. When a subset of the Objective Grid features are built or you make a change to Objective Grid source or header files, the signature of the library changes. Therefore, when you build a DLL that incorporates a subset of Objective Grid features or your own changes, the target DLL must be treated as a completely unique DLL. If you change the signature of Objective Grid and do not specify a DLL target name, other applications that link to the Objective Grid DLL may fail.

6.3.2 Building the Libraries

If you are going to build with Unicode, uncheck the option ExcelReadWrite in Build Wizard.

After you have run Build Wizard, you can build the libraries. The following steps show how to build them:


2. Open \(<stingray-installdir>\src\grid<ver>.sln\), where <ver> represents the VC++ version.

3. After the .sln file is loaded, choose the build configuration you want. The choices are Non-Unicode (default) and Unicode builds.

   Unicode is not supported in this version of the Excel read/write classes. When building Grid libraries with Unicode, an alternative to the read/write classes is Excel Automation, discussed in Section 26.7, “Excel Automation with Objective Grid.”

4. Start the build.
6.4 Component Reference

This section provides detailed description of all cell types and features that can be chosen in the tree window of the Build Wizard.

In the Build Wizard, you can choose the grid features you want to use in your application(s). The wizard will create a makefile and/or control factory class based on your selection. You can adjust your selections and rebuild the libraries any time.

Some components depend on other components. For example, the CGXComboBox is derived from CGXEditControl and CGXStatic. When you mark ComboBox as checked, the cell types Static and Edit will also be linked into the grid library or application.

Another example is clipboard support. If you choose StyleDataExchange, the CGXStyle::Serialize API will also be linked into the grid library or application.

If you wish to understand the details of the component dependencies, open the file buildwiz.ini and search for the m keys.

6.4.1 Control Factory Class

This section describes the components in the tree window that specify the output for the control factory class. Each of the components below add one or several lines of code in the Control Factory Class.

The list of file names will be written into the makefile for the grid library.

6.4.1.1 Basic Cell Types

Choose the cell types to be used in your application(s). Entries for all selected cell types will be inserted in the control factory’s CreateControl method. You can toggle the state of all components at once by clicking the checkbox of the parent node.

- **Static cell type** - Highly recommended for all applications
  
  Classname: CGXStatic
  
  Cell type ID: GX_IDS_CTRL_STATIC
  
  Do not confuse static with read-only! The user can modify static cells (for example the user can delete the cell or paste text into it) if you don’t explicitly set the cell read-only.
  
  The cell value will be displayed in the cell.

- **Header cell type** - Highly recommended for all applications
  
  Classname: CGXHeader
  
  Cell type ID: GX_IDS_CTRL_HEADER
  
  The cell value will be displayed in the cell.
◆ **Edit control.** Highly recommended for all applications.

Classname: *CGXEditControl*

Cell type ID: *GX_IDS_CTRL_EDIT*

The cell value will be displayed and can be edited by the user in the cell.

◆ **Combobox**

Classname: *CGXCombobox*

Cell type IDs:
- *GX_IDS_CTRL_COMBOBOX*: combobox allows user to input any text
- *GX_IDS_CTRL_TEXTFIT*: combobox only allows text input which fits the choice list
- *GX_IDS_CTRL_ONEBASED*: combobox uses an integer as cell value
- *GX_IDS_CTRL.ZEROBASED*: combobox uses an integer as cell value
- *GX_IDS_CTRL.ONEBASED_EX*: use an integer as value and display choice as cell text
- *GX_IDS_CTRL.ZEROBASED_EX*: use a zero-based integer as value and display choice as cell text

Supply the listbox entries through the choice list separated with a newline character. The representation of the selected choice as cell value depends on the combobox variant you use. You can register custom variants for *CGXComboBox* and adjust some attributes for the cell type object (e.g., for pushbutton to be always visible). See *CGXComboBox* in the Objective Grid Class Reference for more details.

◆ **Checkbox**

Classname: *CGXCheckbox*

Cell type IDs:
- *GX_IDS_CTRL_CHECKBOX*: Flat checkbox look
- *GX_IDS_CTRL_CHECKBOX3D*: 3D-look

Supply the checkbox description to be displayed in the cell through the choice list character. By default the cell value is 0, 1, or 2 (as defined by the user attributes below), but you can also adjust how checked, unchecked, and undetermined states shall be represented through the cell value. (In a tri-state checkbox an empty string in the corresponding style's value represents the indeterminate state.)

You can specify the checked/unchecked representation in the style object with the user attributes *GX_IDS_UA_CHECKBOX_CHECKED* and *GX_IDS_UA_CHECKBOX_UNCHECKED*. The default setting is “1” for checked and “0” for unchecked. Other values are treated as undetermined.
If you want "T" or "F" to be stored in the grid for a checkbox you can specify this with:

```cpp
// Checkbox: "T" is checked, "F" is unchecked
// instead of "1" and "0"
StandardStyle().SetUserAttribute(GX_IDS_UA_CHECKBOX_CHECKED, _T("T"));
StandardStyle().SetUserAttribute(GX_IDS_UA_CHECKBOX_UNCHECKED, _T("F"));
```

The state of the checkbox is passed through the style object's value-attribute (CGXStyle::SetValue).

See CGXCheckbox in the Objective Grid Class Reference for details.

- **Listbox**
  
  Classname: CGXListBox

  Cell type ID: GX_ID_CTRL_LISTBOX

  Supply the listbox entries through the choice list separated with a newline character. The cell value is the choice list text of the selected choice.

- **Pushbutton**

  Classname: CGXPushbutton

  Cell type ID: GX_ID_CTRL_PUSHBTN

  Supply the cell text to be displayed in the button through the choice list.

- **Radiobutton**

  Classname: CGXRadioButton

  Cell type IDs:

  - GX_IDCTRL_RADIOBTN: Flat button look
  - GX_IDCTRL_RADIOBTN3D: 3d-look

  Supply the list of selections through the choice list separated with a newline character. The cell value is a zero-based integer that represents the selected choice.

### 6.4.1.2 Advanced Cell Types

- **Dropdown checklist box**

  Classname: CGXCheckListComboBox

  Cell type ID: GX_IDCTRL_CHECKLIST_COMBOBOX

  Supply the list of selections through the choice list. See CGXCheckListComboBox in the Objective Grid Class Reference to see how the cell value is determined.

- **MFC Combobox**

  Classname: CGXComboBoxWnd

  Cell type IDs:
- **GX_IDS_CTRL_CBS_DROPDOWN**: MFC CComboBox, CBS_DROPDOWN style. User can edit the selected choice.

- **GX_IDS_CTRL_CBS_DROPDOWNLIST**: MFC CComboBox, CBS_DROPDOWNLIST style. No editing of the selected choice.

Supply the listbox entries through the choice list separated with a newline character. By default the cell value will be the text of the selected choice. You can register custom variants for CGXComboBoxWnd and adjust some attributes for the cell type object (e.g. zero-based index as value). See CGXComboBoxWnd in the Objective Grid Class Reference for more details.

- **Tabbed combobox**
  
  Classname: CGXTabbedComboBox

  Cell type ID: GX_IDS_CTRL_TABBED_COMBOBOX

  Supply the listbox entries through the choice list separated with a newline character. Separate the value for each column with a tab character (\t). The cell value will be the text of the key column for the selected choice. See CGXTabbedComboBox for user attributes that specify the appearance and behavior of the dropdown list.

- **Tabbed MFC combobox**
  
  Classname: CGXTabbedComboBoxWnd (not supported for Win32s).

  Cell type IDs:

  - **GX_IDS_CTRL_CBS_TABBED_DROPDOWN**: Tabbed combobox with CBS_DROPDOWN style. User can edit text and while entering text the combobox will fill up the remaining text with a choice list entry that fits the specified text.

  - **GX_IDS_CTRL_CBS_TABBED_DROPDOWNLIST**: Tabbed combobox with CBS_DROPDOWNLIST style. User cannot edit the text. Optimal if you want to select and display bitmaps in the cell.

Supply the listbox entries through the choice list separated with a newline character. Separate the value for each column with a tab character (\t). The cell value will be the text of the key column for the selected choice. See CGXTabbedComboBoxWnd for user attributes that specify the appearance and behavior of the dropdown list.

- **Masked Edit**

  Classname: CGXMaskControl

  Cell type ID: GX_IDS_CTRL_MASKEDIT

  Supply the input mask through user attributes. The cell value will be the text displayed in the cell without literals (literals are non-editable characters in input mask). See CGXMaskControl for the user attributes.

- **Password**

  Classname: CGXPasswordControl

  Cell type ID: GX_IDS_CTRL_PASSWORD

  The cell value contains the value.
Progress

Classname: *CGXProgressCtrl*

Cell type ID: **GX_IDS_CTRL_PROGRESS**

Cell Value contains the value to be graphically displayed in the cell. Specify the upper and lower boundaries with user attributes.

Rich Edit

Rich Text Format Editing

Classname: *CGXRichEditCtrl*

Cell type ID: **GX_IDS_CTRL_RICHEDIT**

Cell value contains the RTF value for the cell. Use **GetControlText** to get the plain text value without RTF commands.

Date Time

Date and Time Control with popup calendar

Classname: *CGXDateTimeCtrl*

Cell type IDs:

- **GX_IDS_CTRL_DATETIME**: DateTime cell with popup calendar
- **GX_IDS_CTRL_DATETIMENOCAL**: DateTime cell without popup calendar

By default, the cell value contains the date string. If you need to write international applications check the **m_bDateValueAsNumber** member attribute of the *CGXDateTimeCtrl* class. This will force the date to be represented as number in the cell value.

Use user attributes to customize the behavior of individual date time cells.

Changing the Spinner Speed on the *CGXDateTimeCtrl*—Two public members of *CGXDateTimeCtrl* manage how often a value is incremented as you click and hold down the spinner. **m_nInitialTimer** is the time (in milliseconds) that the button needs to be depressed before the first increment. Thereafter, **m_nNormalTimer** is the time (in milliseconds) between increments. The default value of **m_nInitialTimer** is 500 msecs and the default value of **m_nNormalTimer** is 100 msecs. For example, to change the incrementing speed once the increment begins, use code such as:

```cpp
{{CGXDateTimeCtrl*})GetRegisteredControl
    (GX_IDS_CTRL_DATETIME)) ->m_nNormalTimer = 50;
```

after the baseclass call to **OnInitialUpdate()** or **Initialize()**.

Currency

Currency input with popup calculator

Classname: *CGXCurrencyEdit*

Cell type ID: **GX_IDS_CTRL_CURRENCY**

The cell value is the number in the currency edit in ANSI format without any formatting (thousand separators etc.). The decimal point is ..
For information about how to use user attributes to customize the behavior of individual currency cells, see CGXCurrencyEdit in the Objective Grid Class Reference.

6.4.1.3 Edit Extensions

- **Hotspot**
  Edit control with hotspot button
  Classname: CGXHotSpotEdit
  Cell type ID: GX_IDS_CTRL_HOTSPOT
  When user clicks on the button, the grid will fire an OnClickedButton event.

- **Spinner**
  Edit control with spinner buttons
  Classname: CGXSpinEdit
  Cell type ID: GX_IDS_CTRL_SPINEDIT
  Use user attributes to specify the boundaries of the spin control.

- **Vertical Scrollable**
  Edit control with vertical scrollbar
  Classname: CGXVScrollEdit
  Cell type ID: GX_IDS_CTRL_SCROLLEDIT
### 6.4.1.4 Basic Features

#### Sorting
ImplementGridSort: Provides implementation for sorting rows or columns. Call `CGXGridParam::SetSortRowsOnDb1C1k` or `CGXGridParam::SetSortColsOnDb1C1k` to enable sorting the grid when the user double-clicks on a row or column header. You can also call the SortRows or SortCols API to manually sort the grid with multiple key columns.

#### HideCells
ImplementHideCells: Provides implementation for hiding rows and columns. The user can hide rows and columns with the mouse if resizing rows and column is supported. See “UserResizeCells” under “Mouse Actions”. Use the HideRows or HideCols API to manually hide or unhide rows and columns.

#### FindReplace
ImplementFindReplace: Provides implementation for Find and Replace. Add `Edit|Find` and `Edit|Replace` menu entries to your application with the ids `ID_EDIT_FIND` and `ID_EDIT_REPLACE` to add Find and Replace support to a gridview. You can also call the FindText API to manually find text in the grid.

#### MarkEditHeader
ImplementMarkEditHeader: Outline row and column header of current cell. This behavior is controlled through `CGXProperties::SetMarkColHeader` and `CGXProperties::SetMarkRowHeader`.

#### FreezeMenu
ImplementFreezeMenuHelper: Provides implementation for FreezeRows, UnfreezeRows, FreezeCols and UnfreezeCols menu handlers.

#### Serialization
C`G`XGridParam::ImplementSerialize: Provides implementation for serializing `CGXGridParam`, `CGXProperties`, `CGXData` and `CGXStylesMap` objects. Forces also that the implementation for serializing style objects gets linked into the application or DLL.

#### Profile
CGXStyle::ImplementProfile: Provides support for reading and writing `CGXProperties`, `CGXStyle`, `CGXFont` and `CGXStylesMap` objects to the registry.

#### UndoRedo
ImplementUndoRedo: Implements Undo and Redo support for undoing and redoing operations. Undo and Redo is controlled by the `CGXGridParam::EnableUndo` and `CGXGridParam::SetUndoLimit` parameter. Add `Edit|Undo` and `Edit|Redo` handlers to your Edit menu or use the `Undo()`, `Redo()` and `Rollback()` API to manually perform Undo and Redo.

#### UpdateHint
ImplementUpdateHint: Provides implementation of the update hint mechanism that synchronizes changes between several views of one document. The Update Hint behavior is controlled by the `EnableHints()` API for each grid. You have to call `EnableHints()` from your `OnInitia1Update()` routine in order to enable the update hint mechanism for individual grids.

#### ResizeToF1t
ImplementResizeToF1t: Provides implementations for the `ResizeRowHeightsToF1t()` and `ResizeColHe1thsToF1t()` methods. ImplementResizeToF1t is also needed if you want to support automatic growing of cells while the user enters text `(CGXStyle::SetAutoSize(TRUE))`. 
<table>
<thead>
<tr>
<th><strong>NumberFormatting</strong></th>
<th>CGXDrawingAndFormatting::ImproveNumberFormatting(): Provides concrete implementation for format numeric cells with CGXStyle::SetFormat.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LongOperation</strong></td>
<td>CGXLongOperation::ImplementLongOperation(): Provides implementation for the CGXLongOperation class. This class allows you to give feedback to the user about the status of time-consuming operations. The user can cancel time-consuming operations with the ESC-key. The behavior of the CGXLongOperation class is also controlled by the SetTicksFirstTime parameter. If you call CGXLongOperation::SetTicksFirstTime(LONG_MAX) the user can’t cancel long operations with the ESC key.</td>
</tr>
</tbody>
</table>
6.4.1.5 Mouse Actions

**ToolTips**
Implement ToolTips: Support for Tooltips in cells. You can turn on and off tooltips for each grid individually with EnableToolTips().

**AutoScroll**
Implement AutoScroll: Provides implementation for Autoscrolling that will scroll the grid when the user presses a mouse button and drags the mouse out of the grid window. Also, will accelerate the scrolling when the user navigates through the grid by holding down an arrow key or pressing a scrollbar arrow button.
Scrolling when the user drags the pressed mouse out of the grid is controlled with SetAutoScroll() .
Acceleration of the scrolling when the user holds down an arrow key is controlled with SetAccelArrowKey() .
Acceleration of the scrolling when the user presses a scrollbar button is controlled with SetAccelScrollbars() .

**UserSelectRange**
Implement UserSelectRange: Provides implementation for selecting cell ranges with mouse or keyboard or SelectRange API.
The “Select Range” behavior is controlled by CGXGridParam::EnableSelection.
You can manually select ranges with the SelectRange and SetSelection API.

**ExcelLikeFrame**
Implement ExcelLikeFrame: Provides implementation for Excel-like Selection Frame.
In order to turn on Excel-like Selection Frame for individual grids call GetParam()->SetExcelLikeCurrentCell(TRUE); GetParam()->SetSyncCurrentCell(TRUE); GetParam()->SetExcelLikeSelectionFrame(TRUE); GetParam()->

**UserDragSelectRange**
Implement UserDragSelectRange: Provides implementation for dragging selected rows or columns with mouse.
Dragging selected rows or columns with mouse allows the user to rearrange the row or column order in the grid.
You can turn on and off dragging rows and column with CGXGridParam::EnableMoveRows() and CGXGridParam::EnableMoveCols(). You can manually rearrange rows and columns by calling MoveRows() or MoveCols().

**UserResizeCells**
Implement UserResizeCells: Provides implementation for resizing rows or columns with the mouse.
Call CGXGridPram::EnableTrackRowHeights() and CGXGridPram::EnableTrackColWidths() in order to enable or customize the resize behavior for individual grids.

6.4.1.6 Spanned Cells
This section discusses various approaches for joining neighboring cells.
<table>
<thead>
<tr>
<th><strong>CoveredCells</strong></th>
<th>ImplementCoveredCells: Allow usage of covered cells in the grid. You can specify covered cells with the <code>SetCoveredCellsRowCol()</code> method.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MergeCells</strong></td>
<td>ImplementMergeCells: Implementation for Merge Cells. You can enable and customize merge cells behavior with the <code>SetMergeCellsMode()</code> method in your grid. For individual cells use <code>CGXStyle::SetMergeCell()</code>.</td>
</tr>
<tr>
<td><strong>FloatCells</strong></td>
<td>ImplementFloatCells: Implementation for Floating Cells. You can enable and customize floating cells behavior with the <code>SetFloatCellsMode()</code> method in your grid. For individual cells use <code>CGXStyle::SetFloatCell()</code> and <code>CGXStyle::SetFloodCell()</code>.</td>
</tr>
</tbody>
</table>
6.4.1.7 Printing

**Grid Printing**

ImplementPrinting: Provides implementation of printing routines that allow you to print and print preview grid. Printing and Print Preview is implemented in the MFC framework. The MFC framework calls various virtual methods when printing a view. ImplementPrinting provides concrete implementations for the OnBeginPrinting(), OnPrint() and OnEndPrinting() methods.

**Printer Setup**

CGXPrintDevice::ImplementPrintDevice(): Store printer settings into grid parameter object and serialize printer settings. Also needed for WYSIWYG mode. See CGXPrintDevice for more details.

6.4.1.8 Data Exchange

Data Exchange functionality is necessary for doing clipboard operations, OLE drag and drop and reading and writing text files.

**DirectCopyMoveCells**

ImplementDirectCopyMoveCells: Implements support for direct copying, moving of cells within one grid (MoveCells and CopyCells API). You can enable DirectCopyMoveCells for clipboard and OLE Drag and Drop operations if you set m_bDirectCopyPaste, m_bDirectCutPaste and m_bDirectDragDrop equal to TRUE. For regular grids, these parameters are set to FALSE as default. For formula grids, these parameters are set to TRUE as default. Use the MoveCells and CopyCells API to programmatically perform a direct copy or move operation.

**TextDataExchange**

ImplementTextDataExchange: Implements support for reading and writing cells with text (CSV) format. Clipboard and OLE Drag and Drop operations use TextDataExchange when you specify the GX_DNDTEXT flag. Call CopyTextToFile() or PasteTextFromBuffer() to programmatically read and write cells with text format.

**StyleDataExchange**

ImplementStyleDataExchange: Implements support for reading and writing cells in internal format. Clipboard and OLE Drag and Drop operations use StyleDataExchange when you specify the GX_DNDSY Tyles flag. Call CopyCellsToArchive() or PasteCellsFromArchive() to programmatically read and write cells in binary format.
6.4.1.9 Drawing

Drawing related features that are implemented through abstract base classes. Concrete classes will be instantiated for chosen features.

CutPaste

ImplementCutPaste: Implements support for clipboard operations: Cut, Copy and Paste.
This component provides concrete implementations for the `Cut()`, `CanCut()`, `Copy()`, `CanCopy()`, `Paste()` and `CanPaste()` methods. `CGXGridView` provides handlers for `ID_EDIT_COPY`, `ID_EDIT_PASTE` and `ID_EDIT_CUT` menu commands that call Cut, Copy and Paste methods.
You can customize Cut, Copy and Paste behavior through the `m_nClipboardFlags` attribute.

ClearCells

ImplementClearCells: Clear out cells when user pressed DELETE key.
This component provides a concrete implementation for the `ClearCells()` method.

OldDrawing

ImplementOldDrawing: Old Drawing, backward compatible with version 1.2.
Use old drawing if you want to support transparent background (`CGXGridParam::SetTransparentColor(TRUE)`) or if you depend on a OG 1.2-compatible behavior for drawing grid lines in case you have override the `DrawGridLine()` method.
OldDrawing can be forced by setting `m_bForceOldDrawing` attribute equal to `TRUE`.

DrawRotatedText

CGXDrawingAndFormatting::ImplementDrawRotatedText: Support for rotated text in cells.
This component provides a concrete implementation for the `GXDrawRotatedText()` method which is called for drawing the text in a cell if you specified `CGXStyle::SetOrientation(nValue)` with `nValue` not equal to 0.

OptimizedUpdate

ImplementOptimizedUpdate: Provides implementation for optimized updating of grid when inserting, moving, removing or resizing rows or columns. If optimized updating is not available the grid will simple redraw the window after these operations (which is already fast enough for typical grids).
### 6.4.1.10 Date Time Formatting

Choose the date and time implementation. **ColeDateTime** is recommended for Win32 applications. Uncheck both if you don’t need date and time parsing and formatting at all.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OleDateTime</strong></td>
<td>Use <strong>ColeDateTime</strong> for formatting and parsing inGXParseDateTime() and GXFormatTimeStamp() methods. Recommended for Win32. (<strong>GXImplementOleDateTime()</strong> in control factory)</td>
</tr>
<tr>
<td><strong>NoOleDateTime</strong></td>
<td>Use alternative to <strong>ColeDateTime</strong> for formatting and parsing inGXParseDateTime() and GXFormatTimeStamp() methods. Not recommended for Win32. (<strong>GXImplementOleDateTime()</strong> in control factory)</td>
</tr>
</tbody>
</table>

### 6.4.1.11 Advanced Settings

Check those APIs you are using in your project. Check also if you need to read back serialized documents written with OG 5.0 or earlier.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Copy Parameter Object</strong></td>
<td>CGXGridParam::ImplementCopyOperator(): Implements copy operator for CGXGridParam, CGXProperties, and CGXStylesMap. (CGXData and CGXStyle have a fully functionally copy-operator, no matter if you enable “Copy Parameter Object” or not.)</td>
</tr>
<tr>
<td><strong>MoveRows MoveCols API</strong></td>
<td>ImplementMoveRowsCols: Check this if you have disabled &quot;UserDragSelectRange&quot; and you want to move columns and rows programmatically by calling MoveRows() or MoveCols().</td>
</tr>
<tr>
<td><strong>CGXStyle::Serialize API</strong></td>
<td>CGXStyle::ImplementSerialize: Check this if you have disabled &quot;Serialization&quot; and &quot;StyleDataExchange&quot; and you want to programmatically call CGXStyle::Serialize().</td>
</tr>
<tr>
<td><strong>CGXStyle::IsSubset API</strong></td>
<td>CGXStyle::ImplementCompareSubset(): Check this if you want to call CGXStyle::IsSubset() in your project.</td>
</tr>
<tr>
<td><strong>CGXStyle::Serialize backward compatibility</strong></td>
<td>CGXStyle::ImplementSerializeOG5Compatible(): Check this if you want to read serialized documents created with OG 5.0 or earlier.</td>
</tr>
</tbody>
</table>

### 6.4.1.12 Window Class Registration

Choose classes that you want to use as custom windows in dialog templates. If you use **SubclassDlgItem**, you don’t need to register the window classes and you can turn off the following components.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register CGXTabWnd</strong></td>
<td>Use <strong>CGXTabWnd</strong> as custom window in dialog templates. If you use <strong>SubclassDlgItem</strong> there is no need to register the window classes.</td>
</tr>
</tbody>
</table>
6.4.2 Library Linking

This branch is only available for the Build Wizard.

Choose components to be linked into the libraries.

Excluding features from being linked into the library and using them in your projects will cause unresolved externals.

6.4.2.1 WndView

Use grid as CWnd or CView.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGXGridView</td>
<td>Force CGXGridView to be linked into the library.</td>
</tr>
<tr>
<td>CGXGridHandleView</td>
<td>Force CGXGridHandleView to be linked into the library.</td>
</tr>
<tr>
<td>CGXGridWnd</td>
<td>Force CGXGridWnd to be linked into the library.</td>
</tr>
<tr>
<td>CGXGridHandleWnd</td>
<td>Force CGXGridHandleWnd to be linked into the library.</td>
</tr>
</tbody>
</table>

6.4.2.2 Ole Drag and Drop

Use grid as OLE Drop Target or data source.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OleDropTarget</td>
<td>Force the OleDropTarget module to be linked into the library if you want to register grids as OLE drop target. Call EnableOleDropTarget() from OnInitialUpdate() to use this feature in your grid(s).</td>
</tr>
<tr>
<td>OleDataSource</td>
<td>Force the OleDataSource module to be linked into the library if you want to use grids as OLE data source. Call EnableOleDataSource() from OnInitialUpdate() to use this feature in your grid(s).</td>
</tr>
</tbody>
</table>
### 6.4.2.3 Nonstandard Cell types

Cell types that need to be manually registered in your grid. The control factory can’t instantiate these cell types because additional information (resource id for bitmap or pointer to a `CWnd` object) is necessary when constructing the cell type object.

<table>
<thead>
<tr>
<th>Cell Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CWnd Wrapper</strong></td>
<td>Force <code>CGXWndWrapper</code> class to be linked into the library. No default cell type ID. The gridapp sample (<code>gridsvw5.cpp</code> and <code>gridsvw6.cpp</code>) demonstrates how to use this cell type.</td>
</tr>
<tr>
<td><strong>Bitmap Button</strong></td>
<td>Force <code>CGXBitmapButton</code> class to be linked into the library. No default cell type ID. The gridapp sample (<code>gridsvw3.cpp</code>, Welcome page) demonstrates how to use this class.</td>
</tr>
</tbody>
</table>

The code for gridapp sample is located at `<stringray-installdir>\Samples\Grid\General\GridApp`.

### 6.4.2.4 Visual Components

IntelliMouse and scrolltips support.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MouseWheel</strong></td>
<td>Force <code>CGXMouseWheelPlugin</code> class to be linked into the library. IntelliMouse support: wheel scrolling, zooming. No support for panning. Call <code>EnableMouseWheel()</code> in your <code>OnInitialUpdate()</code> routine to use this feature in grids.</td>
</tr>
<tr>
<td><strong>IntelliMouse</strong></td>
<td>Force <code>CGX IntelliMousePlugin</code> class to be linked into the library. IntelliMouse support: wheel scrolling, zooming and panning. Call <code>EnableIntelliMouse()</code> in your <code>OnInitialUpdate()</code> routine to use this feature in grids.</td>
</tr>
<tr>
<td><strong>ScrollTips</strong></td>
<td>Force the <code>CGXScrollTip</code> class and <code>CGXScrollTipPlugin</code> class to be linked into the library. Scrolltips give feedback about row and column position when user is dragging the scroll thumb. Call <code>EnableScrollTips()</code> in your <code>OnInitialUpdate()</code> routine to use this feature in grids. Another alternative for using scroll tips in your windows is to override <code>OnHScroll()</code> and <code>OnVScroll()</code> and forward these messages to an embedded <code>CGXScrollTip</code> variable. See the <code>CGXScrollTip</code> class for more details.</td>
</tr>
</tbody>
</table>
### 6.4.2.5 Formula Support

Formula Engine and worksheet functions.

<table>
<thead>
<tr>
<th><strong>Formula Engine</strong></th>
<th>Force formula engine code to be linked into the library. Call <code>EnableFormulaEngine()</code> in your <code>OnInitialUpdate()</code> routine to enable formula support for a grid.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>All Worksheet Functions</strong></td>
<td>Force all worksheet functions to be linked into the library. Call <code>GXSetDefaultWorksheetFunctions()</code> in your <code>InitInstance()</code> method to load all worksheet functions into your application.</td>
</tr>
</tbody>
</table>

### 6.4.2.6 Dialogs

Pre-built dialogs for customizing the grid appearance or changing cell formatting.

<table>
<thead>
<tr>
<th><strong>Property Settings</strong></th>
<th>Force Property settings, Page setup, Header and Footer dialog to be linked into the library. See <code>mygridvw.cpp</code> in gridapp for an example of displaying property dialogs.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Style Sheet</strong></td>
<td>Force Base Styles dialog and <code>CGXStyleSheet</code> to be linked into the library. See <code>mygridvw.cpp</code> in gridapp for an example of using <code>CGXStyleSheet</code> and <code>CGXStylesDialog</code>.</td>
</tr>
</tbody>
</table>

### 6.4.2.7 Frame Window Classes

Check the frame window classes you want to use in your projects.

<table>
<thead>
<tr>
<th><strong>CGXTabWnd</strong></th>
<th>Force <code>CGXTabWnd</code> and <code>CGXTabBeam</code> class to be linked into the library.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CGXRecordInfoWindow</strong></td>
<td>Force <code>CGXRecordInfoWindow</code> class to be linked into the library.</td>
</tr>
<tr>
<td><strong>CGXRecordInfoSplitterWindow</strong></td>
<td>Force <code>CGXRecordInfoSplitterWindow</code> class to be linked into the library.</td>
</tr>
<tr>
<td><strong>CGXSpliterWnd</strong></td>
<td>Force <code>CGXSpliterWnd</code> class to be linked into the library.</td>
</tr>
</tbody>
</table>

### 6.4.2.8 Collection Classes

Check those collection classes you want to use in your projects.

<table>
<thead>
<tr>
<th><strong>CGXCollMapDWordToPtr</strong></th>
<th>Force <code>CGXCollMapDWordToPtr</code> class to be linked into the library.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CGXCollMapDWordToString</strong></td>
<td>Force <code>CGXCollMapDWordToString</code> class to be linked into the library.</td>
</tr>
</tbody>
</table>
### 6.4.2.9 Basic Classes

Check those classes that you want to use in your projects. Only advanced user should exclude any of these classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGXCollMapStringToHandle</td>
<td>Force <code>CGXCollMapStringToHandle</code> class to be linked into the library.</td>
</tr>
<tr>
<td>CGXCollMapStringToWord</td>
<td>Force <code>CGXCollMapStringToWord</code> class to be linked into the library.</td>
</tr>
<tr>
<td>CGXGridHint</td>
<td>Force <code>CGXGridHint</code> class to be linked into the library.</td>
</tr>
<tr>
<td>CCGXUserAttribute</td>
<td>Force <code>CCGXUserAttribute</code> class to be linked into the library.</td>
</tr>
</tbody>
</table>

### 6.4.2.10 BrowserGrid

Use the browser grid as CWnd or CView.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGXBrowserWnd</td>
<td>Force <code>CGXBrowserWnd</code> (and <code>CGXBrowserGrid</code>) class to be linked into the library.</td>
</tr>
<tr>
<td>CGXBrowserView</td>
<td>Force <code>CGXBrowserView</code> (and <code>CGXBrowserGrid</code>) class to be linked into the library.</td>
</tr>
</tbody>
</table>

### 6.4.2.11 ADO Grid

Link ADO classes into the library.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGXAdoRecordView</td>
<td>Force <code>CGXAdoRecordView</code> (and <code>CGXAdoGrid</code>) class to be linked into the library.</td>
</tr>
<tr>
<td>CGXAdoRecordWnd</td>
<td>Force <code>CGXAdoRecordWnd</code> (and <code>CGXAdoGrid</code>) class to be linked into the library.</td>
</tr>
<tr>
<td>Attach Foreign ADO Table</td>
<td>Allow attaching foreign table to a column.</td>
</tr>
</tbody>
</table>

### 6.4.2.12 ODBC Grid

Link ODBC classes into the library.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGXRecordView</td>
<td>Force <code>CGXRecordView</code> (and <code>CGXODBCGrid</code>) class to be linked into the library.</td>
</tr>
<tr>
<td>CGXRecordWnd</td>
<td>Force <code>CGXRecordWnd</code> (and <code>CGXODBCGrid</code>) class to be linked into the library.</td>
</tr>
</tbody>
</table>
6.4.3 Resources

The Build Wizard lets you choose resources to be included from gxres.rc.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CGXDynamicRecordset</td>
<td>Force CGXDynamicRecordset class to be linked into the library.</td>
</tr>
<tr>
<td>Attach Foreign ODBC Table</td>
<td>Allow attaching foreign table to a column.</td>
</tr>
</tbody>
</table>

6.4.4 A Note On the Implementation

To create the smallest possible grid applications, we leveraged the C++ concept of abstract base classes. Nearly all of the previously described features are implemented through a concrete C++ class that is derived from an abstract base class. Using this concept eliminated any dependencies on the concrete implementation of a specific feature. If you look at the Objective Grid header files (especially gxabstr.h) you see many abstract base classes. All these abstract base classes have been given the name of the feature plus the suffix “Imp”.

You might be worrying that this new concept and the many changes in Objective Grid source code might break your existing applications. We are glad to say this is not the case at all. Objective Grid is fully source-code compatible with earlier versions. All your virtual overrides will still be called the same way they have been called in earlier versions of Objective Grid. Even more, we strongly recommend that you continue to use Objective Grid the same way as before. That means, if you want to customize the behavior of a specific feature override only those virtual methods provided by CGXGridCore (or other related classes). Do not subclass the concrete or abstract base class for that feature. We consider all the abstract base classes and their concrete implementations implementation-specific code. They access protected members in CGXGridCore and the classes themselves are subject to be changed in future grid versions. Don’t rely on source code-compatibility if you subclass any of the abstract base classes with the “Imp” suffix. We therefore did also not consider providing many of the concrete class implementation in header files.
Take the “resize cells” feature as example. The abstract base class is
CGXAbstractGridUserResizeCellsImp. Its declaration can be found in Stingray Studio
<version>\include\grid\gxabstr.h. The concrete implementation
CGXGridUserResizeCellsImp can be found in Stingray Studio
<version>\src\grid\gxsizeu.cpp. CGXGridUserResizeCellsImp is not declared in any Objective Grid header file and therefore not accessible by your application.

If you want to customize the behavior of "resize cells" there are virtual methods in CGXGridCore that you can override: OnStartTracking, OnMoveTracking, OnCancelTracking and OnEndTracking. These methods are still called in the same way as with earlier grid versions even though their concrete implementation for performing the action has moved to
CGXGridUserResizeCellsImp.
Chapter 7

1stGrid Tutorial

7.1 Introduction

In this section, you will learn how to create a small application that uses Objective Grid. The name of that application is 1stGrid and it uses Objective Grid as a view.

The tutorial takes you through the following major steps:

- Step 1 - Generate a skeleton application with a CGXGridView derived view class.
- Step 2 - Add Edit | Find, Edit | Redo and View | Properties menu handlers for the grid view.
- Step 3 - Add document/view serialization and splitter view support to the application.

The code for Steps 1 and 3 is located at <stringray-installdir>\Samples\Grid\Tutorial\1stGrid\[Step1|Step3]. The project filename is 1stGrid*.vcproj. Code for Step 2 is included in the Step 3 project. If you’re simply reading along with the tutorial without adding code, you can still compile 1stGrid at each step to see what it looks like and how it behaves. For each step, go to the specific subdirectory and open the solution.

The code for Step 2 of the tutorial is available on the Rogue Wave Web site, as described in Section 3.6.1, “Location of Sample Code,” of the Stingray Studio Getting Started Guide.

The following sections show step by step how to develop this application.
Figure 53 – Final 1stGrid application in action
7.2 1stGrid - Step 1

Step 1 will guide you through the process of creating a skeleton application using a CGXGridView-derived view class.

7.2.1 Create a New 1stGrid Project

1. Start up Microsoft Visual Studio.
2. Select File | New | Project from the menu.
3. The New dialog box appears.
4. In the dialog, select the Visual C++ Projects folder.
5. Enter 1stGrid for the Project name and <stingray-installdir>/Samples/Grid/Tutorial/1stgrid/1stgrid for the Location.
6. Select MFC Application from the Templates list box.

![Figure 54 – Visual C++| MFC New Project dialog]

7. Click OK.
7.2.2 1stGrid AppWizard Options

1. The MFC Application Wizard dialog box will appear.
   You can scan through the settings by clicking the links, or click Finish at any time to create the project.

2. Open Application Type page, then:
   - Uncheck Tabbed documents.
   - Select MFC standard for Project style.
   - Uncheck Enable visual style switching.
   - Select Windows Native/Default for Visual style and color.
   - Uncheck Use Unicode libraries.
   - Click Finish.

7.2.3 1stGrid Project Settings

This step specifies that the application should use Objective Grid as a shared library. You should skip this step if you want to link statically to the Objective Grid libraries.

1. Select View | Solution Explorer from the menu.
2. The Solution Explorer docking view appears.
3. Right click on 1stgrid and select Properties.
4. Expand the C/C++ node and select Preprocessor. Add _GXDLL to the Preprocessor definitions. This step indicates that the application will link dynamically to Objective Grid. Do not specify _GXDLL if you wish to link to Objective Grid as a static library. _GXDLL is similar to the _AFXDLL preprocessor in that _AFXDLL specifies that the application will link to MFC as a dynamic library.
5. Click OK.

7.2.4 Modify 1stGrid's stdafx.h

1. Open stdafx.h.

2. Add the following lines:
   - Add `#include <grid\GXALL.h>` at the end of the includes in this file.
     
     `GXALL.h` is the core Objective Grid header file. Once this file is added, all required classes are available to you inside your application. This is similar to including `afx*.h`.
   - Add `#include <ManifestDefs.h>` again at the end of the includes.
     
     This file facilitates the inclusion of manifest definitions for Windows Visual Styles.
   - Add `#include <SupportedPlatforms.h>` at the top of the includes.
     
     The conditional platform information is displayed in the application's output window. The output information is helpful when developing and deploying across one or more platforms.
3. Save and close stdafx.h.

### 7.2.5 Include Objective Grid Resources in 1stGrid

Including the Objective Grid resource header file in your application's resource script gives your application access to the Objective Grid Resources. When linking statically to the Objective Grid libraries, it is also necessary to include the Objective Grid resource script. The following steps demonstrate how to add the Objective Grid resource files to your project.

1. Open the **Resource View** in Microsoft Visual Studio.

2. Select the root of the **Resource View** tree (1stGrid.rc) and click with the right button on your mouse.

   A context menu pops up.
3. Choose **Resource Includes**.

4. The **Resource Includes** dialog box appears.

5. Add the following line to the **Read-only symbol directives** box.
   
   ```
   #include "grid\gxresrc.h"
   ```
6. Add the following line to the end of the **Compile-time directives** box.

```cpp
#include "grid\gxres.rc"
```

*Skip this step if your application is linking to Objective Grid as a shared library (DLL). Including the following file will not upset the function of your application, but it will cause unnecessary bloat to the final application file size. The Objective Grid resources are already included in Objective Grid DLLs and therefore do not need to be included in the application.*

7. Click **OK**.

8. A message box pops up warning you that the "Directive text will be written verbatim into your resource script and may render it un compilable." This is fine. You can safely choose **OK**.


### 7.2.6 Modify the 1stGrid Application Implementation File

`GXInit()` initializes resources and variables used by Objective Grid. It should be called from your application’s `InitInstance()` to be sure Objective Grid is initialized before calling any Objective Grid functions. For more information about `GXInit()`, please refer to the Objective Grid **Class Reference**.

1. Open `1stgrid.cpp` and search for `InitInstance()`.
2. Add the following two lines to the beginning of `InitInstance()`:

   ```cpp
   // This call will initialize the grid library
   GXInit();
   ```
3. Save and close `1stGrid.cpp`.

### 7.2.7 Modify 1stGrid View's Header File

In this step, the application’s view class is changed from a `CView` derivative to a `CGXGridView` derivative.

1. Open `1stGridView.h`.
2. Change the derivation of the view by replacing all occurrences of `CView` with `CGXGridView`.
3. Save and close `1stGridView.h`. 
7.2.8 Modify 1stGrid View's Implementation File

This step modifies the `OnDraw()`, `OnBeginPrint()`, and `OnEndPrinting()` methods to call the `CGXGridView` base class versions. This is essential for the Objective Grid view to work properly.

**Note:** You need to uncomment the parameter names for `OnDraw()`, `OnBeginPrinting()` and `OnEndPrinting()`.

1. Open `1stGridView.cpp`.
2. Replace all occurrences of `CView` with `CGXGridView`.
3. Modify `OnDraw()` to call the `CGXGridView` implementation by replacing the `// TODO: ...` line with:

   ```cpp
   // Delegate drawing to the base class
   CGXGridView::OnDraw(pDC);
   ```
4. Modify `OnBeginPrinting(...)` to call the `CGXGridView` implementation by replacing the `// TODO: ...` line with:

   ```cpp
   CGXGridView::OnBeginPrinting(pDC, pInfo);
   ```
5. Modify `OnEndPrinting(...)` to call the `CGXGridView` implementation by replacing the `// TODO: ...` line with:

   ```cpp
   CGXGridView::OnEndPrinting(pDC, pInfo);
   ```
6. Save `1stGridView.cpp`.

7.2.9 Override `CMy1stGridView::OnInitialUpdate()`

`OnInitialUpdate()` is the primary location for initializing the grid object.

1. Select `View | Class View` from the menu.
2. Select the `CMy1stGridView` node under `1stGrid`.
3. Right click on `CMy1stGridView` and select `Properties`.
4. Click `Overrides` in the properties box.
5. Select `<Add> OnInitialUpdate` from the `OnInitialUpdate` field pull down combo box.
6. The `OnInitialUpdate()` member function is added to the `CMy1stGridView` class.
7. Replace the existing `OnInitialUpdate()` function with the following:

   ```cpp
   void CMy1stGridView::OnInitialUpdate()
   {
       // Call the base class OnInitialUpdate. This call is
       // essential for grid specific initialization (In
       // contrast to GXInit which performs application specific
       // initialization)
       CGXGridView::OnInitialUpdate();
   }
   ```
// Disable Undo mechanism for the following commands.
// Objective Grid has a built in Undo/Redo architecture
// that can be disabled/enabled with this call.
// We disable it here so that users are not able to
// rollback changes that we make in code.
GetParam()->EnableUndo(FALSE);

// Initialize grid with 30 rows and 10 columns
SetRowCount(30); // 30 rows
SetColCount(10); // 10 columns

// Display "Hello world" in the first two cells. A
// range in the grid is simply an object that is a range
// of cells defined by four coordinates (top, left,
// bottom, right)
SetValueRange(CGXRange(1, 1), "Hello");

// SetValueRange() is the call that is to be used to
// set simple text to grid cells.
SetValueRange(CGXRange(2, 1), "world");

// "world" should be displayed right aligned.
// SetStyleRange() is to be used to set other formatting,
// control type, etc... along with the value
SetStyleRange(CGXRange(2, 1),
CGXStyle().SetHorizontalAlignment(DT_RIGHT));

// Re-enable undo mechanism
GetParam()->EnableUndo(TRUE);
}

8. Save and close 1stGridView.cpp.

### 7.2.10 Compile and Run 1stGrid - Step 1

If you are using Visual Studio 2010, you must add the user property sheet that defines paths needed by the project.

1. Open the Property Manager and expand the 1stGrid node to see the available build configurations.

2. Right click on Debug | Win32 (or whatever configuration you want to build with), and select Add Existing Property Sheet ...
3. Navigate to the file `SS-Win32-PropSheets.props` in `<stingray-installdir>/src` and add this file to the project.

To complete this first step in the tutorial:

1. Compile the application and run it.
2. Interact with the grid object in the view.

If the tutorial is run under Windows XP, the VC++ Application Wizard, by default, enables the XP Visual Styles. The Application Wizard does this by including a manifest resource pointing the application to `COMCTRL32.DLL` ver 6.0. If you would like the grid to work with XP Visual Styles, or any of the other supported Visual Styles, please refer to Section 4.3.3.1, “Vista Look and Feel.”
Figure 60 – 1stGrid-Step 1 in action

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ho</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>word</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7.3 1stGrid - Step 2

7.3.1 Modify 1stGrid View's Menu Resource

This step adds new menu items for Redo, Find, and Properties.

1. Open the 1stGrid project resources (1stGrid.rc).

   (To open resources in Visual Studio, select the View | Resource View from the menu.)

   Figure 61 – Visual Studio Resource View

   2. Expand Menu in the resource tree and double click IDR_My1stGridTYPE.

   3. The IDR_My1stGridTYPE menu resource appears in a resource edit view.
4. Click **Edit** in the menu resource.

5. Click the Separator after **Undo**.

6. Right click and select **Insert New**.

7. A blank menu item is inserted.

8. Right click on the newly inserted item and select **Properties**.

9. Type `&Redo\tCtrl+R` in the **Caption** field and **ID_EDIT_REDO** in the ID field.

10. Press **Enter**.

11. Click the last empty menu item under **Edit** in the menu resource edit view.

12. Right click and select **Insert Separator**.

13. Select the last empty menu item under Edit in the menu resource edit view.

14. Right click the empty menu item and select **Insert New**.
15. Right click the newly inserted item and select **Properties**.

16. Type &Find... Alt+F3 in the **Caption** field. The ID for the menu item should be ID_EDIT_FIND.

   Figure 64 – Find Menu Item Properties dialog - Visual Studio

17. Click **View** in the menu resource.

18. Select the last empty menu item under **View** in the menu resource edit view.

19. Right click and select **Insert New**.

20. Type &Properties... in the **Caption** box.

21. Set the ID for this item to ID_VIEW_PROPERTIES, then type **Change Display Properties** in the **Prompt** box.
7.3.2 Map the 1stGrid Menu Messages

This step demonstrates how to map the new menu items from the previous step to message handlers in the view class. Objective Grid already provides handlers for Edit | Undo, Edit | Redo and Edit | Find in the CGXGridView class. Only the View | Properties menu needs to be mapped in this step.

The code provided in this section is more advanced than code presented in other sections. This is to give you a feel for the object oriented nature of Objective Grid. It also demonstrates the ease with which Objective Grid can be customized.

To understand the code below it may help to understand that the properties object is an instance of CGXProperties. CGXProperties stores display and print properties for the grid object. The property object is contained in the parameter object. Recall from the Design Overview section that the...
The parameter object is analogous in some ways to the MFC \textit{CDocument}. The parameter object is the data storehouse for the grid object in much the same way as the \textit{CDocument} derived classes are for MFC applications. Also recall that parameter objects can be stored in MFC documents.

1. Open the \textit{View} \textbar Class View menu.
2. Right click \texttt{CMy1stGridView} and select \textbf{Properties}.
3. Select the \textbf{Events} icon.
4. Expand the \texttt{ID\_VIEW\_PROPERTIES} node.
5. Click on the \textbf{COMMAND} field and select \texttt{<Add> OnViewProperties} from the combo box.

The source file is opened in the editor and the \texttt{OnViewProperties} function is added.

\textbf{Figure 68 – Adding the Properties menu handler}

\begin{verbatim}
Properties

\begin{tabular}{|c|}
\hline
\texttt{CMy1stGridView} \texttt{VCCodeClass} \\
\hline
\hline
\end{tabular}
\end{verbatim}
6. Replace the following method in `lstgrvw.cpp`:

```cpp
void C1stGridView::OnViewProperties()
{
    BOOL bSaveDefault = FALSE;
    // Get the properties object from the parameter object
    CGXProperties* pPropertyObj =
        GetParam()->GetProperties();
    ASSERT(pPropertyObj->IsKindOf(RUNTIME_CLASS(CGXProperties)));

    CGXProperties* pNewSettings = new CGXProperties;
    *pNewSettings = *pPropertyObj;

    // This is a pre-built dialog that can modify a
    // properties object based on user input. We simply
    // create a copy of our existing parameter object and
    // pass it on to this dialog.
    CGXDisplayPropertiesDialog dlg(pNewSettings,
        GetParam()->GetStylesMap(),
        &bSaveDefault,
        FALSE);

    int result = dlg.DoModal();
    if (result == IDOK)
    {
        // Allow the grid to use the changed properties object
        ChangeProperties(*pNewSettings);

        // Write new information to the profile if required.
        if (bSaveDefault)
            pPropertyObj->WriteProfile();
    }
    delete pNewSettings;
}
```

7.3.3 Map the 1stGrid Accelerator Keys

This step maps accelerator keys to the new menu items. Note the fact that this editing can be done in two ways: directly in the Accelerator editor window, or in the associated Properties dialog.

1. Open `1stgrid.rc` (View | Resource View).
2. Open Accelerator and double-click IDR_MAINFRAME.
3. The Accelerator editor window appears.
4. Double click the last empty line in the Accelerator table. The Properties dialog box appears.

5. In the Accelerator table, select ID_EDIT_FIND in the combo box for the ID column.

6. Click in theModifiers column, and select Alt from the combo box. This causes Alt to be set to True in the Properties dialog, and Shift and Ctrl to be set to False.

7. Click in the Key column and select VK_F3 from the combo box.

Figure 70 – Accelerator Properties dialog -- Find

8. Press Enter.
9. Double-click on the last empty line of the Accelerator table. The **Properties** dialog box appears again.

10. In the Accelerator table, select **ID_EDIT_REDO** in the **ID** combo box.

11. In the **Behavior** group of the **Properties** dialog, set **Ctrl** to **False**, and **Shift** and **Alt** to **True**.

12. Type **R** in the **Key** text field.

    ![Figure 71 – Accelerator Properties dialog -- Redo](image)

Figure 71 – Accelerator Properties dialog -- Redo

<table>
<thead>
<tr>
<th><strong>Properties</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Accelerator Editor</strong></td>
</tr>
<tr>
<td><strong>Behavior</strong></td>
</tr>
<tr>
<td>Alt</td>
</tr>
<tr>
<td>Ctrl</td>
</tr>
<tr>
<td><strong>Key</strong></td>
</tr>
<tr>
<td>Shift</td>
</tr>
<tr>
<td>Type</td>
</tr>
<tr>
<td><strong>Misc</strong></td>
</tr>
<tr>
<td>(Name)</td>
</tr>
<tr>
<td>ID</td>
</tr>
</tbody>
</table>

13. Press **Enter**.


### 7.3.4 More Formatting

This step initializes the grid view. You can find a complete listing of the code changes for this step in `1stGrid\Step2` sample available from the Rogue Wave Web site, as explained in Section 3.6.1, “Location of Sample Code,” in the *Stingray Studio Getting Started Guide*. Also refer to the comments in the code below for an explanation of its purpose.

1. Open `1stGridView.cpp`.

2. Add the following code to `OnInitialUpdate()` immediately before the line with `GetParam() -> EnableUndo(TRUE)`.

   ```cpp
   // Make a big headline from cell (4,2) to cell (6,5). You can use
   // this call to cover several cells as one cell.
   SetCoveredCellsRowCol(4, 2, 6, 5);

   // Display "Welcome" in the covered
   // cell with some formattings.
   // Notice that some of the calls
   // made on the style object take
   // other objects (CGXFont etc.,)
   ```
SetStyleRange(CGXRange(4,2),
  CGXStyle()
  .SetValue("Welcome") // Text to be displayed
  .SetControl(GX_IDS_CTRL_STATIC) // Static Text, no editing
  .SetVerticalAlignment(DT_VCENTER) // Center vertical
  .SetHorizontalAlignment(DT_CENTER) // Center horizontal
  .SetFont(CGXFont() // Some decorative font
    .SetFaceName(_T("Bookman Old Style"))
    .SetSize(28)
    .SetBold(TRUE)
  )
  .SetInterior(CGXBrush()
    .SetColor(RGB(192,192,192)) // light gray
  )
  .SetDraw3dFrame(gxFrameRaised) // With raised effect.
  .SetEnabled(FALSE) // User cannot click on it
);

// change the default font for all cells
// The call to
//
// CGXStyle()
//  .SetValue("Welcome") // Text to be displayed
//  .SetControl(GX_IDS_CTRL_STATIC)
//
// This is possible because each of these calls
// returns a reference to the object itself. For example
// SetValue(...) returns a reference to CGXStyle (CGXStyle&)
// and hence the next call to SetControl is possible and so on.

// The StandardStyle is the style for the entire grid (this will be
// used in the absence of any other styles (that are cell specific)
ChangeStandardStyle(CGXStyle())
  .SetFont(CGXFont().SetSize(10)) // Bigger font
  .SetAutoSize(TRUE) // Cells will grow automatically
    // when user enters large text.
  .SetWrapText(TRUE) // Wrap text when it does not
    // fit into a single line.
  .SetAllowEnter(TRUE) // When user presses <Enter>
    // a new line will be inserted
    // into the text of the cell.
  , gxOverride );

7.3.5 Compile and Run 1stGrid - Step 2

At this point, you can compile and run the 1stGrid application.

1. Use the **Edit|Find** menu command to type in text, undo and redo your changes and search text.

2. Compile the application and run it.
You can copy menu items from gridapp.rc in `<stringray-installdir>`\Samples\Grid\General\GridApp to your application. Open gridapp.rc and take a look at the IDR_GRIDAPTYPE menu resource and copy the menu items you would like to support in your own application. The file mygridvw.cpp in `\samples\grid\general\gridapp` contains code that you may want to include in your view class for menu handlers.
7.4 1stGrid - Step 3

Objective Grid stores all document-specific attributes in the parameter object. When you store the parameter object in the document, all grid views connected to the document can access the same data. In `OnInitialUpdate()` you will see how to pass a pointer to the parameter object to the grid.

### 7.4.1 Modify the 1stGrid Document's Header File

This step adds a `CGXParam` object to the application’s `CDocument` class. This enables all grid views to share the same parameter object.

1. Open `1stgrdoc.h`
2. Add a public member variable to the attributes section of `C1stGridDoc`:
   ```cpp
   CGXGridParam* m_pParam;
   ```

### 7.4.2 Modify the 1stGrid Document's Implementation File

It is good practice to add clean-up code in `OnNewDocument()`. This makes it easier to use the document in a single document interface (SDI) application later.

1. Open `1stgrdoc.cpp`
2. Replace the `C1stGridDoc` constructor with the following version:
   ```cpp
   C1stGridDoc::C1stGridDoc()
   {
       m_pParam = NULL;
   }
   ```
3. Replace the `C1stGridDoc` destructor with the following version:
   ```cpp
   C1stGridDoc::~C1stGridDoc()
   {
       delete m_pParam;
   }
   ```
4. Change the `C1stGridDoc::Serialize()` method as follows:
   ```cpp
   void C1stGridDoc::Serialize(CArchive& ar)
   {
       if (ar.IsStoring())
       {
           // Store parameter-object
           ar << m_pParam;
       }
       else
       {
           // Create parameter-object
           ar >> m_pParam;
       }
   }
   ```
5. Modify `C1stGridDoc::OnNewDocument()` as follows:

```cpp
BOOL C1stGridDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    // Clean up parameter-object
    delete m_pParam;
    m_pParam = NULL;

    return TRUE;
}
```

### 7.4.3 Modify the 1stGrid View's Implementation File

In your `OnInitialUpdate()` method you need to pass the pointer to the parameter object to the grid object. If the first view for a document is opened, the parameter object must be allocated and the grid data initialized. If the end user opens additional views for the same document, no initialization is necessary. The grid data is already available in the parameter object.

`bFirstView = TRUE` indicates that this is the first view connected to the document and therefore the data needs to be initialized.

Next, construct the parameter object and pass the pointer to the grid view.

Your existing initialization code should be bracketed with an if statement where you should check if `bFirstView` is `TRUE`.

1. Add the following code to the beginning of `C1stGridView::OnInitialUpdate()`:

```cpp
BOOL bFirstView = FALSE;
if (GetDocument()->m_pParam == NULL)
{
    bFirstView = TRUE;
    GetDocument()->m_pParam = new CGXGridParam;
}
SetParam(GetDocument()->m_pParam, FALSE);
```

2. Enclose the existing initialization code with an if statement that checks for `bFirstView = TRUE`:

```cpp
if (bFirstView)
{
    // ... (Existing init code)
}
```

3. Please refer to `CMy1stGridView::OnInitialUpdate()` in `<stingray-installdir>\Samples\Grid\Tutorial\1stGrid\Step3` for a complete listing of these changes.

At this point, you can compile and run with these modifications. You can now serialize data into a document and open new views for the same data using Window | New Window. In the next section we will add support for splitting the view.
7.4.4 Add a Splitter Frame Class for 1stGrid

Next, override the `OnCreateClient()` method:

1. Select `View | Class View`.
2. Right click on `1stGrid` and select `Add`, then `Class`.
   The Add Class dialog box appears.
3. Select `MFC` in the `Templates` section and `MFC Class` in the `Categories` section. Then click the `Add` button.
   The `MFC Add Class Wizard` appears.
4. In the `MFC Add Class Wizard`, enter `CSplitterMDIChildWnd` for the `Name`.
5. Specify `CMDIChildWnd` as the `Base class`.
6. Click `Finish` on the `MFC Add Class Wizard` dialog.
7. In the `Class View`, right click on the newly generated class and select `Properties`.
8. Click on the `Overrides` icon.
9. Select the `OnCreateClient` event.
10. Select `<Add> OnCreateClient`.
11. The function `OnCreateClient(...)` is added to `CSplitterMDIChildWnd`.
12. Edit the code in the file `SplitterMDIChildWnd.cpp`, changing `OnCreateClient(...)` as follows:

   ```cpp
   BOOL CSplitterMDIChildWnd::OnCreateClient(LPCREATESTRUCT /*lpcs*/, CCreateContext* pContext)
   {
       return m_wndSplitter.Create(this,
                                    2, 2,
                                    CSize(10, 10),
                                    pContext);
   }
   ```
13. Open `SplitterMDIChildWnd.h`.
14. Add a member variable to the attributes section of `CSplitterMdiChildWnd`:

   ```cpp
   CSplitterWnd m_wndSplitter;
   ```
15. Close and save `SplitterMDIChildWnd.h` and `SplitterMDIChildWnd.cpp`. 
7.4.5 Change the 1stGrid View to the Splitter Frame Class

1. Open 1stgrid.cpp.

2. After `#include "1stgrvw.h"`, add:

   `#include "SplitterMDIChildWnd.h"

3. Locate:

   ```cpp
   pDocTemplate = new CMultiDocTemplate(
       IDR_1STGRITYPE,
       RUNTIME_CLASS(C1stGridDoc),
       RUNTIME_CLASS(CMDIChildWnd){// standard MDI child frame
       RUNTIME_CLASS(C1stGridView));
   AddDocTemplate(pDocTemplate);
   ```

   and replace `CMDIChildWnd` with `CSplitterMDIChildWnd`:

   ```cpp
   CMultiDocTemplate* pDocTemplate;
   pDocTemplate = new CMultiDocTemplate(
       IDR_1STGRITYPE,
       RUNTIME_CLASS(C1stGridDoc),
       RUNTIME_CLASS(CSplitterMDIChildWnd),
       // splitter child frame
       RUNTIME_CLASS(C1stGridView));
   AddDocTemplate(pDocTemplate);
   ```
7.4.6 Compile and Run 1stGrid - Step 3

At this point, you can compile and run the 1stGrid application.

1. Compile the application and run it.

Figure 73 – 1stGrid-Step 3 in action

Congratulations on completing the 1stGrid tutorial! For more information, try these other tutorials:

- Chapter 8, “DbQuery Tutorial.”
- Chapter 9, “DlgGrid Tutorial.”
- Chapter 10, “VirtGrid Tutorial.”
- Chapter 17, “CSliderCtrl Tutorial.”
- Chapter 18, “BrwsGrid Tutorial.”
8.1 Introduction

In this section you will learn how to create ODBC applications with Objective Grid. The name of the application is DbQuery, and it displays the results of a CRecordset in a grid.

DbQuery shows you how to:

- Generate a skeleton application.
- Add menu handlers for CRecordset.
- Add a record status beam to the scrollbar.

Source code files, project and solution files, and other necessary files for Steps 1 and 2 of the tutorial are obtainable from the Rogue Wave Web site, as described in Section 3.6.1, “Location of Sample Code,” in the Stingray Studio Getting Started Guide. The code for Step 3 is located at <stingray-installdir>\Samples\Grid\Tutorial\DbQuery\Step3.

The tutorial steps do the following:

- Step 1 generates a skeleton application with a CGXRecordView-derived view class.
- Step 2 adds Delete and Requery menu handlers for the CRecordset.
- Step 3 embeds CGXRecordView in a CGXRecordInfoWnd to give it a look similar to Microsoft Access.

If you're simply reading along with the tutorial without adding code, you can still compile DbQuery at each step to see what it looks like and how it behaves. Figure 74 shows what the final application will look like.
Figure 74 – Final DbQuery application in action
8.2 **DbQuery - Step 1**

8.2.1 **Create a New DbQuery Project**

1. Launch Microsoft Visual Studio.
2. Select File | New | Project from the menu.
   The New Project dialog box appears.
   
   ![Visual Studio New Project dialog](image)

3. Enter **DbQuery** for the **Project name** and `<stingray-installdir>`\Samples\Grid\Tutorial\DbQuery\Mygrid for the **Location**.

4. Select **MFC Application** from the list box.
5. Click **OK**.

8.2.2 **DbQuery AppWizard Options**

1. The **MFC Application Wizard** dialog box appears.
2. Select **Multiple documents** for **MFC AppWizard** under **Application type**.
3. Click **Database Support**.

4. Select **Header files only** for MFC Application Wizard under Database Support and Client Type as ODBC.

5. Accept the defaults for all other steps. You can scan through the settings by clicking the links, or click **Finish** to create the project.

### 8.2.3 DbQuery Project Settings

This step specifies that the application should use Objective Grid as a shared library. You should skip this step if you want to link statically to the Objective Grid libraries.

1. Select **Solution Explorer** from the **View** menu.

2. Right click on **DbQuery** and select **Properties** to get the **Property Pages** window.
Figure 77 – Accessing the Project Property Pages

The **Property Pages** window appears.
3. Choose All Configurations in the Configuration: combo box in the top-left corner of the dialog.
4. Select the C/C++ tab and add \_GXDLL to the Preprocessor definitions. This step indicates that the application will link dynamically to Objective Grid. Do not specify \_GXDLL if you wish to link to Objective Grid as a static library. \_GXDLL is similar to the \_AFXDLL preprocessor in that \_AFXDLL specifies that the application will link to MFC as a dynamic library.

Figure 80 – Adding a Preprocessor Definition

<table>
<thead>
<tr>
<th>Preprocessor Definitions</th>
<th>WIN32; WINDOWS _GXDLL &lt;different options&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefine Preprocessor Definitions</td>
<td>No</td>
</tr>
<tr>
<td>Undefine All Preprocessor Definitions</td>
<td>No</td>
</tr>
<tr>
<td>Ignore Standard Include Paths</td>
<td>No</td>
</tr>
<tr>
<td>Preprocess to a file</td>
<td>No</td>
</tr>
<tr>
<td>Preprocess Suppress Line Numbers</td>
<td>No</td>
</tr>
<tr>
<td>Keep Comments</td>
<td>No</td>
</tr>
</tbody>
</table>

5. Click OK.

8.2.4 Modify DbQuery's stdafx.h

Gxall.h is the core Objective Grid header file. Once this file is added, all required classes will be available to you inside your application. This is similar to including \afx*.h. Gxodbc.h is the Objective Grid ODBC header file.

1. Open stdafx.h

2. Add the following lines at the end of the file:
   
   ```
   #include "grid\gxall.h"
   #include "grid\gxodbc.h"
   ```
3. Save and close `stdafx.h`.

### 8.2.5 Include Objective Grid Resources in DbQuery

Including the Objective Grid resource header file in your application’s resource script gives your application access to the Objective Grid Resources. When linking statically to the Objective Grid libraries, it is also necessary to include the Objective Grid resource script. The following steps demonstrate how to add the Objective Grid resource files to your project.

1. Select **Resource View** from the **View** menu.
2. Right-click on the root of the **Resource View** tree (DbQuery.rc).
3. A context menu pops up.

**Figure 82 – Visual Studio Resource View short cut menu**

<table>
<thead>
<tr>
<th>Resource View - DbQuery</th>
<th>Ctrl+X</th>
<th>Ctrl+C</th>
<th>Ctrl+V</th>
<th>Del</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cut</td>
<td>Copy</td>
<td>Paste</td>
<td>Delete</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resource Includes...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ID=</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resource Symbols...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Open</td>
<td>Open Binary Data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Save DbQuery.rc</td>
<td>Ctrl+S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add Resource...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Resource Includes dialog box appears.

Figure 83 – Visual Studio Resource Includes dialog

5. Add the following line to the Read-only symbol directives box.

```
#include "grid\gxresrc.h"
```

6. Add the following line to the end of the Compile-time directives box.

```
#include "grid\gxres.rc"
```

Skip this step if your application is linking to Objective Grid as a shared library (DLL). Including the grid\gxres.rc file will not upset the function of your application, but it will cause unnecessary bloat to the final application file size. The Objective Grid resources are already included in Objective Grid DLLs and therefore do not need to be included in the application.
7. Click **OK**.

8. A message box pops up warning you that the "Directive text will be written verbatim into your resource script and may render it un compilable." This is fine. You can safely choose **OK**.

### 8.2.6 Modify the DbQuery Application Implementation File

**GXInit()** initializes resources and variables used by Objective Grid. It should be called from your application's **InitInstance()** to be sure Objective Grid is initialized before calling any Objective Grid functions. **GXInitODBC()** initializes the ODBC support for Objective Grid. For more information about **GXInit()** or **GXInitODBC()**, please refer to the Objective Grid **Class Reference**.

1. Open **dbquery.cpp** and search for **InitInstance()**.

2. Add the following lines to the beginning of **InitInstance()**:

```cpp
// These calls will initialize the grid library
GXInit();
GXInitODBC();
```

**Figure 84 – Adding Lines to InitInstance**

### 8.2.7 Register a Database

Use the ODBC Data Source Administrator to register the **stdreg32.mdb** database file as **Student Registration**. Section 8.2.8, "Add a CRecordset Derived Class," uses this data source, so you’ll need to register this data source before continuing with the tutorial.

This section gives you step by step instructions. If you don’t need the details, skip this section.

1. Locate the **stdreg32.mdb** file in the `<stringray_installdir>\Samples\Grid\Database` folder. Make sure your copy of **stdreg.mdb** is not read-only.

2. If Windows 2000 is your operating system, start the ODBC Data Source Administrator by double-clicking the Data Sources (ODBC) icon in the Control Panel\Administrative Tools.
If your machine is running a different OS, you may need to search around a bit to locate the icon for the ODBC Data Source Administrator.

3. Click the tab corresponding to the category of data source you want to register. For example, choose the System DSN tab if the data source should be visible to all users on the machine including NT services. Figure 85, Figure 86, and Figure 87 show the difference between the different categories of ODBC data sources.

Figure 85 – User Data Source

An ODBC User data source stores information about how to connect to the indicated data provider. A User data source is only visible to you and can only be used on the current machine.

Figure 86 – System Data Source

An ODBC System data source stores information about how to connect to the indicated data provider. A System data source is visible to all users on this machine, including NT services.

Figure 87 – File Data Source

An ODBC File data source allows you to connect to a data provider. File DSNs can be shared by users who have the same drivers installed.

Figure 88 – System DSN Tab

4. Click the **Add** button.
5. Select the driver that corresponds to Access (*.mdb).

6. Click **Finish**.

7. In the ODBC Microsoft Access Setup dialog, type **Student Registration** as the Data Source Name. (You don’t need to add a description.)

8. Click **Select**.

9. Navigate to the `samples\grid\tutorial` folder and select the name of the database (`stdreg32.mdb`).
10. Click OK.

11. Make sure the Student Registration database shows up correctly.

12. Click OK.

### 8.2.8 Add a CRecordset Derived Class

The following steps use the Student Registration database (stdreg32.mdb). Make sure that you've registered the database as described in Section 8.2.7, "Register a Database."

1. Select Class View from the View menu.
2. Right click on DbQuery and click Add, then Class.
3. Select MFC from the tree node, and MFC ODBC Consumer from the list box.
4. Click the **Data Source...** button and select **Student Registration** under the **Machine Data Source** tab.

**Figure 94 – Selecting a Datasource**
5. Press OK.

The Login dialog box for the selected DataSource appears.

6. Enter the Login and Password if applicable.

7. Click OK to open the Select Database Object dialog box.

8. Select the Student table and press OK.

9. Click OK.

10. Enter CStudentSet in the Class: field, StudentSet.h in the .h file: field, and StudentSet.cpp in the .cpp file: field.

11. Place a checkmark in the Bind all columns check box, and select the Dynaset radio button.

If Snapshot is selected and a Unicode build is used, recordset is read-only.

Figure 95 – Selecting a Table

Figure 96 – Entering Class Information
12. Click **Finish** to generate the necessary files.

13. The wizard may put an error message in the *StudentSet.cpp* file as follows:

```c
#error Security Issue: The connection string may contain a password
// The connection string below may contain plain text
// passwords and/or other sensitive information. Please remove // the #error
after reviewing the connection string for any
// security related issues. You may want to store the password // in some
other form or use a different user authentication.
```

This error message can be removed for the purpose of this tutorial.

### 8.2.9 Modify DbQuery View's Header File

In this step, the application's view class is changed from a *CView* derivative to a *CGXGridView* derivative.

1. Open *DbQueryView.h*.

2. Add the line:
   ```
   class CStudentSet;
   ```
   as a forward declaration before the class declaration and add:
   ```
   CStudentSet* m_pSet;
   ```
   as a member variable to *CDbqueryView*’s attributes section.

   The completed code should look like this:

   ```
   // dbqueryv.h : interface of the CDbqueryView class
   //
   ///                                                                                   
   ///                                                                                   
   ///
   /// class CStudentSet;
   ///
   ///
   class CDbqueryView : public CView
   {
   protected: // create from serialization only
   CDbqueryView();
   GRID_DECLARE_DYNCREATE(CDbqueryView)
   
   // Attributes
   public:
   CDbqueryDoc* GetDocument();
   CStudentSet* m_pSet;
   ...
   ```

3. Change the derivation of the view by replacing all occurrences of *CView* with *
   CGXRecordView*.

### 8.2.10 Modify DbQuery View's Implementation File

1. Open *DbQueryView.cpp*.

2. Update the class definition as follows:

```cpp
class CDbqueryView : public CGXRecordView
```

3. Update the view’s initialization function as follows:

```cpp
CDbqueryView::CDbqueryView(): CDbqueryDoc(), m_pSet(NULL)
```

4. Update the view’s destructor as follows:

```cpp
CDbqueryView::~CDbqueryView()
```
2. Add:

```cpp
#include "studentset.h"
```

The completed code should look like this:

```cpp
// DbQueryView.cpp : implementation of the CDbQueryView class
//
#include "stdafx.h"
#include "DbQuery.h"
#include "DbQueryDoc.h"
#include "DbQueryView.h"
#include "studentset.h"
...```

3. Replace all occurrences of `CView` with `CGXRecordView`.

4. Change the view's constructor and destructor as follows:

```cpp
CDbQueryView::CDbQueryView()
{
    m_pSet = new CStudentSet;
}

CDbQueryView::~CDbQueryView()
{
    delete m_pSet;
}
```

5. Add a new method, `OnGetRecordset()`:

```cpp
CRecordset* CDbQueryView::OnGetRecordset()
{
    return m_pSet;
}
```

Also insert the definition for this method in `DbQueryView.h`:

```cpp
virtual CRecordset* OnGetRecordset();
```

6. Modify `OnDraw()`, `OnBeginPrint()`, and `OnEndPrinting()`:

```cpp
void CDbQueryView::OnDraw(CDC* pDC)
{
    CDbQueryDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    CGXRecordView::OnDraw(pDC);
}

void CDbQueryView::OnBeginPrinting(CDC* pDC,
                                    CPrintInfo* pInfo)
{
    CGXRecordView::OnBeginPrinting(pDC, pInfo);
}

void CDbQueryView::OnEndPrinting(CDC* pDC, CPrintInfo* pInfo)
{
    CGXRecordView::OnEndPrinting(pDC, pInfo);
}
8.2.11 Compile and Run DbQuery - Step 1

At this point, you can compile and run the DbQuery application.

1. Compile the application and run it.

Figure 97 – DbQuery - Step 1 in action

<table>
<thead>
<tr>
<th>StudentID</th>
<th>Name</th>
<th>GradYear</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Smith, Randy</td>
<td>96</td>
</tr>
<tr>
<td>1002</td>
<td>Maples, Alex</td>
<td>95</td>
</tr>
<tr>
<td>1003</td>
<td>Jones, Thomas</td>
<td>95</td>
</tr>
<tr>
<td>1004</td>
<td>Shannon, Eric</td>
<td>96</td>
</tr>
<tr>
<td>1005</td>
<td>Foster, Susan</td>
<td>96</td>
</tr>
<tr>
<td>1006</td>
<td>Jefferson, Nancy</td>
<td>95</td>
</tr>
<tr>
<td>1007</td>
<td>Turner, Bob</td>
<td>96</td>
</tr>
<tr>
<td>1008</td>
<td>Holm, David</td>
<td>95</td>
</tr>
<tr>
<td>1009</td>
<td>Reynolds, Dan</td>
<td>96</td>
</tr>
<tr>
<td>1010</td>
<td>Taylor, Robert</td>
<td>95</td>
</tr>
<tr>
<td>1011</td>
<td>Karr, Dave</td>
<td>95</td>
</tr>
<tr>
<td>1012</td>
<td>Tannant, Tim</td>
<td>95</td>
</tr>
<tr>
<td>1013</td>
<td>Marcus, Susan</td>
<td>95</td>
</tr>
<tr>
<td>1014</td>
<td>Butterfield, Rita</td>
<td>96</td>
</tr>
<tr>
<td>1015</td>
<td>Amon, Craig</td>
<td>95</td>
</tr>
<tr>
<td>1016</td>
<td>Anderson, Sandra</td>
<td>96</td>
</tr>
<tr>
<td>1017</td>
<td>Cooper, Linda</td>
<td>95</td>
</tr>
<tr>
<td>*</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8.3 DbQuery - Step 2

8.3.1 Modify the View's Menu Resource

This step adds new menu items for Record | Delete Records and Record | Requery and removes the Window | New Window menu item.

1. Open the DbQuery project resources (DbQuery.rc).
   (To open resources in Visual Studio, select View | Resource View. Right click on DbQuery.rc and select Properties.)

   Figure 98 – Visual Studio Resource View

   ![Resource View - DbQuery](image)

   2. Expand Menu in the resource tree and double-click IDR_DBQUERTYPE.

3. The IDR_DBQUERTYPE menu resource appears in a resource edit view.
4. Click **View** in the menu resource.
5. Right click and select **Insert New**.
   
   A blank menu item is inserted.
6. Right click on the newly inserted item and select **Properties**.
7. Type **&Record** in the **Caption** field.
8. Press **Enter**.

Figure 100 – Record Menu Item Properties dialog - Visual Studio

**Caption**

Specifies the text of the menu item.
9. Double-click the empty menu item under Record in the menu resource edit view.

Figure 101 – Record Menu Item

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Type Here</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Edit</td>
<td>Record</td>
<td>View</td>
<td>Window</td>
<td>Help</td>
<td></td>
</tr>
</tbody>
</table>

10. Provide a unique ID (**ID_RECORD_DELETERECORDS**) to the menu item.

11. Type **&Delete Records** in the Caption field and Delete selected records\nDelete Records for the Prompt.

12. Press Enter.

Figure 102 – Delete Records Menu Item Properties dialog - Visual Studio

<table>
<thead>
<tr>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Menu Editor</strong> IMenuEd</td>
</tr>
</tbody>
</table>

- **Appearance**
  - **Caption**: &Delete Records
  - Checked: False
  - Enabled: True
  - Grayed: False
  - Popup: False

- **Behavior**
  - Break: None
  - Right Justify: False
  - Right Order: False

- **Misc**
  - (Name): Menu Editor
  - Help: False

13. Click on the empty menu item under Record in the menu resource edit view.

14. Right click and select Insert New.

15. Double click the newly inserted item.

16. Provide a unique ID (**ID_RECORD_REQUERY**) to the menu item.

17. Type **&Requery** in the Caption field and Requery result set.\nRequery for the Prompt.

18. Press Enter.
19. Click **Window** in the menu resource.

   The **Window** menu opens.

20. Click **New Window**.

21. Press `<Delete>` to remove this menu item.

22. Close and save the resource file.

### 8.3.2 Map the DbQuery Menu Messages

1. Select **Class View** from the **View** menu.

2. Select **CDbQueryView** for the **Class name**.

3. Right click **CDbQueryView** and select **Properties**.

4. Click the **Events** icon, then expand **ID_RECORD_REQUERY**.

5. Double-click the **COMMAND** field and pull down the combo box.
6. Select <Add> OnRecordRequery from the drop down list.

The source file is opened in the editor and the OnRecordRequery() function is added.

7. Similarly, add an update handler for the menu item.

Figure 105 – Adding the Requery menu handler

8. Replace the following methods in DbQueryView.cpp:

   /// Code for handling record requery
   void CDbQueryView::OnRecordRequery()
   {
      BOOL bLock = LockUpdate();
      SetTopRow(1);
      MoveCurrentCell(GX_TOPLEFT);
      Requery();
      LockUpdate(bLock);
      Redraw();
   }
void CDbQueryView::OnUpdateRecordRequery(CCmdUI* pCmdUI)
{
    CRecordset* prs = OnGetRecordset();

    // enable if opened and updatable && not last row
    pCmdUI->Enable(prs->IsOpen());
}

9. Search for the message map declaration in DbQueryView.cpp.

10. Add the following menu handlers for the Delete menu item:

    ON_COMMAND(ID_RECORD_DELETERECORDS, 
               CGXRecordView::OnRecordDelete)
    ON_UPDATE_COMMAND_UI(ID_RECORD_DELETERECORDS, 
                          CGXRecordView::OnUpdateRecordDelete)

The completed message map should look like this:

BEGIN_MESSAGE_MAP(CDbQueryView, CGXRecordView)
   //{{AFX_MSG_MAP(CDbQueryView)
    ON_COMMAND(ID_RECORD_REQUERY, OnRecordRequery)
    ON_UPDATE_COMMAND_UI(ID_RECORD_REQUERY, OnUpdateRecordRequery)
   //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CGXRecordView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, 
               CGXRecordView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, 
               CGXRecordView::OnFilePrintPreview)
    ON_COMMAND(ID_RECORD_DELETERECORDS, 
               CGXRecordView::OnRecordDelete)
    ON_UPDATE_COMMAND_UI(ID_RECORD_DELETERECORDS, 
                          CGXRecordView::OnUpdateRecordDelete)
END_MESSAGE_MAP()

8.3.3 Compile and Run DbQuery - Step 2

Now, you can compile the application and run it.
Figure 106 – DbQuery-Step 2 in action
8.4 DbQuery - Step 3

8.4.1 Add a New Frame Class

1. Select Class View from the View menu.
2. Right click on the root node (DbQuery) and select Add, then Class.
3. Select MFC in the left tree, and MFC Class from the right list box.
4. In the MFC Add Class Wizard, enter CRecordStatusMDIChildWnd for the Name.
5. In the MFC Add Class Wizard, specify CMDIChildWnd for the Base class.
6. Click Finish to generate the class files.
7. Right click CRecordStatusMDIChildWnd in Class view, and select Properties.
8. Click on overrides and select OnCreateClient in the list.
9. Pull down the combobox and select <Add> OnCreateClient in the Messages dropdown.
10. Change OnCreateClient() as follows:

```cpp
BOOL CRecordStatusMDIChildWnd::OnCreateClient(
    LPCREATESTRUCT /*lpcs*/,
    CCreateContext* pContext)
{
    return m_wndRecordInfo.Create(this, pContext);
}
```
11. Open RecordStatusMDIChildWnd.h and insert the following into the attributes section of the class:

```cpp
CGXRecordInfoWnd m_wndRecordInfo;
```
12. Save and close RecordStatusMDIChildWnd.h and RecordStatusMDIChildWnd.cpp.

8.4.2 Change the DbQuery View to the New Frame Class

1. Open DbQuery.cpp.
2. Add

```cpp
#include "RecordStatusMDIChildWnd.h"
```
3. Search for:

```cpp
pDocTemplate = new CMultiDocTemplate(
    IDR_DBQUERTYPE,
    RUNTIME_CLASS(CDbQueryDoc),
```
and replace it with:

```cpp
pDocTemplate = new CMultiDocTemplate(
    IDR_DBQUERTYPE,
    RUNTIME_CLASS(CDbQueryDoc),
    RUNTIME_CLASS(CRecordStatusMDIChildWnd),
    RUNTIME_CLASS(CDbQueryView));
AddDocTemplate(pDocTemplate);
```

### 8.4.3 Compile and Run DbQuery - Step 3

Now, you can compile the application and run it.

1. Compile the application and run it.
2. Note the record number at the bottom of the window.

Figure 107 – DbQuery-Step 3 in action
Congratulations on completing the DdQuery tutorial. For more information try these other tutorials:

- Chapter 7, “1stGrid Tutorial.”
- Chapter 8, “DbQuery Tutorial.”
- Chapter 9, “DlgGrid Tutorial.”
- Chapter 10, “VirtGrid Tutorial.”
- Chapter 17, “CSliderCtrl Tutorial.”
- Chapter 18, “BrwsGrid Tutorial.”
9.1 Introduction

In this section, you will learn how to create a small application that uses Objective Grid in a dialog. The name of that application is DlgGrid, and it uses Objective Grid as a window.

The code for this sample is located at `<stringray-installdir>\Samples\Grid\Tutorial\DlgGrid`.

Whenever you are asked to add code, there is a brief explanation of what that code refers to and what it does in the Objective Grid context.

If you’re simply reading along with the tutorial without adding code, you can still compile DlgGrid to see what it looks like and how it behaves.

Figure 108 – DlgGrid application in action.
The following sections show step by step how to develop this application:
9.2 Create a New DlgGrid Project

1. Start up Microsoft Visual Studio.
2. Select File | New | Project from the menu.
   The New Project dialog box appears.
3. In the dialog, choose the Visual C++ node and select MFC.
4. Select MFC Application from the list box.
5. Enter DlgGrid for the Project name and <stringray-installdir>\Samples\Grid\Tutorial\DlgGrid\Mygrid for the Location.

   Figure 109 – Visual Studio New Project dialog

6. Click OK.
9.3 DlgGrid AppWizard Options

1. The MFC Application Wizard dialog box will appear.
2. Choose Dialog Based for the type of application under Application Type.
3. Accept the defaults for all subsequent steps. You can scan through the settings by clicking the links or click Finish at any time to create the project.

9.4 DlgGrid Project Settings

This step specifies that the application should use Objective Grid as a shared library. You should skip this step if you want to link statically to the Objective Grid libraries.

1. Select Solution Explorer from the View menu.
2. Right click DlgGrid and select Properties.

The Property Pages dialog box appears.

Figure 110 – Visual Studio Property Pages window

3. Choose All Configurations in the Configurations: combo box in the top-left corner of the dialog.
4. Expand the C/C++ node and select **Preprocessor**. Add `_GXDLL` to the **Preprocessor definitions**. This step indicates that the application will link dynamically to Objective Grid. Do not specify `_GXDLL` if you wish to link to Objective Grid as a static library. `_GXDLL` is similar to the `_AFXDLL` preprocessor in that `_AFXDLL` specifies that the application will link to MFC as a dynamic library.

5. Click **OK**.

### 9.5 Modify DlgGrid's stdafx.h

_Gxall.h_ is the core Objective Grid header file. Once this file is added all required classes will be available to you inside your application. This is similar to including _afx*.h_.

1. Open _stdafx.h_.

2. Add the following line at the end of the file:

   ```
   #include "grid\gxall.h"
   ```

3. Save and close _stdafx.h_.

9.6 Include Objective Grid Resources in DlgGrid

Including the Objective Grid resource header file in your application’s resource script gives your application access to the Objective Grid Resources. When linking statically to the Objective Grid libraries, it is also necessary to include the Objective Grid resource script. The following steps demonstrate how to add the Objective Grid resource files to your project.

1. Open the Resource View in Microsoft Visual Studio (View | Resource View).

2. Select the root of the Resource View tree (DlgGrid.rc) and click with the right button on your mouse.

3. A context menu pops up.


5. The Resource Includes dialog box appears.
6. Add the following line to the **Read-only symbol directives** box.

   ```
   #include "grid\gxresrc.h"
   ```

7. Add the following line to the end of the **Compile-time directives** box.

   ```
   #include "grid\gxresrc.rc"
   ```

   *Skip this step if your application is linking to Objective Grid as a shared library (DLL). Including the following file will not upset the function of your application, but it will cause unnecessary bloat to the final application file size. The Objective Grid resources are already included in Objective Grid DLLs and therefore do not need to be included in the application.*

8. Click **OK**.

9. A message box pops up warning you that the "Directive text will be written into your resource script and may render it un compilable." This is fine. You can safely choose **OK**.

10. Save **DlgGrid.rc** (press **Ctrl-S**).
9.7 Modify the DlgGrid Application Implementation File

`GXInit()` initializes resources and variables used by Objective Grid. It should be called from your application’s `InitInstance()` to be sure Objective Grid is initialized before calling any Objective Grid functions. For more information about `GXInit()`, please refer to the Objective Grid Class Reference.

1. Open `DlgGrid.cpp` and search for `InitInstance()`.
2. Add the following two lines to the beginning of `InitInstance()`:
   ```cpp
   // This call will initialize the grid library
   GXInit();
   ```
3. Save and close `DlgGrid.cpp`.

9.8 Create a New CGXGridWnd Derived Class

In this step, a grid window class is derived from `CGXGridWnd`. In a later step, this class will be tied to a custom control on the dialog template.

1. Select Class View from the View menu.
2. Right click on DlgGrid and select Add, then Class.
3. Select MFC|MFC Class in the dialog box and click Add.
   The MFC Add Class Wizard dialog box appears.
4. Specify `CMyGridWnd` in the Name: field.
5. Choose CWnd in the Base class: combo box.
6. Click Finish to close the MFC Add ClassWizard.
7. Open `MyGridWnd.h`.
8. Change the base class of `CMyGridWnd` from `CWnd` to `CGXGridWnd`.
9. Save and close `MyGridWnd.h`.
10. Open `MyGridWnd.cpp`.
11. Replace all occurrences of `CWnd` with `CGXGridWnd`.
12. Save and close `MyGridWnd.cpp`. 

246
9.9 Put a Custom Control on the Main Dialog Template

In this step, a user control is drawn on the dialog template as a placeholder for the grid window. The actual link to the grid object will be made at run time during the dialog’s initialization.

1. Open the IDD_DLGGRID_DIALOG template in the resource editor.
2. Remove the "TODO:" static control (IDC_STATIC).
3. Select the custom control from the Toolbox.
4. Draw a custom control on the dialog template.
5. Open the Custom Control Properties dialog box (press Alt-Enter or choose Properties… from the context menu).

![Custom Control Properties dialog](image)

6. Change the ID to IDC_GRID.
7. Specify GXWND in the Class: field. Please note that this specifies the WNDCLASS registered by Objective Grid during initialization and not the name of your C++ CGXGridWnd derived class. **Do not** specify the C++ class name of your grid here, you **must** specify GXWND and GXWND **must** be all caps.
8. Change the Style: field to 0x50b10000. This indicates that both scroll bars and a border should be drawn. To specify border only (no scroll bars) use 0x50810000.
10. Close the IDD_DLGGRID_DIALOG template.
11. Save DlgGrid.rc (press CTRL+S).
9.10 Modify CDlgGridDlg's Declaration

In this step, the application's main dialog class is modified to include member variable of our derived grid class.

1. Open DlgGridDlg.h.

2. Add the following two lines to the public section of the CDlgGridDlg class declaration:

   // Grid window
   CMyGridWnd m_wndGrid;

3. Save and close DlgGridDlg.h

9.11 Include the Derived Grid's Header

1. Open DlgGrid.cpp.

2. Add the following line after #include "stdafx.h"

   #include "MyGridWnd.h"

3. Save and close DlgGrid.cpp.
9.12 Modify CDlgGridDlg::OnInitDialog()

This step modifies OnInitDialog to subclass the user control on the dialog template and to perform some basic grid initialization.

1. Open DlgGridDlg.cpp.

2. Add the following line after #include "stdafx.h":

   #include "MyGridWnd.h"

3. Search for OnInitDialog.

4. Replace the // TODO: line with the following code:

   // Please refer to the MFC documentation on
   // SubclassDlgItem for information on this
   // call. This makes sure that our C++ grid
   // window class subclasses the window that
   // is created with the User Control.
   m_wndGrid.SubclassDlgItem(IDC_GRID, this);

   // Initialize the grid. For CWnd based grids
   // this call is essential. For view based
   // grids this initialization is done in
   // OnInitialUpdate.
   m_wndGrid.Initialize();

   // Sample setup for the grid
   m_wndGrid.GetParam()->EnableUndo(FALSE);
   m_wndGrid.SetRowCount(100);
   m_wndGrid.SetColCount(20);

   m_wndGrid.SetStyleRange(CGXRange(1,1),
                           CGXStyle()
                           .SetValue("Hello")
                           );
   m_wndGrid.SetStyleRange(CGXRange(1,2),
                           CGXStyle()
                           .SetValue("World!")
                           );

   m_wndGrid.GetParam()->EnableUndo(TRUE);

5. Save and close DlgGridDlg.cpp.
9.13 Compile and Run DlgGrid

At this point, you can compile and run the DlgGrid application. You can interact with the grid object in the dialog.

Figure 114 – DlgGrid in action
10.1 Introduction

In this section you will learn how to bind Objective Grid to an external data source using a technique known as virtual binding. A virtually bound grid does not store its data internally, but rather, fetches the data from the data source every time that data is needed. The Objective Grid data bound grids utilize this same technique to bind to ADO, DAO and ODBC data sources. For the purposes of this discussion, we will be binding to a generic data source stored as a member object of the document class. It is important to note that the actual implementation of the data source is not important as long as the necessary data is available.

VirtGrid, the application you build in this tutorial, is a program that displays data dynamically from an external data source and stores changes made by the user back to the data source immediately. It will demonstrate the necessary overrides for the CGXGridWnd/CGXGridView class to accomplish virtual binding.

Code for the tutorial is at `<stringray-installdir>`\Samples\Grid\Tutorial\VirtGrid.

If you’re simply reading along with the tutorial without adding code, you can still compile VirtGrid to see what it looks like and how it behaves.

10.2 VirtGrid Overview

The explanation in this tutorial assumes that you are familiar with the Styles and Drawing and Updating architectures of Objective Grid. If you are not familiar with these architectures, please refer to Chapter 2, “Design Overview.”

For illustration purposes we are going to use an instance of CGXData, m_data, as the external data source to which the grid will be bound. Note that in your application you might have your own data structure or stubs that talk to an external data source. Again, the nature of the external data source is not pertinent to this discussion.

The majority of the virtual binding functionality for this sample has been implemented inside overridden Objective Grid virtual functions.
10.3 The Program

This section examines the relevant VirtGrid source code. The grid control in this sample displays and allows end users to edit data stored in the document's member object m_data.

The following are code listings and discussions of the relevant overrides in this sample.

10.3.1 Initialization and Column Count

```cpp
void CVirtGridView::OnInitialUpdate()
{
    CGXGridView::OnInitialUpdate();                   // 1
    SetColCount(GetDocument()->m_data.GetColCount()); // 2
    SetStyleRange(CGXRange().SetCols(4),              // 3
                  CGXStyle().SetControl(GX_IDS_CTRL_CHECKBOX));
    SetStyleRange(CGXRange().SetCols(1),              // 4
                  CGXStyle().SetInterior(RGB(150,150,150)));
    SetStyleRange(CGXRange().SetCols(2),              // 5
                  CGXStyle().SetTextColor(RGB(0,0,255)));
    GetParam()->EnableMoveCols(FALSE);
    GetParam()->EnableMoveRows(FALSE);
    m_nClipboardFlags = GX_DNDTEXT;
}
```

Override `OnInitialUpdate` to initialize the virtual grid. This override can also be used to initialize other aspects of the grid object.

//1 Calls the base class to allow for default initialization of the grid.

//2 Sets the column count in the grid object based on the column count of the external data source. This allows you to set the base styles for the columns in subsequent lines.

//3 Changes the column base style in the corresponding columns thereby affecting all the cells in those columns. Note that `SetStyleRange()` cannot be called for individual cells because `SetRowCount()` has not been called to force the grid object to allocate space for individual cells.

//4 Disallows column and row movement. For ease of explanation in this tutorial, we will not allow columns or rows to be moved. Before columns or rows can be moved, a mechanism must be put in place to track or sync the data source and the grid object. There are two techniques that will accomplish this.

First, an index map could be created to map grid coordinates to the underlying data structure's coordinates. The map would be used in `GetStyleRowCol()` and `StoreStyleRowCol()` to retrieve and set the appropriate data from the data source. The second technique is to actually move the data in the underlying data structure.

Both techniques have advantages and disadvantages. The nature of your application and data will determine which technique is more appropriate. In either case, two new virtual overrides will come into play, `StoreMoveCols` and `StoreMoveRows`. Please refer to Section 2.3 for more information.

//5 Forces the grid object to copy only the cell's data rather than the full style during clipboard operations. This setting is necessary because the grid is not maintaining individual cell styles.
10.3.2 Row Count

ROWCOL CVirtGridView::GetRowCount()
{
    return GetDocument()->m_data.GetRowCount(); // 1
}

Override GetRowCount() to furnish the number of data rows to the grid object. This differs from an explicit call to SetRowCount() in that no style objects are allocated for each individual cell. In a virtual grid, the data is stored externally to the grid object making individual style objects unnecessary. An explicit call to SetRowCount() would only waste memory and resources.

//1 Return the row count based on the row count of the external data source.

GetRowCount() and GetColCount() are used extensively to validate values within routines. In addition, because HitTest() is called repeatedly on every mouse move and because this method touches a lot of grid code, including GetRowCount(), it is not unusual for GetRowCount() to be called thousands of times in a short period of time. Because GetRowCount() and GetColCount() are called quite often, your overrides of these methods are not an appropriate place to do extensive calculation.

10.3.3 Making the Virtual Connection Between Grid Object and Data Source

BOOL CVirtGridView::GetStyleRowCol(ROWCOL nRow,
    ROWCOL nCol,
    CGXStyle & style,
    GXModifyType mt,
    int nType)
{
    BOOL bRet = CGXGridView::GetStyleRowCol(nRow, // 1
        nCol, style, mt,
        nType);
    if(nType >= 0)                                 // 2
    {
        if(nCol != 0)                                // 3
        {
            style.SetValue(GetDocument()->m_data. // 4
                GetValueRowCol(nRow, nCol));
        }
    }
    return bRet;
}
BOOL CVirtGridView::StoreStyleRowCol(ROWCOL nRow,  //5
    ROWCOL nCol,
    const CGXStyle *pStyle,
    GXModifyType mt,
    int nType)
{
    if(nType == -1) // 6
    {
        return CGXGridView::StoreStyleRowCol(nRow,
            nCol,
            pStyle,
            mt,
            nType);
    }
    else if(nType >= 0) // 7
    {
        if(nCol != 0 && pStyle->GetIncludeValue()) // 8
        {
            GetDocument()->m_data. // 9
                StoreValueRowCol(nRow, nCol,
            pStyle->GetValue(),
            gxOverride);
            return TRUE;
        }
    }
    return FALSE;
}

Override GetStyleRowCol() to supply the data dynamically from the external data source. This virtual function is called every time a cell is redrawn on the DC. Getting the data from the data source from within this override will ensure that the grid is in sync with the data source.

//1 Call the base class to gather any base style information.

//2 Recall from the architecture discussion that GetStyleRowCol() is called four times for each cell that is redrawn. The four calls gather the cell specific style, the column base style, the row base style, and the table base style respectively. The nType parameter specifies which call is being made. An nType >= 0 indicates the cell specific style is being retrieved. An nType = -1 indicates a column, row or table base style is being retrieved. We are only concerned with cell specific styles in GetStyleRowCol(), therefore, this line masks out all calls to GetStyleRowCol() for styles other than cell specific.

//3 This check masks out requests for row header (column 0) styles. These are different from row-wide base styles (indicated by nType = -1 and nCol = 0). We are only interested in supplying column headers and individual cell data.

//4 Get the value from the external data source and set it in the composed style.

//5 Override StoreStyleRowCol() to store the data dynamically to the external data source. This virtual function is called when the end user has changed the contents of a cell and the cell is deactivated.

//6 Recall that nType == -1 means that StoreStyleRowCol() was called to store base style information. We are not storing base style information in the external data source therefore this check masks out base style calls and forwards them to the base class.

//7 This masks out calls to store specific cell styles, which we are storing in the external data source.
This line is performing two checks. The first check, `nCol != 0`, masks out calls to store row header information. The second check, `pStyle()->GetIncludeValue`, verifies that the style passed in actually contains a value. In this example, we are storing only cell values to the external data source.

This retrieves the new cell value from the style parameter and stores it in the external data source.

### 10.3.4 VirtGrid Conclusion

The VirtGrid tutorial demonstrates the minimum requirements to virtually bind a grid object. Keep in mind that virtual binding simply means that a grid’s data is stored externally to the grid object. The stored data is not limited to cell values. It is possible to store any aspect of a cell’s style externally. For the sake of discussion, we demonstrated only cell value storage, but the technique is comparable for all segments of cell styles.
11.1 Troubleshooting

This section discusses build problems while building the grid libraries.

When I try to build the ODBC/ADO libraries it reports that it cannot find some libraries?

You will have to build the corresponding grid core libraries before you attempt to build the odbc/ado libraries.

I get an error saying that “gxall.h” is not found.

This usually happens when the Visual Studio environment wizard is not properly set up to compile the grid. You can run the environment wizard (enwiz32.eze) that ships with the grid to correct this. You will have to restart Developer Studio after running the environment wizard for the changes to take effect.

Why does a grid initialized using SubclassDlgItem not show up in a dialog?

The “Class Name” in your Custom Control’s Properties dialog should be GXWND and not the C++ class name of your subclass. Also, check for GXInit and see if resources are included. Look for any TRACE messages. Dialog creation could have failed due to other reasons (not caused by the grid).

The grid never shows up in release mode. In debug mode, it ASSERTS inside the grid code when it tries to load a resource.

You probably forgot to call GXInit() in your application’s InitInstance(). In addition, do not forget to include the grid resources depending on your build configuration.
12.1 Advanced Design Overview Introduction

The Microsoft Foundation Class (MFC) framework has become the *de facto* standard for Microsoft Windows development worldwide. While MFC contains more than 200 classes it does not cover every aspect of Windows development. For many specific applications programmers must write their own solutions or use third party developed controls. Many of these controls are sold without source code as binary black boxes. MFC extension DLLs rectify this by offering users the same kind of extensibility and functionality that has made MFC a great and ubiquitous product. It is against this backdrop that the design goals for Objective Grid are set. These are listed in brief below:

1. The grid component should be extensible.
2. The grid component should integrate closely with the MFC framework. In other words, do not reinvent the wheel. Make the Objective Grid classes a seamless extension of MFC.
3. The grid component should be resource sensitive. Allow users to pick and choose which functionality should be included in their project.
4. The product should have all the features that a grid component should have to be successful in the grid market.

Some of the goals listed above were achieved in the early releases. Others were achieved much later. This document does not cover the chronology of this process but rather lists the means that were used. Reading this chapter is not essential to using Objective Grid, nor is it a prerequisite to building a great product with Objective Grid. But it does give an insight into the product that cannot be gleaned from the other documentation or even from browsing the source.
### 12.2 Objective Grid Defined

Objective Grid is a class framework that provides a Microsoft Foundation Class (MFC) developer with extension classes that implement a grid control. A grid control is a user interface component that displays data in rows and columns, allowing end users to manipulate that data. The Objective Grid classes fit seamlessly with MFC and in many cases inherit from existing MFC classes such as `CView` or `CWnd`. Objective Grid implements a full set of grid features.

The Objective Grid class framework lets you create full featured spreadsheet applications with only minor programming effort. The required functionality is implemented by the grid framework. You can derive classes from the base classes provided with Objective Grid and extend the functionality as required.

Some of the important features of Objective Grid are:

- Ease of enhancement via C++ mechanisms and flexibility.
- Providing a variety of cell types (e.g. edit field, combo box, check box, list box).
- Object-oriented cell architecture lets developers add new custom cell types.
- Component-based architecture.
- Support for:
  - Database connectivity (ODBC, ADO and any other data sources).
  - Undo/Redo for grid operations.
  - Printing and print preview.
  - Find and Replace.
  - Cut and Paste.
  - Serialization.
  - International character sets through Unicode and DBCS.

---

**Unicode** is not supported in this version of the Excel read/write classes. When building Grid libraries with Unicode, an alternative to the read/write classes is Excel Automation, discussed in Section 26.7, “Excel Automation with Objective Grid.”
12.3 Objective Grid Core Architecture In Brief

A grid consists of cells that display information in the grid window. Each cell is identified through a unique row and column number pair. Cells have attributes associated with them such as text, font and borders. Attributes are stored in a cell data object. For Objective Grid, the cell data object is an instance of the `CGXStyle` class. The cell type itself is also an attribute of the cell. Some examples of cell types are edit box, static text, combo box, list box, radio button, push button and check box. In Objective Grid, each cell type is implemented through a class derived from `CGXControl`. `CGXControl` is an abstract base class that defines the interface between a grid object and a cell type object.

Objective Grid separates cell data from the cell type object. This means the programmer can change the attributes of a cell independent of the cell type. There is no need to get a pointer to the cell type object in order to change the content of a specific cell. This also implies that the cell type object does not store cell specific information. Objective Grid shares one cell type object among many cells. Responsibilities of the cell type object are drawing the cell and interpreting user input. If the user changes the cell text or value, all changes are stored in the cell data object for the cell.

Another interesting aspect in the interaction between cells and the grid object is that the cell attributes are determined at run time. The cell data objects are not persistent in memory. Whenever the grid object draws a specific cell, a cell data object is created for the cell and filled by calling virtual member functions. A programmer can override these virtual functions and specify the cell attributes at run time depending on the current context of the application. Because cell data objects are determined at run time, it is possible to implement a kind of cell attribute inheritance.

Suppose you want to change the font for a whole row of cells. Normally, you would have to loop through all cells in that row and change the font attribute, but with the cell attribute inheritance implemented in Objective Grid, you only have to change the font in the row style. All cells in the affected row will automatically inherit the font settings from the row style. Only cells that have already stored individual font settings in the cell data object will not be affected.

Cell attribute inheritance lets you group specific kinds of cells and make them have similar attributes. For example, all row headers can be changed through the "Row Header Style" and all column headers can be changed through the "Column Header Style". Default cell attributes can be changed through the "Standard Style". You can easily add your own base styles and let certain cells inherit their cell attributes from these base styles.

Objective Grid is composed of several groups of MFC extension classes which work together to provide the above features and design principles:

- **Drawing Classes** - The Objective Grid drawing classes perform all of the drawing and updating of the grid cells. Both `CWnd` and `CView` derived grid classes are available for maximum flexibility.

- **Control Classes** - Objective Grid cells can be a variety of control types. A control MFC extension class is provided for each type of control. An abstract base class is also included that helps the developer create new kinds of controls to supplement the existing controls.

- **Style Classes** - A style refers to the attributes of a cell or group of cells. The Objective Grid style classes manage these attributes and provide a pre-built dialog for modifying them.
**Browser Classes** - The Objective Grid Browser classes let you easily browse any external data sources by overriding some virtual methods. Special derivatives of the Browser Classes let you connect to ODBC and ADO data sources.

**Utility Classes** - Objective Grid uses a variety of MFC extension classes for internal utility classes. These classes may be helpful in other parts of your applications. Examples of the utility classes are a long operation class that lets you monitor and cancel long operations, a tabbed window, and an Undo/Redo architecture.

### 12.4 Objective Grid Control Architecture

Objective Grid consists of several cells that display information in the grid window. Each cell is associated with a cell type object. Some examples of cell types are edit box, static text, combo box, list box, radio button, push button and check box. As we mentioned earlier, in Objective Grid, each cell type is implemented through a cell type class derived from `CGXControl`. `CGXControl` is an abstract base class that defines the interface between a grid object and a cell type object. Responsibilities of the cell type object are drawing the cell and interpreting user input. Additional interfaces in the control class support find/replace and copy/paste.

MFC already provides classes that wrap many Windows controls and we wanted to reuse these classes as cell types in Objective Grid. For example, the `CEdit` class could be used as edit field, the `CComboBox` class as combo box cell type. In general, a programmer can very easily inherit from these window classes. However, with Objective Grid the problem was that these window controls all have a very different interface. Therefore we had to come up with ways in which we could use these controls from the grid in a consistent manner adapting each to our requirement.

The solution to this problem was to adapt the interface of the window class to the `CGXControl` interface. We did this in two ways:

1. By multiply inheriting from the `CGXControl` interface and the window class.
2. By composing a `CWnd` instance within a wrapper control. Since `CWnd` is a consistent interface that is implemented (or rather available) on all windows controls (For those ActiveX people, 'Windowed', Windows controls).

**Figure 115** shows the resulting class hierarchy for controls in Objective Grid. The implementation of complex controls like `CListBox`, `CEdit`, `CComboBox` and `CRichEditCtrl` could be reused by multiply inheriting from `CGXControl` and the window class. You can also observe that primitive controls like push button or check box are implemented directly in Objective Grid.
Figure 115 – Control classes hierarchy

- CGXControl
  - CGXCheckBox
    - CGXCheckBoxEx
  - CGXPushButton
  - CGXRadioButton
  - CGXPushButtonEx
  - CGXListBox
  - CGXStatic
    - CGXEditControl
      - CGXComboBox
        - CGXTabbedComboBox
          - CGXCheckListComboBox
      - CGXHotSpotEdit
      - CGXSpinEdit
      - CGXScrollEdit
      - CGXPasswordControl
    - CGXMaskControl
      - CGXHeader
        - CGXComboBoxWnd
      - CGXTabbedComboBoxWnd
    - CGXProgressCtrl
    - CGXRichEditCtrl
      - CGXBitmapButton
      - CGXWndWrapper
      - CGXCurrencyEdit
      - CGXDateTimeCtrl
      - CGXBDropEdit
        - CGXBDateTimeCtrl
12.4.1 The pure CGXControl approach

The grid control classes, CGXEditControl, CGXListBox, CGXComboBoxWnd and CGXRichEditCtrl (among several others) use multiple inheritance to adapt the CWnd interface to the CGXControl interface. Let us take CGXEditControl as an example. The implementation of the other controls is very similar.

Figure 116 – CGXEditControl implementation

CGXEditControl is not only an adapter that adapts the functionality of the CEdit class to the grid, it also provides a lot of functionality not provided by the CEdit control class. For example, CGXEditControl is responsible for hiding, showing and setting the focus to the cell, changing the text in the window control and the implementation of Find/Replace, Cut/Paste and text formatting.

The nice part is that the editing functionality could be reused from the CEdit class. Once the focus is set to the edit window, the windows edit control lets the user scroll and edit the text, move the caret, copy/paste text.

There are two types of interfaces in the CGXControl class. Some functions are called from the grid window (e.g. Init(), Draw()) and their implementations will translate the call to the edit window, others will be called from within the control and translate the call to the grid object (e.g. Store(), OnValidate()). Therefore the CGXControl object needs a pointer to the grid object.

Let us look at some important interfaces that implement the core of this control.
12.4.1.1 Functions that are called from the grid and translate into calls to the edit window

virtual void
Init(ROWCOL nRow, ROWCOL nCol);

This function is called from the grid when the user has moved the current cell to a new position. This function will initialize the contents of the edit window with the data of the current cell.

virtual void
Draw(CDC* pDC, CRect rect, ROWCOL nRow, ROWCOL nCol,
     const CGXStyle& style, const CGXStyle* pStandardStyle);

This is the function that adapts the behavior of the edit window to the grid. The Draw() function is only called from the grid. It will position the edit window in the grid window, show the window and set the focus to it. After the focus has been set to the edit window, the edit window will continue interpreting user actions like keystrokes and mouse messages.

Draw() operates in two different ways. It will be called for every cell in the grid, but only one cell in a grid can have the focus. Therefore, Draw() checks if the row (nRow) and column (nCol) coordinates passed to the function are the same coordinates as the current cell’s coordinates. Only if the coordinates match with the current cell’s coordinates, Draw() will show the edit window and set the focus to it. All other cells will only be drawn static. That means only the text is drawn to the screen and no editing functionality is needed.

virtual BOOL
GetValue(CString& strResult);

GetValue() is called from the grid to determine the current value in the edit control. An interesting aspect of this function is that it combines the very different behavior of many controls. For example, the value of the edit window is the text displayed in the cell, but for a list box or combo box it is the index of the selected item and not the displayed text. Therefore, CGXListBox and CGXComboBoxWnd have special implementations of this function that adapts their behavior to the grid.

virtual void
SetValue(LPCTSTR pszRawValue);

SetValue() is called from the grid and is the converse to GetValue(). It updates the displayed text in the edit window.

virtual BOOL
LBButtonDown(UINT nFlags, CPoint pt, UINT nHitState);
virtual BOOL
KeyPressed(UINT nMessage, UINT nChar,
          UINT nRepCnt = 1, UINT flags = 0);

Mouse and keyboard messages are delegated from the grid to the CGXControl class (for that cell) when the edit window does not already have the focus (for example this is how tab keys are handled when no cell has focus). Once the focus has been set to the edit window, these messages don’t need to be delegated any more because the messages will be sent directly to the edit window from the MS Windows.
12.4.1.2 Functions called from within the control which translate the call to the grid object

virtual BOOL Store();

This function is called from within the control when the user has changed the cell contents and is about to leave the current cell. The function writes changes back into the cell data object in the grid.

virtual BOOL OnValidate();

This function is called from within the control when the user is about to leave the current cell. It sends a notification to the grid window to give the grid a chance to validate the cell’s contents. If the value is invalid, a message box will appear and the current cell will remain at its old position.

virtual CRect GetCellRect(ROWCOL nRow, ROWCOL nCol);

This function is called from within the control and determines the cell rectangle for the given coordinates which needs to be computed in the grid object because only the grid object knows the row heights and column widths of other rows and columns.

Other interfaces in the CGXControl class handle special events in the grid and provide feedback about changes in the control (e.g. OnModifyCell will be called when the user modifies text).

12.4.2 The Window Wrapper Approach

This approach is used by the CGXWndWrapper class. This class can be used to host any CWnd-based object in the grid. CGXWndWrapper relies on object composition.
**CGXWndWrapper** is a kind of “one-way” adapter in opposite to the implementation of class adapters like **CGXEditControl**. Only interfaces that are called from the grid window (e.g. Init(), Draw()) could be implemented because the **CWnd** object (the composed object) does not know anything about the grid object. This has the advantage that any kind of windows control can be used through the **CGXWndWrapper** class but has the disadvantage that several limitations arise because the **CWnd** cannot send feedback about its state or changes to the grid (except through the limited **CWnd** interface).

Another disadvantage is that there is no way of transferring data between the grid and the contained object because the **CWnd** class does not provide a consistent interface (the manipulation interfaces are specific to the control) which lets the grid exchange data with the control. But this problem could be easily solved by providing special **CGXWndWrapper** derivatives that know how to transfer data with special kinds of Windows controls (e.g. a **CGXListBoxWrapper**). We do this in the grid to adapt ActiveX controls to the grid.

As there is no way to get feedback from the contained object, the only important interface for the **CGXWndWrapper** class is:

```cpp
virtual void Draw(CDC* pDC, CRect rect, ROWCOL nRow, ROWCOL nCol, const CGXStyle& style, const CGXStyle* pStandardStyle);
```

This is the function that primarily adapts the general behavior of any **CWnd** to the grid. The Draw() function will position the contained object window in the grid, show the window and set the focus to it. After the focus has been set to the window, the window will continue interpreting user actions like keystrokes and mouse messages.

We have illustrated both approaches that were used in creating grid controls. The pure **CGXControl** (both multiple inheritance and plain derivation from **CGXControl**) is preferred over the object wrapper (**CGXWndWrapper** approach). Some of the issues that relate to the use of these discrete approaches are listed below.
1. The pure *CGXControl* approach allows us to have special adaptation for the particular controls. This essentially enables data transfer to and from the control. This results in the possibility that one control object can be reused between several cells. In fact this is used in the grid and is the basis for Objective Grid’s control sharing architecture that is explained later in this chapter.

2. While it is often easy to port a control into the grid with the window wrapper approach it is often more difficult to maintain the control (besides this being resource intensive as mentioned above)

There are some situations when the window wrapper method may be the only usable (or more usable) method. In general this is the case with controls that implement special functionality that they retain as part of their internal state. For example take the case of ActiveX control containers that are used as controls. To have a system to serialize and de-serialize the data that is associated with these and to retain this in the grid would be more work that to use window wrapper and single instance per cell approach. This is also the more easily workable approach with ActiveX controls.
12.5 Control Sharing In the Grid

Using lots of objects can be expensive. Therefore, sharing objects of the same class can bring a lot of resource savings. The previous section alluded to ‘Control sharing’ in the grid. In this section, we will look at this very important aspect of the grid architecture in more detail. Each cell in the grid is identified through a unique row and column number pair. The user should be able to specify the cell type and also the appearance of the cell with attributes like the text, the font, borders and some others. The cell type should be replaceable at run time and provide a common interface for both the programmer and the end-user of the grid component. The programmer should be able to exchange data and specify the cell appearance independent of the cell type. The end-user should be able to operate with different cell types in a unique way. For example, different cell types should be able to be copied and pasted in the same way. Also, doing Find/Replace should be possible with a unique interface.

As explained earlier, the object-oriented answer to this kind of problem is to create an abstract base class and inherit the individual cell types from this base class. Each cell should be an object and can be accessed through a common interface.

Some of the problems with this approach are:

1. Windows will go out of resources because the number of windows is limited.
2. Memory costs can be very high.
3. Performance will decrease because each window has to be repositioned when the user scrolls the grid.
4. Creating many window controls will cost a lot of time because creating a window is a very expensive operation.

Another problem is that if the cell appearance is stored in the cell type object, the appearance settings would get lost when the cell type is replaced.

We implemented a solution that enabled us to reduce the use of resources dramatically. The essence of this solution can be explained as below:

1. Make the cell attributes an extrinsic state which can be passed to the cell type object.
   
   Cells can have certain attributes like the text, the font, borders and some other attributes which let you specify the appearance of the cell. Objective Grid separates data to be displayed in a cell and the cell attributes from the cell type object. All attributes are stored in the cell data object, which is an instance of the CGXStyle class. The cell data objects are stored in the grid and can be accessed through grid routines.

2. Provide a base class for all cell types (in this case CGXControl).

   Each cell type is implemented through a class derived from CGXControl. CGXControl is an abstract base class, which defines the interface between a grid and a cell type object. Responsibilities of the cell type object are drawing the cell and interpreting user input (keyboard and mouse messages). The cell type object does not store cell specific information. If the user changes the cell text or value, all changes are stored in the cell data object for the cell.
Member functions in \textit{CGXControl} can receive and act on extrinsic state and don’t rely on internal stored state. For example, when drawing a cell, the grid passes the rectangle, the row and column coordinates and the cell data object to Draw method of the cell type object. The cell type object will draw the cell based on these parameters.

3. Maintain a pool of cell type objects.

The grid maintains a list of \textit{CGXControl} objects. \textit{CGXControl} objects are created and registered at initialization time of the grid. Each control is associated with an unique integer id and can be looked up in the pool through this integer id. This integer ID is an attribute of the \textit{CGXStyle} class. Therefore cell types can be easily and simply replaced at run time by changing the integer id in the cell data object. Controls own a pointer to the grid object and therefore cannot be shared among different grid objects.

We should also explain the concept of a current cell. This is a real exception of the rule that the cell type object should not store information about the cell. When a cell is the current cell and the user modifies its contents, all changes are stored in the cell type object directly. Only when the user accepts the changes (e.g. by moving the current cell to a new position) will the changes be stored back into the cell data object in the grid.

\textbf{Figure 118} illustrates the main interactions between a grid, a cell and a control:

\begin{verbatim}
LookupStyleRowCol() returns the cell data object for the cell with the cell type identifier. The grid can look up a pointer to the cell type object in the \textit{CGXControl} object pool (by calling GetRegisteredControl()) and execute operations on this control, such as Draw(), Store(), or Init().

\textbf{Figure 118 – Interactions between a grid, a cell and a control}
\end{verbatim}
The cell type objects are instantiated in `OnInitialUpdate()`. This method is called from the MFC framework before the grid window is shown (in the case of dialog-based grids this is explicitly called). `RegisterControl()` simply stores a pointer to the cell type object in the control pool.

When the grid needs to be repainted, the MFC framework calls the `OnDraw()` member function of the grid. This function loops through all cells and calls the `DrawItem()` member function. `DrawItem()` looks up a pointer to the control object and calls the `Draw()` member function of the associated control object.

`SetCurrentCell()` is a member function in the grid which moves the current cell to a new position. It first stores and deactivates the old current cell and then initializes the new current cell.

The instance diagram in Figure 119 shows how controls can be shared among several cells. `aCell` objects are style-objects constructed in `LookupStyleRowCol()`. `anEditControl` and `aBitmapButton` are control-objects registered in the grid. They are referenced through the cell type ID in the style-object. The grid maintains the control pool.

**Figure 119 – Sharing cell type objects among cells**

As we mentioned earlier, a major difference comes with the concept of the current cell because when the user modifies its contents, all changes are stored in the cell type object directly. Two different types of interfaces were defined for this behavior:

1. Functions which accept external state as attributes.
2. Functions which operate and initialize the state of the current cell. The Draw function is an exception and falls in both of these groups.
Let us first explain why the Draw() function falls in both groups.

```cpp
virtual void
Draw(CDC* pDC, CRect rect, ROWCOL nRow, ROWCOL nCol,
    const CGXStyle& style, const CGXStyle* pStandardStyle);
```

When drawing a cell, the grid passes the rectangle, the row and column coordinates and the cell data object to the Draw() method of the cell type object. Draw() operates in two different ways. Draw() checks if the row (nRow) and column (nCol) coordinates passed to the function are the same coordinates as the current cell’s coordinates. If the coordinates match with the current cell’s coordinates, Draw() will show the edit window and set the focus to it. All other cells will only be drawn static based on the extrinsic state passed to the function.

### 12.5.1 External State Functions

One characteristic for functions that operate with extrinsic state is that the row and column coordinates are passed as attributes. Examples for this type of interface are:

```cpp
virtual BOOL
GetControlText(CString& strResult, ROWCOL nRow, ROWCOL nCol,
                LPCTSTR pszRawValue, const CGXStyle& style);
```

GetControlText() determines the text that will be displayed in the cell. The display text can be different from the cell value when the displayed text should be formatted with 2 digits, for example, or as international date based on the Windows System settings.

GetControlText() will be called when cells should be copied to the clipboard or exported to a text file.

```cpp
virtual BOOL
SetControlText(ROWCOL nRow, ROWCOL nCol, const CString& str, ...);
```

SetControlText() is the counterpart to GetControlText(). It converts the display text to the cell value and stores it in the cell data object in the grid.

SetControlText() will be called when cells should be pasted from the clipboard or imported from a text file.

```cpp
virtual BOOL
FindText(const GX_FR_STATE& find, ROWCOL nRow, ROWCOL nCol, ...);
```

This function is called to find text in the cell data object at the given row.

### 12.5.2 Current Cell State Functions

The main characteristic for functions that operate on the current cell state is that no row and column coordinates are passed as attributes. Examples for this type of interface are:

```cpp
virtual BOOL Store();
virtual BOOL OnValidate();
virtual BOOL LButtonDown(UINT nFlags, CPoint pt, UINT nHitState);
virtual BOOL KeyPressed(UINT nMessage, UINT nChar, UINT nRepCnt = 1, UINT flags = 0);
```
The following function accepts row and column coordinates because it stores the row and column coordinates into the cell type object.

```c
virtual void Init(ROWCOL nRow, ROWCOL nCol);
```

While this control-sharing architecture may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state. However, such costs are offset by space savings, which increase as more objects are shared. With Objective Grid, increased run-time costs can be the initialization of the control every time a cell needs to be drawn. You can’t simply apply a large choice list with many hundreds items to one combo box. All the items will be transferred to the control and the control will be initialized with this list every time it is drawn. But, especially in conjunction with the cell attribute inheritance (See Section 12.8, “Objective Grid Style Architecture.”) discussed later, space savings can be enormous if that one large choice list can be applied to a base style which is used among many cells.
12.6 Objective Grid Integration With Document/View Architecture

For those readers who are familiar with design patterns, the MFC document/view concept is very similar to the Observer pattern: A view is associated with a document, and the document maintains a list of pointers to all associated views. When a view changes data in the document, it can trigger a notification to all associated views that the document data has changed. In Objective Grid, we extended the Observer pattern in such a way that we systematically separated methods, which update the display from methods, which changes data.

The document-view architecture in MFC allows the programmer to separate data from the views. An MFC application can offer the user many ways to view the document (the data), and some views may present only a part of the data or results computed from the raw data. For instance, a statistical application could present the data as a grid of numbers, a chart, or a summary showing computed values such as the mean and standard deviation.

A common situation in complex applications is the need to propagate information to all views. For example, suppose four view windows show different views of the same document and the user changes data in one view. Now it's necessary to notify all views about the change. The views should be able to update themselves most efficiently. Note also that views can draw to devices other than the video display — e.g., on plotters or printers.

In MFC, a document object keeps a list of its views, provides member functions for adding and removing views, and supplies the UpdateAllViews() member function for letting multiple views know when the document's data has changed. Each view has an OnUpdate() member function which is called from UpdateAllViews() method for each view.

Let us explain how drawing works in a view.

With the exception of mouse drawing, nearly all drawing in your application occurs in the view's OnDraw() member function, which you must override in your view class. Your OnDraw() override:

1. Gets data by calling the document member functions you provide.
2. Displays the data by calling member functions of a device-context object that the framework passes to OnDraw().

When a document's data changes in some way, the view must be redrawn to reflect the changes. Typically, this happens when the user makes a change through a view on the document. In this case, the view calls the document's UpdateAllViews() member function to notify all views on the same document to update themselves. UpdateAllViews() calls each view's OnUpdate() member function. The default implementation of OnUpdate() invalidates the view's entire client area. You can override it to invalidate only those regions of the client area that map to the modified portions of the document.

The UpdateAllViews() member function of class CDocument and the OnUpdate() member function of class CView let you pass information describing what parts of the document were modified. This hint mechanism lets you limit the area that the view must redraw. OnUpdate() takes two hint arguments. The first, lHint, of type LPARAM, lets you pass any data you like, while the second, pHint, of type CObject*, lets you pass a pointer to any object derived from CObject.
When a view becomes invalid, Windows sends it a WM_PAINT message. The view's OnPaint() handler function responds to the message by creating a device-context object of class CPaintDC and calls your view's OnDraw() member function. You do not normally have to write an overriding OnPaint() handler function.

Your code for drawing in the view first retrieves a pointer to the document and then makes drawing calls through the device context. The following simple OnDraw() example illustrates the process:

```cpp
void CMyView::OnDraw( CDC* pDC )
{
    CMyDoc* pDoc = GetDocument();
    CString s = pDoc->GetData();   // Returns a CString
    CRect rect;

    GetClientRect( &rect );
    pDC->SetTextAlign( TA_BASELINE | TA_CENTER );
    pDC->TextOut( rect.right / 2, rect.bottom / 2,
                 s, s.GetLength() );
}
```

In this example, you would define the GetData() function as a member of your derived document class.

The example prints whatever string it gets from the document, centered in the view. If the OnDraw() call is for screen drawing, the CDC object passed in pDC is a CPaintDC whose constructor has already called BeginPaint(). Calls to drawing functions are made through the device-context pointer.

This structure is illustrated in Figure 120, “Document - View Relationship.”

**Figure 120 – Document - View Relationship**
The CDocument class provides operations for adding and removing views from the document and functions to iterate through the views. UpdateAllViews() will loop through all views stored in m_viewList and call the Update() method of the CView class.

Here are some code snippets for the class declaration of CDocument and CView:

class CDocument : public CCmdTarget
{
    // Operations
    void AddView(CView* pView);
    void RemoveView(CView* pView);
    virtual POSITION GetFirstViewPosition() const;
    virtual CView* GetNextView(POSITION& rPosition) const;

    // Update Views (simple update - DAG only)
    void UpdateAllViews(CView* pSender, LPARAM lHint = 0L,
                         CObject* pHint = NULL);

    // Attributes
    CPtrList m_viewList;                // list of views

    ...;
};

class CView : public CWnd
{
    ...
    // General drawing/updating
    virtual void OnUpdate(CView* pSender, LPARAM lHint,
                          CObject* pHint);
    virtual void OnDraw(CDC* pDC) = 0;
};

Some points worth mentioning:

- **Who triggers the update?** The document and its associated views rely on the notification mechanism to stay consistent. The MFC approach makes views responsible for calling UpdateAllViews() at the right time.

- **Dangling references to deleted subjects.** Deleting a document should not produce dangling references in its views. The MFC approach ensures that documents are instantiated and deleted through the MFC class framework. The MFC framework will close and destroy all views before a document is destroyed.

- **The push and pull model.** The MFC approach uses the push model: The document sends detailed information about the change (a hint) to all views, whether they want it or not. This approach works fine when a view triggers the update and other views understand the hint. If the document itself has to trigger an update, this method has the disadvantage that the document has to know implementation details of the attached views.
12.6.1 Push Model Details

In Objective Grid, we extended the push model concept. In the grid view class, we systematically separated user interactions and methods that update the display from methods which change data. Each method that updates the display is associated with an ID. This ID can be used as a hint to be sent among views.

For example, if the user does a specific interaction, such as typing text in a cell and moving the current cell, the view will call the `SetStyleRange()` operation in the grid to store the value of the current cell. All associated views have to update this cell. The grid-component uses the following scheme to keep all views up to date. As mentioned, a user interaction results in calling a command such as `SetStyleRange()`. For each command, the grid-component contains two further methods. One method (which gets called once) stores and actualizes the data. The other method (which gets called for each view) updates the display. Here we explain the scheme by example:

The command method `SetStyleRange()` gets called after the user has changed the cell:

```cpp
BOOL CGXEditControl::Store()
{
    // Calls SetStyleRange() and resets the modify flag
    CString sValue;
    if (GetModify() && GetValue(sValue))
    {
        return Grid()->SetStyleRange(
            CGXRange(m_nRow, m_nCol),
            sValue);
    }
    return TRUE;
}
```

`SetStyleRange()` calls the store method `StoreStyleRowCol()` for each cell in the range. `StoreStyleRowCol()` actualizes the data.

```cpp
BOOL CGXGridCore::SetStyleRange(
    const CGXRange& range,
    const CGXStyle* pStyle)
{
    for each nRow, nCol in range
        // store style information for the specific cell
        StoreStyleRowCol(nRow, nCol, style);

    // update the view (bCreateHint = TRUE)
    UpdateStyleRange(range, pStyle, TRUE);
}
```

`SetStyleRange()` calls the update method `UpdateStyleRange()`, telling the method to create a hint and send it to the document. `UpdateStyleRange()` updates the window and creates a hint. Each update method in the grid is associated with an integer id. This integer is used as hint and will be sent together with some additional information to all views. The hint is sent to the document by calling `CDocument::UpdateAllViews()`.
void CGXGridCore::UpdateStyleRange(const CGXRange& range, const CGXStyle* pStyle, BOOL bCreateHint)
{
    // redraw all cells in range
    Redraw(range);

    // Create Hint
    if (bCreateHint)
    {
        CGXGridHint hint(gxHintUpdateStyleRange, m_nViewID);
        hint.range = rgBoundary;
        hint.pStyle = pStyle;

        UpdateAllViews(m_pGridWnd, 0, &hint);
    }
}

CDocument::UpdateAllViews() sends the hint to all associated views by calling CView::OnUpdate(). Each view analyzes the hint ID and calls the corresponding update-method telling the method not to create a hint. The update method decides depending on the specific context of the view how to update the view. The integer-id will be used to call the appropriate update method in the view. Note that the bCreateHint parameter will be passed as FALSE to avoid an infinite recursion.

void CGXGridCore::OnUpdate(CView* /*pSender*/, LPARAM /*lHint*/, CObject* pHint)
{
    CGXGridHint &info = *((CGXGridHint*) pHint);

    switch (info.m_id)
    {
    case gxHintUpdateStyleRange:
        UpdateStyleRange(
            info.range,
            info.pStyle,
            FALSE
        );
        break;

    ...;
    }
}
Figure 12.1 – Update mechanism in the grid component

- **View**
  - User has changed the cell
  - Command method \( M_1 \)
  - Store data into the document \( M_2 \)
  - Update the view \( M_3 \)
  - Create hint and send it to the document, if requested.

- **Document**
  - \( \text{CDocument::UpdateAllViews()} \)

- **Other views**
  - \( \text{CView::OnUpdate()} \)

**Legend:**
- \( M_1 \): Command method
- \( M_2 \): Store method
- \( M_3 \): Update method
12.7Undo/Redo Architecture

It is essential for user-friendly applications to support Undo and Redo. Therefore, we wanted to implement this feature in Objective Grid. One advanced requirement for Undo and Redo is the need for a transaction like behavior in the grid. This means, several commands should be packed together and undone or redone in one step.

The solution to this problem was to create command objects for each operation. Each operation executed in the grid creates a command object and stores information how to undo the operation in the command object. The command object will be stored in a history list. Every time a command needs to be undone, the command object will be popped from the history list and executed.

Objective Grid provides many operations that let you change the data in the grid. Some examples for operations in the grid are: changing the cell contents, insert, remove or move rows or columns, freeze rows or columns, cut and paste cells and much more. As we mentioned earlier, we took care to separate user interactions from commands, which perform changes in the grid. The code for performing the command is not part of the user interaction code itself. That means, when the user performs an action in the grid, this action will result in a call to a command method. A consequence of this approach is that grid operations can easily be executed programmatically and through user interactions. For example, when the user changes the row height of a row, the grid will execute the command `SetRowHeight()`. `SetRowHeight()` can also be called directly from within the grid without user interaction.

The `CGXCommand` class is an abstract base class for command objects. For every type of operation in the grid, a special `CGXCommand`-derivative has been implemented which stores all information necessary for undoing the operation into the command.

In Objective Grid, commands are methods in the grid class which instantiate a `CGXCommand` object only for Undo support. When the user clicks on a menu item, a command method is executed which creates the command object with Undo information.

All the functionality for the grid operations is implemented in the command methods in the grid. Command objects only store information how to call a command method in the grid.

The following sample code illustrates how `CGXCommand` objects are implemented. The Undo information is passed to the `CGXSetFrozenRowsCmd` object when the constructor is called. The `Execute()` method will be called when the command needs to be undone.

```cpp
class CGXSetFrozenRowsCmd: public CGXCommand
{
public:
    // Construction
    CGXSetFrozenRowsCmd(ROWCOL nFrozenRows, ROWCOL nHeaderRows);

    // Operation
    virtual BOOL Execute(CGXGridCore* pGrid, GXCmdType ctType);

    // Data (Undo information)
    ROWCOL    m_nFrozenRows;
    ROWCOL    m_nHeaderRows;
};
`
CGXSetFrozenRowsCmd::CGXSetFrozenRowsCmd(ROWCOL nFrozenRows, ROWCOL nHeaderRows)
{
    m_nFrozenRows = nFrozenRows;
    m_nHeaderRows = nHeaderRows;
}

BOOL CGXSetFrozenRowsCmd::Execute(CGXGridCore* pGrid, GXCmdType ctCmd)
{
    return pGrid->SetFrozenRows(m_nFrozenRows, m_nHeaderRows,
    GX_UPDATENOW, ctCmd);
}

In the associated command method, the CGXCommand-derived object will be instantiated and initialized with the previous state in the grid. The object will then be added to the Undo/Redo list by calling AddCommand().

BOOL CGXGridCore::SetFrozenRows(ROWCOL nFrozenRows,
    ROWCOL nHeaderRows, UINT flags, GXCmdType ctCmd)
{
    ROWCOL nOldFrozenRows = GetFrozenRows();
    ROWCOL nOldHeaderRows = GetHeaderRows();
    if (StoreFrozenRows(nFrozenRows, nHeaderRows))
    {
        UpdateFrozenRows(nOldFrozenRows, nOldHeaderRows, flags, TRUE);

        if (ctCmd != gxRollback && m_pParam->m_bUndoEnabled)
        {
            AddCommand(new CGXSetFrozenRowsCmd(nOldFrozenRows, nOldHeaderRows),
            ctCmd);
        }

        return TRUE;
    }

    return FALSE;
}

The ctCmd parameter tells the command method if the operation is executed the first time, if it is undone or redone or if should be rolled back (that is, undone without creating Redo information).

AddCommand() interpretes the ctCmd parameter. If the command is executed the first time (= gxDo) or if it is redone (= gxRedo), the command object will be placed in the Undo list. If the command is undone (= gxUndo), the command object will be placed in the Redo list.

void CGXGridCore::AddCommand(CGXCommand* pCmd, GXCmdType ctCmd)
{
    CObList* pList;

    if (ctCmd == gxDo || ctCmd == gxRedo)
        pList = &GetParam()->GetUndoList();
    else
        pList = &GetParam()->GetRedoList();

    pList->AddHead(pCmd);
}
The Undo() operation pops a command object from the Undo list and calls its execute method. The implementation is straightforward:

```
BOOL CGXGridCore::Undo()
{
    CObList& undoList = GetParam()->GetUndoList();

    if (!undoList.IsEmpty())
    {
        CGXCommand* pCmd = (CGXCommand*) undoList.RemoveHead();
        pCmd->Execute(this, gxUndo);
        delete pCmd;
        return TRUE;
    }
    return FALSE;
}
```

The Redo operation is implemented in the same way.

The advanced requirement for Undo/Redo was the need for a transaction-like behavior in the grid. This means, several commands should be packed together and undone/redone with one call. The command pattern suggests creating a MacroCommand which can hold several command objects. This approach could also be used in Objective Grid for implementing the CGXBlockCmd command. The CGXBlockCmd class owns a list of command objects. When the Execute command of the CGXBlockCmd is called, it loops through all commands in the lists and calls each commands Execute() method.

The Undo operation is designed to work for the grid, but not for individual edit cells within the grid. In general, changes in the current cell will simply be discarded and no Undo information will be generated. For instance, if you type in several different cells and then fire Undo several times,
the focus will move in reverse order from one cell to another inside the grid, performing the Undo operation for a sequence of cells. **If you want to be able to undo changes in the current active cell,** you must add the following line to your code:

```cpp
pGrid->TransferCurrentCell();
```

The following citation from source code demonstrates this technique:

```cpp
BOOL CGXCommandFactory::Undo(CGXGridCore* pGrid)
{
    ROWCOL nRow, nCol;

    // Uncomment the following line if you want to be able
    // to undo changes in the current active cell. Otherwise
    // changes in current cell will simply be discarded and
    // no Undo information generated.
    //
    // pGrid->TransferCurrentCell();
    ...
}
```

### 12.7.1 Other Issues Related to the Undo/Redo Architecture

- **How intelligent should a command be?** In the Objective Grid approach, a command object merely defines a binding between the grid and a grid action to be carried out. It provides no additional functionality other than simply calling a grid method.

- **Supporting Undo and Redo.** The pattern suggest several options for storing state information in the command objects. In Objective Grid, state information are the arguments to the operation performed on the grid to return the grid to its previous state.

- **De-coupling.** Command de-couples the Undo and Redo methods, which invoke the operation from the methods which perform the operation. That is, Undo and Redo don’t have to know what action will be performed when they call the Execute method of a command object.

- **Transaction.** Commands can be assembled into a composite command. This allows transaction-like behavior in the grid.

Also, it should be noted that it is not possible to add new commands without changing the grid class. The user will have to subclass the grid and add a command method to the grid and also subclass the `CGXCommand` class and create a command class. You can do this if you have a discrete action that causes a state change that you would like to have Undo/Redo capability.
12.8 Objective Grid Style Architecture

Some applications require that the end-user apply specific settings to certain data objects in a document, while other settings of these data objects should retain their default settings. The default settings may be stored in independent objects allowing the end-user or programmer to create a “gallery” of default types and change these default settings at run time. Whenever default settings are changed at run time, all data objects that did not override the default settings should reflect the new settings.

For example, in a typical word processing system the end user should be able to format individual paragraphs, words or characters in a document with certain formatting like font type, weight, size, alignment and text color. Certain categories of information should always look the same way. For example, this document follows some documentation conventions where the name of the paragraph, the title and the individual category titles all have their own characteristic formatting, which are used throughout the document. Category titles are outlined by using a bold font. In a word processing system, the settings for category titles, paragraph title and standard text are all stored in a document template. Whenever formatting for one of these styles is changed, all text in the document will reflect the new changes. For example, if the user changes the font size for the standard text style, all text will be drawn with a bigger font.

The styles architecture in the Objective Grid library is very similar to this. A grid consists of several cells that display information in the grid window. Each cell is identified through a unique row and column number pair. Cells can have certain attributes like the text, the font, borders and some other attributes which let you specify the appearance of the cell. All these attributes are stored in the cell data object, which is an instance of the `CGXStyle` class. Only changes need to be stored in the cell data object. Unchanged attributes will be determined at run time and loaded from base styles.

When the grid draws specific cells, all cell attributes must be determined for these cells (and not only the changed attributes for the cells). Therefore, whenever the grid needs to draw cells, cell data objects are created for each cell to be drawn and filled by calling virtual member functions. A programmer can override these virtual functions and specify the cell attributes at run time depending on the current context of the application or simply let the grid load the default settings from base styles. We call this the ‘Style Gallery’. This is the plumbing for the implementation of cell attribute inheritance in Objective Grid.

Cell attribute inheritance lets you group specific kinds of cells and make them have similar attributes. For example, all row headers can be changed through the “Row Header Style” and all column headers can be changed through the “Column Header Style”. Default cell attributes can be changed through the “Standard Style”. You can easily add your own base styles and let certain cells inherit their cell attributes from these base styles.

Typical style objects in the Objective Grid style gallery are:

<table>
<thead>
<tr>
<th>Style name</th>
<th>Initialized attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>Font size, alignment, white background, …</td>
</tr>
<tr>
<td>Row Header</td>
<td>Left aligned, bold, gray background</td>
</tr>
<tr>
<td>Column Header</td>
<td>Centered, bold, gray background</td>
</tr>
</tbody>
</table>
The following figure shows the main relations between the grid and the cell data objects. When the grid requests cell information, it calls `LookupStyleRowCol()`, which creates a style-object and initializes the cell's attributes. The base style settings will be loaded by calling the `LoadBaseStyle()` member function. `GetParam()->GetStylesMap()` is a pointer to a `CGXStylesMap` object which holds all the base style objects in Objective Grid and represents the style gallery.

As you can see, `LookupStyleRowCol()` not only lets you load settings from base styles in the `CGXStylesMap`, but also from row styles and column styles. Suppose you want to change the font for a whole row of cells. Normally you would have to loop through all cells in that row and change the font attribute, but with the above approach you only have to change the font in the row style. All cells in the affected row will automatically inherit the font settings from the row style. Only cells that have already stored individual font settings in the cell data object will not be affected.
For each attribute, the style class stores information whether an attribute is changed or if it is unchanged (and therefore retains its default value). This information is used to determine which attributes should be copied from a base style. Each attribute which is a member variable of the \texttt{CGXStyle} class has a corresponding include-bit in the \texttt{CGXStyle} class and each attribute can be accessed through four methods. An example is the text color:

- \texttt{GetIncludeTextColor()} - returns the include bit.
- \texttt{SetIncludeTextColor(BOOL bInclude)} - changes the include bit.
- \texttt{GetTextColor()} - returns the text color.
- \texttt{SetTextColor(COLORREF rgbColor)} - changes the color. This method automatically calls \texttt{SetIncludeTextColor(TRUE)}.

One style object can be applied to another style object. Following operations are supported:

- **Applying only new attributes:** Only those attributes are copied from the source style to the destination style that are included in the source style and not yet included in the destination style.

- **Overriding attributes:** Those attributes are copied from the source style to the destination style which are included in the source style no matter if they are included in the destination style.

- **Copying the style:** All attributes are copied from the source style to the destination style. Attributes that are not included in the source style will be removed from the destination style.

- **Removing attributes:** Attributes that are included in the source style will be removed from the destination style.

Here is how a programmer can apply formatting to some columns and specify that the columns should load their default values from a specific base style (`BS_NUMERIC`):

```cpp
SetStyleRange(CGXRange().SetCols(2,4),
              CGXStyle()
              .SetFont(CGXFont()
                           .SetSize(10)
                           .SetBold(TRUE)
                      )
              .SetBaseStyle(BS_NUMERIC)
              , gxOverride
);
```

The `BS_NUMERIC` style could be specified with

```cpp
BaseStyle(BS_NUMERIC)
    .SetHorizontalAlignment(DT_RIGHT) // right aligned
    .SetFormat("###.##"); // numeric format
```
Style Gallery introduces some run-time costs associated with loading default settings from base styles on demand. However, such costs can be taken into account compared with the run-time costs when you would have to loop through all style objects in your document when a base style object from the gallery is changed.

On the other hand, Style Gallery can save you lots of memory by sharing base style objects among cells. If you have many style objects referring to the same information (e.g. a large string), you can make this style objects refer to one base style object where you store the large text.

Let us now take a look at some of the implementation issues with this architecture:

We encapsulate all state information for document data in one style class. The document itself is implemented as collection of these style objects.

For each attribute in the style class, we add an include-bit so that you can later determine whether a specific attribute is changed or not.

We implement a method (`ChangeStyle()`) in the style class which lets you merge style objects together. This method has to check for each attribute the include bit of the other style and copy the attribute when the same attribute in the current style object is uninitialized.

```cpp
void CGXStyle::ChangeStyle(const CGXStyle& pOther)
{
    // HorizontalAlignment
    if (!GetIncludeHorizontalAlignment()) &&
        pOther.GetIncludeHorizontalAlignment())
    {
        SetHorizontalAlignment(pOther.GetHorizontalAlignment());
    }
}```
// VerticalAlignment
if (!GetIncludeVerticalAlignment() && pOther.GetIncludeVerticalAlignment())
{
    SetVerticalAlignment(pOther.GetVerticalAlignment());
}

...

We provide a method that can fill up all the default settings for a style object. This method needs to have access to the style gallery. Before you fill up the style object with base style information, you should copy the style object, so that you do not overwrite the cells individual style object:

// Create an empty style object
CGXStyle* pStyle = CreateStyle();

// Copy the style settings for the specific cell
GetStyleRowCol(nRow, nCol, *pStyle, gxCopy, 0);

// Inherit attributes from base styles
pStyle->LoadBaseStyle(*GetParam()->GetStylesMap());

An extension to only loading style objects from style objects in the gallery object is to let style objects load default settings also from style objects outside the style gallery. For example, in a grid, you may want to apply changes to a column or row and not loop through all the cells in that row or column. By extending the Style Gallery pattern, you can also load data from row or column base styles as shown in the following code snippet:

const CGXStyle& ComposeStyleRowCol(ROWCOL nRow, ROWCOL nCol,
    CGXStyle* pStyle)
{
    ...
    // First, get changes for the individual cell
    GetStyleRowCol(nRow, nCol, *pStyle, gxCopy, 0);

    // Next, load column and row styles
    if (nRow > 0 && nCol > 0)
    {
        // Apply row style
        GetStyleRowCol(nRow, 0, *pStyle, gxApplyNew, -1);

        // Apply column style
        GetStyleRowCol(0, nCol, *pStyle, gxApplyNew, -1);

        // Apply table style
        if (nRow > 0 || nCol > 0)
            GetStyleRowCol(0, 0, *pStyle, gxApplyNew, -1);
    }

    // Finally inherit attributes from base styles
    pStyle->LoadBaseStyle(*GetParam()->GetStylesMap());
    ...
}
We have a collection of **user attributes** in the style class, each identified through a unique integer number. This supports data that is unique to every style without having to allocate unused space.

```cpp
class CGXStyle
{
  ...

  CGXUserAttributeMap* m_pmapUserAttributes;  // The programmer can use extra
  // attributes for derived CGXControls

public:
  // access methods:
  BOOL GetIncludeUserAttribute(WORD nID) const;
  CGXStyle& SetIncludeUserAttribute(WORD nID, BOOL b);
  const CGXAbstractUserAttribute& GetUserAttribute(WORD nID) const;
  void GetUserAttribute(WORD nID, CString& s) const;
  CGXStyle& SetUserAttribute(WORD nID, const CGXAbstractUserAttribute& attribute);
  CGXStyle& SetUserAttributePtr(WORD nID, CGXAbstractUserAttribute* pValue);
  CGXStyle& SetUserAttribute(WORD nID, const CString& s);
  CGXStyle& SetUserAttribute(WORD nID, LPCSTR pszValue);
  CGXStyle& SetUserAttribute(WORD nID, DWORD value);
  CGXStyle& SetUserAttribute(WORD nID, LONG value);
  CGXStyle& SetUserAttribute(WORD nID, double value);
  CGXStyle& SetUserAttribute(WORD nID, GXBYTE8 value);
};
```

`m_pmapUserAttributes` is the collection which holds all initialized user attributes for a style object. `GetIncludeUserAttribute()` determines if the attribute can be found in the collection. `SetUserAttribute()` initializes the attribute in the collection.

This lets us store any kind of data in the style object and manipulate it in a standard format. The data stored will have to be derived from `CGXAbstractUserAttribute` so that the grid can manipulate it. This object can have extra methods that are specific to that attribute. The grid uses user attributes extensively to store data that is specific to certain cell types.

### 12.8.1 The styles map

The `CGXStylesMap` is the object that holds all the base style objects.

```cpp
class CGXStylesMap: public CObject
{
public:
  // Base Style
  // Register a new style (or change an already registered style)
  WORD RegisterStyle(LPCSTR pszName, const CGXStyle& style);
  BOOL LookupStyle(WORD wStyleId, CGXStyle*& style) const;

protected:
  // base styles
  CMap<WORD, WORD, CGXStyle, CGXStyle&> m_BaseStyleMap;
  // associative map with unsigned integer as key
};
```
The RegisterStyle() method adds new style objects to the CGXStylesMap and returns an unsigned integer which can be used in the application as an unique identifier to refer to the base style object. LookupStyle() returns the base style object for the given identifier. m_BaseStyleMap() holds all the base style objects. Each style object is identified through an unsigned integer (which is returned by RegisterStyle()).

By default, three base styles are registered in the grid: Standard, Row Header and Column Header. The Standard style is the base style for all style objects.

The CGXStyle::LoadBaseStyle() method should be called from the grid on demand when it needs to have knowledge about a specific cell's state. LoadBaseStyle() loads the default settings from the base style and loops through the base styles until all uninitialized data is resolved.

```cpp
void CGXStyle::LoadBaseStyle(const CGXStylesMap& stylesmap)
{
    CGXStyle* pBaseStyle = this;
    while (pBaseStyle->GetIncludeBaseStyle())
    {
        // check out pointer to new basestyle
        if (!stylesmap.LookupStyle(pBaseStyle->GetBaseStyle(), pBaseStyle))
            break;

        // load default settings from base style
        ChangeStyle(*pBaseStyle, gxApplyNew);
    }

    ...
}
```

By using this unique style architecture, we could provide the functionality to let end-users categorize different types of cells and reuse this information among cells. We have also added dialogs to our grid library, so that the end-user can change add, remove, or change base styles. Any change of existing base styles will be reflected in all cells.
Chapter 13  
Objective Grid Concepts and Features

13.1 Programming Concepts

Objective Grid, as its name implies, is an object-oriented class library that is easily integrated into your applications. This section discusses some of the concepts from C++ that you will encounter as you start to use Objective Grid. For more information on any particular item, you should refer to a C++/MFC reference such as MFC Internals (Addison-Wesley Developers Press).

13.2 Extending Functionality in Objective Grid

The primary tool to change and/or add functionality to one of Objective Grid’s classes is subclassing. In your derived class, you can add new methods to gain functionality, and override existing virtual methods to change existing functionality. One of the hallmarks of Objective Grid is that its classes are rich in virtual functions so you can easily home in on exactly the behavior you want to modify.

Here is an example of how you can change the functionality of an existing class so it can handle things in a different manner. The CGXEditControl class implements a text input control that can be used to display and edit text in cells. Through CGXStyle, you can make this text right-justified, left-justified, or centered within the cell. But suppose you want to make sure there is a 2 pixel margin around the cell so the text will never be displayed within 2 pixels of the border (regardless of the alignment). One way to implement this behavior is to subclass CGXEditControl and override its member method GetCellRect(), which returns the interior rectangle for the cell.

```cpp
CRect CBorderedEditCtrl::GetCellRect(ROWCOL nRow, ROWCOL nCol, LPRECT rectItem, const CGXStyle* pStyle)
{
    // declarations
    class CBorderedEditCtrl: public CGXEditControl
    {
        GRID_DECLARE_CONTROL(CBorderedEditCtrl)
```
public:
CBorderedEditCtrl(CGXGridCore* pGrid, UINT nID);
virtual CRect GetCellRect(ROWCOL nRow, ROWCOL nCol, LPRECT rectItem = NULL, const CGXStyle* pStyle = NULL);
};

// implementation
GRID_IMPLEMENT_CONTROL(CBorderedEditCtrl, CGXEditControl)

CBorderedEditCtrl::CBorderedEditCtrl(CGXGridCore* pGrid, UINT nID):
    CGXEditControl(pGrid, nID) {}
{
    CRect r = GetCellRect( nRow, nCol, rectItem, pStyle);

    r.InflateRect(-2, -2); // deflate the rectangle

    return r;
}

Before you jump in and start subclassing one of Objective Grid's classes, you should always check
the class's description, as there may be a property setting that affects the behavior you want. If so,
you can avoid the work of subclassing and simply set the property.
13.3 Multiple Inheritance

Objective Grid uses multiple inheritance to allow aggregation of functionality. With multiple inheritance, a derived class has more than one base class. For example, \textit{CGXGridView} is derived from both \textit{CGXGridCore} and \textit{CGXView}. This means that \textit{CGXGridView} inherits methods and properties from both \textit{CGXGridCore} and \textit{CGXView}. The reason for using multiple inheritance is that it allows your classes to be more granular, letting you pick and choose the functionality you want in your derived class, without having unneeded methods or properties just because they happen to be a member of the base class. For example, in the diagram below, \textit{CGXGridHandleView} is derived from \textit{CGXView}, but not \textit{CGXGridCore}.

Figure 125 – Drawing classes hierarchy
13.4 Returning References

Take a look at this code. It sets the style properties for cell (4,2).

```cpp
SetStyleRange( CGXRange(4, 2),
    CGXStyle()
    .SetValue("Welcome")  //Text to be displayed
    .SetControl(GX_IDS_CTRL_STATIC)  //Static, no editing
    .SetVerticalAlignment(DT_VCENTER) //Center vertical
    .SetHorizontalAlignment(DT_CENTER)//Center horizontal
    .SetEnabled(FALSE)       // User cannot click on it
);   
```

At first glance, you might think the syntax is wrong. `SetStyleRange()` takes two arguments in the above call. The second argument is:

```cpp
CGXStyle()
    .SetValue("Welcome")
    .SetControl(GX_IDS_CTRL_STATIC)
    .SetVerticalAlignment(DT_VCENTER)
    .SetHorizontalAlignment(DT_CENTER)
    .SetEnabled(FALSE)
```

The methods `SetValue()`, `SetControl()`, `SetVerticalAlignment()`, `SetHorizontalAlignment()`, and `SetEnabled()` are all members of `CGXStyle`. Each of these methods has a return value of the `this` pointer. That is, each method returns a reference to its object. So the call to `CGXStyle().SetValue("Welcome")` returns a reference to the style object which allows it to call `SetControl(GX_IDS_CTRL_STATIC)`, which in turn returns a reference to the style object, which allows it to call `SetVerticalAlignment(DT_VCENTER)`, and so on. The code above is a shortcut for writing the code below. It is intelligible and makes it easier for the programmer by allowing him to write a single statement to initialize all the attributes instead of one statement for each attribute.

```cpp
SetStyleRange(CGXRange(4,2),
    CGXStyle().SetValue("Welcome");
SetStyleRange(CGXRange(4,2),
    CGXStyle().SetControl(GX_IDS_CTRL_STATIC));
SetStyleRange(CGXRange(4,2),
    CGXStyle().SetVerticalAlignment(DT_VCENTER));
SetStyleRange(CGXRange(4,2),
    CGXStyle().SetHorizontalAlignment(DT_CENTER));
SetStyleRange(CGXRange(4,2),
    CGXStyle().SetEnabled(FALSE));
```
13.5 **Automatic Linking**

Objective Grid automatically links the appropriate static library or DLL to your project. You indicate exactly what libraries you want to include by defining combinations of \_\_DEBUG, \_\_AFXDLL, and \_\_GXDLL in your **Configuration Properties/C++ Project Preprocessor** definitions (or in the **Project Options** box). The presence of \_\_AFXDLL indicates dynamic linking to MFC. If it is absent, the project is statically linked to MFC. Similarly, the presence or absence of \_\_GXDLL controls the linking of the Objective Grid Library.

The mechanics of this automatic linking are done in the file `Include/grid/config/gxstndrd.h`. This header file is included in `gxall.h` which you include in your `stdafx.h`. So any file that contains `stdafx.h` will include `gxstndrd.h`. This file uses a series of `#ifdef`s to conditionally define a name depending upon whether \_\_AFXDLL and \_\_GXDLL have been defined. The file then uses `#pragma comment(lib, _GXAUTOLIBNAME)` to tell the linker which library to use.

If you use the Build Wizard to define custom libraries, it places a new header file with the custom library name in the `Include/grid/config` folder. If you build a configuration named `mycustomlibrary`, you can have Objective Grid automatically link with `mycustomlibrary`, by including `config\mycustomlibrary.h` in your `stdafx.h` file.

13.6 **MFC Messages Versus Objective Grid Virtuals**

MFC uses a message-map architecture. If you want your MFC class to handle window messages, you derive it from `CCmdTarget`. A message map routes messages to the member functions you write to handle them. Generally, you use the MFC ClassWizard to add these message handlers to your classes. In this manner, a particular message ID is tied to a member method which handles it.

Objective Grid does not use a messaging architecture. Objective Grid classes define virtual methods that implement particular tasks. To change the functionality, you override the virtual method.

You should note that most Objective Grid classes do inherit from `CWnd`, so they do have message maps associated with them, but these messages are `CWnd` messages to accomplish `CWnd` tasks, and not part of the Objective Grid core architecture (which is implemented through class `CGXGridCore`.)
13.7 Objective Grid and Windows XP Visual Styles

13.7.1 Grids

13.7.1.1 Database Grids

The record browser window will be drawn with visual styles under Windows XP. In addition, when "Attach Foreign Table" is used, the resultant tabbed combo boxes in the grid cells will have visual styles enabled. Any other controls used, such as the Date/Time Control, Currency control, etc., will also be themed under Windows XP.

13.7.1.2 Hierarchical Grids

The tree control in the hierarchical grids will be drawn according to the theme enabled. The cell that contains the grid cell will be treated as the tree item and drawn accordingly.

13.7.1.3 Tabbed Grids

Both the dynamic and static tab classes (CGXTabWnd and CGXDTabWnd) are drawn with header visual styles under Windows XP.

13.7.2 Controls

Objective Grid contains many controls specifically designed to be embedded in grid cells. These controls will have the Windows XP look and feel when they are run under Windows XP. Applications should link to comctrl32.dll (version 6) using a manifest file.

When Windows XP themes are enabled, the behavior of these controls differs under XP and 2000:

- Separate 2D and 3D versions exist for certain controls (such as CGXCheckBox, CGXRadioButton, etc.) Under Windows XP, the 2D and 3D versions look and behave the same.

- When Windows themes are applied to controls, the control background is governed by the theme and hence can't be modified. For example, in CGXCheckBox there is a facility to replace the bitmaps with custom bitmaps. This will not be possible when the controls are themed. All the drawing for the custom controls is normally done in the Draw() method declared in CGXControl. Therefore, if the visual style for a particular control has to be turned off, use RWSetThemeAppProperties() as follows:
void CGridSample8CheckBox::Draw(CDC* pDC, CRect rect, ROWCOL nRow, ROWCOL nCol, const CGXStyle& style, const CGXStyle* pStandardStyle)
{
    RWSetThemeAppProperties(0);
    CGXCheckBox::Draw(pDC, rect, nRow, nCol, style, pStandardStyle);
    RWSetThemeAppProperties(STAP_ALLOW_NONCLIENT | STAP_ALLOW_CONTROLS | STAP_ALLOW_WEBCONTENT);
}

For detailed procedures on registering a control and overriding a control’s behavior, see Chapter 5, “Using Objective Grid.”

Call `RWSetThemeAppProperties()` to turn off themes. Remember to reset it at the right place; otherwise themes for the entire window may be turned off.

- The same technique can be used for `PrintPreview()`. If the controls with visual styles are not needed only for printing, then `RWSetThemeAppProperties()` can be called in `OnBeginPrinting()` and reset in `OnEndPrinting()` of the view class derived from `CGXGridView`.

```cpp
void CVirtGridView::OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: add extra initialization before printing
    RWSetThemeAppProperties(0);
    CGXGridView::OnBeginPrinting(pDC, pInfo);
}
void CVirtGridView::OnEndPrinting(CDC* pDC, CPrintInfo* pInfo)
{
    // TODO: add cleanup after printing
    CGXGridView::OnEndPrinting(pDC, pInfo);
    RWSetThemeAppProperties (STAP_ALLOW_NONCLIENT | STAP_ALLOW_CONTROLS | STAP_ALLOW_WEBCONTENT);
}
```

- All the controls in Objective Grid are derived from `CGXControl`. Unlike the standard Windows controls, the `MouseMove()` messages are not forwarded to these controls. Hence by default the controls will not hover when the mouse is moved over. However, some of the custom controls which have a `CWnd` associated with them, will hover when the mouse is moved over them.

Users can customize this behavior by deriving from these controls and overriding the `Draw()` Method. Using the standard windows controls through the `CGXWndWrapper` approach will also work. For detailed information, refer to Section 2.4, “Control Architecture.”

### 13.7.2.1 CGXCheckBox, CGXCheckBoxEx

The 2D and 3D versions of the CGXCheckBox looks pretty much the same on Windows XP. When drawn with Visual styles, it is not possible to replace the bitmaps with custom ones.

### 13.7.2.2 CGXCheckListComboBox

This control is drawn with visual styles for VC++ 7.1 or later. This control uses `CCheckListBox` from MFC as the base class.
13.7.2.3 CGXProgressCtrl

The foreground color and the background color of the progress bar can't be changed when visual styles are applied.

13.7.2.4 CGXRadioButton & CGXRadioButtonEx

The 2D and 3D versions look the same under visual styles in Windows XP.

13.7.2.5 CGXTabbedComboBoxWnd

The title bar is drawn with a visual styles-like header. In tabbed combo boxes with more than one display column, the active column is marked with the hot style.

13.7.3 Giving Objective Grid an Excel XP Look

13.7.3.1 Headers

When Excel-like headers are used in Windows XP with visual styles enabled, the headers behave like Excel XP. To get the exact look and feel of Excel XP, set solid grid lines by calling:

```c
GXSetNewGridLineMode(TRUE, PS_SOLID);
```

in the InitInstance() of the application.

If the GetParam()->SetExcelLikeHeaders() function is called from OninitialUpdate() with TRUE as the argument, the headers behave like Excel XP headers. If the visual styles are not enabled, then headers will behave like Excel 2000 headers.

13.7.3.2 Color of Selected Cells

Excel XP uses a different color (i.e. other than black) for filling blocked or selected cells. This can be done programmatically as well by overriding DrawInvertCell() and GetStyleRowCol().

13.7.3.3 Defining the Color of Cell Headers

You can define a custom color for the headers of selected cells. The coloring of cell headers is OFF by default. When the coloring of cell headers is turned ON, the default highlight color is orange (RGB 255, 180, 95).

Use the following functions to change the cell header to use colors other than the default, which are found in the CGXHeader class of gxctrl1.h:

```c
BOOL UsesCustomSelectionColor()
void SetUsesCustomSelectionColor(BOOL bEnable)
COLORREF GetCustomSelectionColor()
void SetCustomSelectionColor(COLORREF clr)
```
14.1 Introduction

14.1.1 What is the Objective Grid Formula Engine?

Formulas, the heart of any spreadsheet application, allow you to establish and calculate mathematical relationships among elements in the spreadsheet cells. While numeric entries remain the same until you change them, cells defined by formulas change automatically to reflect changes in referenced cells, even when there are complex interdependencies among cells.

Objective Grid formulas can calculate with numbers, text, logical values, cell references, and other formulas. For example, you can easily calculate the sum of a series of cells, the total of values in a column, a minimum or maximum value within a range, the rounded result of another formula, or the absolute value of a cell entry. Formulas can also express complex interdependencies among cells and they can define calculation constraints, such as limits on acceptable values or specific conditions under which a calculation should take place.

Once you enter a formula into a cell, the formula works behind the scene. The only trace of its existence is the result of the calculation. To view the formula in a cell, select the cell. You can edit the formula or the value in a cell at any time.

14.1.2 Implementing the Formula Engine

Because of the complex interdependencies among cells, Objective Grid treats the formula engine as a separate entity. As a front-end to this engine, Objective Grid loads values from the engine when cells need to be drawn and stores values or formulas in the engine whenever cells are changed. This is very similar to the "virtual" approach, in which you override certain grid methods like GetStyleRowCol() and StoreStyleRowCol() to browse and modify data of any external data source.

The process of feeding the engine with data and loading data from the engine allows the engine to track cell dependencies quickly. Whenever you change a cell, the formula engine updates each cell with formulas that reference the modified cell. The grid that is responsible for drawing the cells
asks the engine which cells need to be redrawn after you make a change to the grid. Cells that need to be redrawn are marked with flags so that no unnecessary drawing (flickering) occurs when a cell is changed. Only the cells that change values as a result of direct or indirect interaction are redrawn. **Direct interactions** are the result of user interaction and **indirect interactions** are the result of cell references in a formula.

The formula engine is accessed through the `CGXFormulaSheet` class, which is derived from `CGXData` and replaces the `CGXData` object that holds all the cell data. `CGXFormulaSheet` has special overrides for `GetStyleRowCol()` and `StoreStyleRowCol()` so that whenever cells are changed in the grid, the actual change is directly stored in the formula engine. Other operations—like moving, removing, and inserting cells—are also forwarded to the engine by overrides in `CGXFormulaSheet`.

When you use the engine in your grid, you should only call functions and methods provided by the `CGXFormulaSheet` class. This ensures that your code is readable and understandable when you interact with a class object. The `CGXFormulaSheet` class is a "wrapper" class to the engine. Consider the code for the engine itself as a separate entity. The product includes full source code for this engine, but you should not need to change the code of the engine or access functions from the engine. The engine itself is based on C-Code, which we intentionally ported to C++ classes. The engine code is not documented. Only the `CGXFormulaSheet` class is documented in the Objective Grid Class Reference.

The engine is extensible. Although you can’t change the grammar of the formula parser, you can add your own worksheet functions. If you need to add your own worksheet functions, use existing worksheet functions as a template when you need to work with the engine functions directly.

### 14.1.3 Adding formula support

Formula support is enabled if you call `EnableFormulaEngine()` at the beginning of your `OnInitialUpdate()` routine or before you call `CGXGridWnd::Initialize()`. The `EnableFormulaEngine()` call forces the grid to replace the default `CGXData` object with the `CGXFormulaSheet` object, which establishes a connection to the formula engine.

If you link your application statically with the Objective Grid libraries, the engine code is only linked into your application when you call `EnableFormulaEngine()`. This gives you the flexibility not to include the formula engine code in your executable, which reduces its footprint size.

After you add this line, you can fill the grid with `SetValueRange()`, `SetStyleRange()`, and other grid functions. To store formulas, numeric, or date values to cells, call `SetExpressionRowCol()`.

Example:

```cpp
void CGridSampleView::OnInitialUpdate()
{
    EnableFormulaEngine();
    BOOL bNew = ConnectParam();
    CMyGridView::OnInitialUpdate(); // Creates all objects and
    // links them to the grid
    // ... and you can execute commands on the grid
    if (bNew)
    {
        EnableHints(FALSE);
    }
```
// Lock any drawing
BOOL bOld = LockUpdate();

// initialize the grid data
// disable Undo mechanism for the following commands
GetParam()->EnableUndo(FALSE);

// no iteration when circular references
GetSheetContext()->SetIterationLimit(0);

// automatic/manual recalculation
GetSheetContext()->SetRecalcMode(GX_RECALC_AUTO);

// reevaluate cells on demand
GetSheetContext()->SetRecalcMethod(GX_RECALC_AS_NEEDED);

// turn off constraint checks
GetSheetContext()->SetConstraintCheck(FALSE);

// Initialize grid with 30 rows and 5 columns
SetRowCount(1000);
SetColCount(40);

// Insert an array with Numeric data
ROWCOL nRow, nCol;
double d = 0.0;
for (nRow = 7; nRow <= 12; nRow++)
{
    d *= 2.0;
    for (nCol = 1; nCol <= 4; nCol++)
    {
        d += 1.0;
        SetStyleRange(CGXRange(nRow, nCol),
        CGXStyle()
        .SetValue(d)
        .SetHorizontalAlignment(DT_RIGHT)
        );
    }
}

// Some string data
SetValueRange(CGXRange(7, 6), _T("Hello ") );
SetValueRange(CGXRange(7, 7), _T("world ") );
SetValueRange(CGXRange(8, 6), _T("Stingray ") );
SetValueRange(CGXRange(8, 7), _T("Software ") );
SetValueRange(CGXRange(9, 6), _T("Objective ") );
SetValueRange(CGXRange(9, 7), _T("Grid ") );
nRow++;
nRow++;
SetStyleRange(CGXRange(nRow, 1),
    CGXStyle()
    .SetValue(_T("String Functions"))
    .SetEnabled(FALSE)
    .SetFont(CGXFont().SetBold(TRUE))
    );
nRow++;
SetExpressionRowCol(nRow, 1, _T("STRCAT"));
SetExpressionRowCol(nRow+1, 1, _T("=STRCAT(F7, G7)"));
SetExpressionRowCol(nRow, 2, _T("LENGTH"));
SetExpressionRowCol(nRow+1, 2, _T("=LENGTH(F7)"));
SetExpressionRowCol(nRow, 3, _T("FIND"));
14.1.4 Storing expressions and values in cells

Objective Grid allows you to store data in a cell in several ways. You can call `SetValueRange()`, `SetStyleRange()`, or `SetExpressionRowCol()`.

`SetExpressionRowCol()` differs from `SetValueRange()` and `SetStyleRange()` in that it parses the string value. If the string value is a formula, it stores a formula. String values that are formulas are preceded by an equal sign (=). If the string value is a string, a string value is stored. If the string value is a number, a number is stored.

Date values are of special interest. If you pass a date or a time to `SetExpressionRowCol()` as a string (for example, "11/21/01"), Objective Grid interprets this string, stores a date value in the cell, and then sets the cell format to date. Objective Grid internally stores date and time information using the same convention as other popular spreadsheet programs: Dates are represented as an integer equal to the number of days since December 31, 1899. Time is represented as fractions of a day, starting at midnight. Cell formats are specified through `CGXStyle::SetFormat()` and the number of digits (for example, for scientific format) is specified via `CGXStyle::SetPlaces()`.

Because the formula engine distinguishes between numeric and string values, if you pass a numeric value as a string via `SetValueRange()`, `SetValueRange(..., "100")` for example, the number is stored and treated as a string. If you pass this string via `SetExpressionRowCol()`, the string is parsed and the number is recognized.

Whenever the user changes data in a cell through an edit cell, the data is stored in the grid by calling `SetExpressionRowCol()`. This allows the user to enter formulas in the current cell.

Here are some examples of how values can be interpreted:

```
SetValueRange(CGXRange(nRow, nCol), 122.22);
// this will set value type to GX_VT_NUMERIC (numeric)

SetValueRange(CGXRange(nRow, nCol), "122.22");
// this will set value type to GX_VT_STRING (string)

SetStyleRange(CGXRange(nRow, nCol), CGXStyle().SetValue ("122.22"));
// this will set value type to GX_VT_STRING (string)

SetStyleRange(CGXRange(nRow, nCol), CGXStyle().SetValue (929.2));
// this will set value type to GX_VT_NUMERIC (numeric)

SetExpressionRowCol (nRow, nCol, "1222.22");
// SetExpressionRowCol parses the string and
// will set value type to GX_VT_NUMERIC (numeric)
```
SetExpressionRowCol (nRow, nCol, "ABDGDDG1222.22");
// SetExpressionRowCol parses the string and
// will set value type to GX_VT_STRING (string)

SetExpressionRowCol (nRow, nCol, "12/21/02");
// SetExpressionRowCol parses the string and
// will store a date value and set the cell format to date.

SetExpressionRowCol (nRow, nCol, "=ABS(A5) ");
// SetExpressionRowCol parses the string and
// will store a formula expression.

## 14.1.5 Formula Engine Support in CGXGridCore

### 14.1.5.1 Access to CGXFormulaSheet

- **EnableFormulaEngine()** — Call this method in the beginning of your
  OnInitialUpdate() routine or before calling CGXGridWnd::Initialize() to enable formula support.

- **GetSheetContext()** — Call this method to get a pointer to the CGXFormulaSheet.

### 14.1.5.2 Access to Cell Data

GetStyleRowCol(), StoreStyleRowCol(), SetStyleRange(), and SetValueRange() support three nType attributes for expression values. You can pass the following values to nType:

- **const int GX_VALUE_BASESTYLE = -1;**
  Use for row, column, and table style.

- **const int GX_VALUE_TEXT = 0;**
  Use to store or retrieve plain style information in cell.

  When you use GX_VALUE_TEXT, GetStyleRowCol() returns the cell text in CGXStyle::GetValue(). That is the computed value of a formula cell.

  On the other hand, StoreStyleRowCol(), SetStyleRange(), and SetValueRange() assume the CGXStyle::GetValue() as plain text or a number and do not try to interpret the value when GX_VALUE_TEXT is specified.

- **const int GX_VALUE_EXPRESSION = 1;**
  Use when calling SetValueRange(), StoreStyleRowCol(), etc. and the value should be parsed (date, time, number, formula, etc.)

  When you use GX_VALUE_EXPRESSION, GetStyleRowCol() returns the formula itself in CGXStyle::SetValue(). When you use GX_VALUE_EXPRESSION, CGXStyle::GetValue() is interpreted and, if necessary, stored as formula.

SetExpressionRowCol() is a wrapper method to SetValueRange() that passes GX_VALUE_EXPRESSION as nType, which forces SetValueRange() to interpret the value. The method is declared as:
BOOL SetExpressionRowCol(ROWCOL nRow, ROWCOL nCol, const
CString& s, UINT flags = GX_UPDATE_NOW);

GetExpressionRowCol() is a wrapper method to GetValueRowCol() that passes
GX_VALUE_EXPRESSION as nType, thus forcing GetValueRowCol() to return the expression value
instead of the calculated value. The method is declared as:

const CString& GetExpressionRowCol(ROWCOL nRow, ROWCOL nCol);

### 14.1.6 Drawing

To support the interdependencies among cells, we added new cell attributes to Objective Grid to
identify grid cells that need to be refreshed after the user makes a change. When the user changes a
cell, this cell is added to a **recalculation list**, and all affected cells are marked with an update flag.
The formula engine uses this list to recalculate each cell that depends on the changed cells. The
CGXGridCore::RefreshViews() method loops through all visible cells and refreshes only the cells
marked with the update flag.

The following table lists methods that implement the update interface.

Table 9 – Methods Implementing the Update Interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetUpdateFlag</td>
<td>Returns update flag.</td>
<td>virtual BOOL GetUpdateFlag(ROWCOL nRow, ROWCOL nCol);</td>
</tr>
<tr>
<td>SetUpdateFlag</td>
<td>Set update flag. <em>(TRUE if cell shall be updated later.)</em></td>
<td>virtual void SetUpdateFlag(ROWCOL nRow, ROWCOL nCol, BOOL bToggle);</td>
</tr>
<tr>
<td>AddRecalcRange</td>
<td>Add cell to recalc list.</td>
<td>virtual void AddRecalcRange(const CGXRange&amp; range);</td>
</tr>
<tr>
<td>RefreshViews</td>
<td>Loop through visible cells and redraw cells with update flag.</td>
<td>virtual BOOL RefreshViews(BOOL bRecalc = TRUE, CString* pErrorStr = NULL, BOOL bCreateHint = TRUE);</td>
</tr>
<tr>
<td>Recalc</td>
<td>Force recalculation either of all formula cells or only of cells depending on cells in the recalc list.</td>
<td>virtual void Recalc(BOOL bForceRecalc = FALSE);</td>
</tr>
<tr>
<td>ResetError</td>
<td>Resets error state of the engine.</td>
<td>virtual void ResetError();</td>
</tr>
<tr>
<td>GetError</td>
<td>Checks if there is an error. <em>(OG calls this method and will display a message box if GetError returned TRUE.)</em></td>
<td>virtual BOOL GetError(CString&amp; strError);</td>
</tr>
</tbody>
</table>

14.1.7 Clipboard

The following flag has been added for `m_nClipboardFlags` to support formulas:

```cpp
const DWORD GX_DNDEXPRESSION = 0x800; // Copy Expression instead of
   // value (when GX_DNDESTYLES // is set)
```

This setting has no effect on direct copy/paste in sheet or CF_TEXT clipboard format.

Support for "direct" Copy and Paste or Cut and Paste is specified through:

- BOOL `m_bDirectCopyPaste`;
- BOOL `m_bDirectCutPaste`;
- BOOL `m_bDirectDragDrop`;

When you use direct Copy (or Cut) and Paste, the data is not being copied/loaded from the clipboard. Instead, the grid stores a reference to the original cells and marks them to be copied. When the user pastes the cells, Objective Grid calls either `MoveCells()` or `CopyCells()` to move or copy the cells directly within the sheet.

When you use direct Copy/Paste, the engine automatically updates any cell references to the moved cells. Direct Copy/Paste only works within one worksheet. If you copy/paste or drag/drop cells to another worksheet, the conventional method of using the clipboard is used instead and no cell references will be updated.

The following methods implement direct cut/copy/paste:

- virtual BOOL `OnPasteDirect(const CGXRange& range)`;
- virtual BOOL `StoreCopyCells(CGXRange rg, ROWCOL nRow, ROWCOL nCol)`;
- virtual BOOL `StoreMoveCells(CGXRange rg, ROWCOL nRow, ROWCOL nCol, CObject* pUndoInfo, BOOL bCreateUndoInfo, GXCmdType ctCmd)`;
- virtual BOOL `CopyCells(CGXRange rg, ROWCOL nRow, ROWCOL nCol, UINT flags = GX_UPDATENOW)`;
- virtual BOOL `MoveCells(CGXRange rg, ROWCOL nRow, ROWCOL nCol, CObject* pUndoInfo = NULL, UINT flags = GX_UPDATENOW, GXCmdType ctCmd = gxDo)`;

When direct cut/paste is enabled, the cells are not cleared until the user performs a paste operation in the grid. When cells are marked to be cut, they are outlined with a gray background pattern.

Other hooks are provided via:

- virtual BOOL `GetClipboardStyleRowCol(ROWCOL nRow, ROWCOL nCol, CGXStyle* pStyle, BOOL bLoadBaseStyles)`;
- virtual CString `GetCopyTextRowCol(ROWCOL nRow, ROWCOL nCol)`;
- virtual BOOL `PasteTextRowCol(ROWCOL nRow, ROWCOL nCol, const CStrings& str, UINT nFlags, const CGXStyle* pOldStyle);`
These methods let you hook into the process of supplying data for a specific cell or pasting data into a specific cell. The methods are called from within `PasteTextFromBuffer()`, `CopyTextToFile()`, and `PasteCellsFromArchive()`.

Full clipboard support for BIFF 8 (Excel 97) is provided in Objective Grid. The current implementation has a couple of restrictions, however:

- When formulas are copied from Excel to Objective Grid only the values that are the result of the Excel formulas are copied over to Grid; the formulas themselves are not copied over.
- Copying formulas from Grid to Excel is supported; formulas are copied, but because of syntax differences, Excel might not support all of the Objective Grid formulas.

### 14.1.8 Undo/Redo

Objective Grid supports multi-level Undo/Redo for all operations in the engine. For example, if cell references are updated because of a move operation, Undo takes care of these changes.

#### 14.1.8.1 Changes that cannot be undone

The only changes that cannot be undone are changes that are the result of an embedded tool or matrix operation because embedded tools store values in a group of adjacent cells. These adjacent cells are set to constant formulas with explicit dependencies on their neighboring cells. For example, if you have a matrix in A7..C9 and cell D10 contains the matrix function 
"=TRANSPOSE(A7..C9)", the cells D10..F12 are the result set of the matrix operation.

If the user changes any cell in the range D10..F12, the change is stored as a value in the cell. If the user then changes a value in the range A7..C9, the cells D10..F12 are overwritten and no Undo information for the manually changed cells is generated.

### 14.1.9 Display Formula Expression in Cells

The following methods control the display of formulas or their resulting values in cells:

- `CGXControl::GetControlText()` — This method checks the `CGXStyle::GetFormat()` setting and will try to format the value accordingly.
- `CGXControl::SetControlText()` — This method will call `SetExpressionRowCol()` instead of `SetValueRange()`, thus allowing you to paste formulas into cells from a text file. (`SetControlText()` is called from `PasteTextFromBuffer()`.)
- `CGXEditControl::SetValue()` — `CGXEditControl` will display the expression of the cell when in edit mode and the calculated value when the cell is inactive. When the user modifies the cell, `SetExpressionRowCol()` will be called to store the cell contents. This allows the user to enter formulas in cells.
You can control the behavior of the `GetControlText()` and `SetValue()` by changing `GetParam()->m_nDisplayExpression`. This toggles the display of formula expressions in inactive cells:

- **GX_EXPR_DISPLAYALWAYS** will display the formula expression in inactive cells.
- **GX_EXPR_DISPLAYACTIVE** will display the formula expression only in the active current cell.
- **GX_EXPR_DISPLAYNEVER** will display no formula expressions at all.

The default setting for a default grid without formula support is **GX_EXPR_DISPLAYNEVER**. When using the formula engine, the default setting is **GX_EXPR_DISPLAYACTIVE**.

With this setting, `CGXEditControl::SetValue()` will behave differently and, based on the needs of your subclassed cell types, you will have to override this method and modify its behavior.

### 14.1.10 Formula Engine Limitations

When using the formula engine, please note the following limitations:

- The maximum number of columns is 4096.
- The maximum number of rows is one million rows.
- The maximal text size for a cell is 512 bytes.
- The engine has to be filled with data. You cannot override `GetStyleRowCol()` and return values on demand. Of course, you can override `GetStyleRowCol()` and return a value, but then this value is unknown to the engine and cannot be used in formula calculations. Overriding `GetStyleRowCol()` and setting other style settings like color and font is no problem.
- The engine can't be used with ODBC or ADO grids.
- Formulas can only refer to cells within the same sheet. There is no support for getting values from other sheets.
14.2 CGXFormulaSheet Class

The CGXFormulaSheet class acts like an interface to the Objective Grid formula engine. CGXFormulaSheet can be attached to a CGXGridCore object and replace the CGXData object that holds all the cell data. CGXFormulaSheet has special overrides for GetStyleRowCol() and StoreStyleRowCol() that ensure that cell changes are stored directly in the formula engine. In addition, other operations such as moving, removing, and inserting cells are forwarded to the engine by overrides in CGXFormulaSheet.

14.2.1 Cell Values

The tables in this section (Table 10 through Table 18) list and describe the methods provided by the CGXFormulaSheet class.

Table 10 – Cell Value Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetFormulaRowCol</td>
<td>Assigns a formula expression to a cell. The formula must start with an equal sign (=).</td>
</tr>
<tr>
<td>SetTextRowCol</td>
<td>Assigns a text label to a cell.</td>
</tr>
<tr>
<td>GetTextRowCol</td>
<td>Returns the text value for a cell. Numeric values are converted to strings. Formulas are reevaluated if necessary. Cells with error state will return &quot;Error!&quot; as text.</td>
</tr>
<tr>
<td>SetNumberRowCol</td>
<td>Assigns a number to a cell.</td>
</tr>
<tr>
<td>GetNumberRowCol</td>
<td>Returns the numeric value for a cell. Text label returns 0. Formula is reevaluated when necessary.</td>
</tr>
<tr>
<td>SetFormatRowCol</td>
<td>Changes the numeric formatting of a cell.</td>
</tr>
<tr>
<td>GetFormatRowCol</td>
<td>Returns the numeric formatting to a cell.</td>
</tr>
<tr>
<td>SetExpressionRowCol</td>
<td>Parses the given text and (based on the text content) assigns a formula, a text label, or a number to the specified cell. To force numeric or date values to be stored as string, insert an apostrophe (') at the beginning of the text. Formulas must start with an equal sign (=).</td>
</tr>
<tr>
<td>GetExpressionRowCol</td>
<td>Returns either the text value or formula for the cell. Numeric values will be converted to a string. Text label cells will have a leading apostrophe (') to identify them as string.</td>
</tr>
<tr>
<td>GetUpdateFlag</td>
<td>Checks if the given cell needs to be redrawn (e.g. if formula result has been changed). CGXGridCore::RefreshViews checks GetUpdateFlag() for all visible and refreshes them if GetUpdateFlag() returns TRUE.</td>
</tr>
<tr>
<td>SetUpdateFlag</td>
<td>Forces the given cell to be redrawn with the next RefreshViews() call.</td>
</tr>
</tbody>
</table>

CGXGridCore::SetValueRange and CGXGridCore::SetExpressionRowCol() call CGXFormulaSheet::SetExpressionRowCol() after they ensure that the cell is not read-only and notify the cell object of the change.
Call the above cell value methods (SetNumberRowCol() and SetTextRowCol()) directly if you want to initialize the grid with a large data-set. It is much faster than calling SetValueRange() or SetExpressionRowCol() for each cell. Be aware that if cells are read-only, they are not checked, cell objects are not notified, and Undo information is not created.

Example:

```
// Performance tests:
// Using SetNumberRowCol/SetTextRowCol directly
// instead of SetValueRange or SetExpressionRowCol
// will speed up the initialization of the grid
// enormously (just as fast as filling an array).
//
// Check it out below!
//
// NOTE: Directly calling these methods will bypass the
// notification of the associated cell type object for a
// cell (CGXControl::StoreStyle will not be called.) and
// the read-only state of the cell will also not be
// checked.
//
DWORD ti = GetTickCount();
CGXFormulaSheet* pSheet = GetSheetContext();
CGXStyle style;
for (; nRow < 300; nRow++)
{
    for (ROWCOL nCol = 1; nCol < 10; nCol++)
    {
        // CString s;
        // s.Format("%d/%d", nRow/100, nCol);
        // style.SetValue("Hello");
        // StoreStyleRowCol(nRow, nCol, &style, gxOverride, 0);
        pSheet->SetNumberRowCol(nRow, nCol, (double) nRow+nCol);
        // pSheet->SetTextRowCol(nRow, nCol, _T("Hello"));
    }
}
CString msg;
msg.Format("%d Ticks", GetTickCount()-ti);
AfxMessageBox(msg);
```

### 14.2.2 Row and Column Coordinates

The following table lists methods that can be used to convert row and column coordinates.

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetEngineRow</td>
<td>Converts a grid row coordinate (of type ROWCOL, zero-based) into an engine row coordinate (of type int, zero-based).</td>
</tr>
<tr>
<td>GetEngineCol</td>
<td>Converts a grid column coordinate (of type ROWCOL, zero-based) into an engine column coordinate (of type int, one-based).</td>
</tr>
</tbody>
</table>
14.2.3 Sheet Settings

The following table contains methods used to modify sheet settings.

Table 12 – Sheet Setting Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetIterationLimit</td>
<td>When the Iteration Limit is set to a non-zero value, iterative calculation is enabled. In this mode, Objective Grid makes multiple recalculation passes and preserves the natural order and minimal recalculation rules until either the iteration limit has been reached or, if constraint checking is enabled, until all constraints are satisfied, whichever comes first.</td>
</tr>
<tr>
<td>GetIterationLimit</td>
<td>Returns the Iteration Limit value.</td>
</tr>
<tr>
<td>SetBlanksAreZeros</td>
<td>Specified if blank cells should be treated as a numeric 0 in formula expressions.</td>
</tr>
<tr>
<td>GetBlanksAreZeros</td>
<td>Returns TRUE if blank cells should be treated as a numeric 0 in formula expressions.</td>
</tr>
<tr>
<td>SetRecalcMode</td>
<td>Objective Grid lets you decide whether to recalculate the spreadsheet after you make changes that affect other cells or whenever the sheet is initially loaded. Objective Grid automatically recalculates the spreadsheets in both cases if Recalculation Mode is set to Automatic. If recalculation is too time-consuming, you can switch off automatic recalculation so Objective Grid will only recalculate upon explicit request.</td>
</tr>
<tr>
<td>GetRecalcMode</td>
<td>Returns the recalc mode setting.</td>
</tr>
<tr>
<td>SetRecalcMethod</td>
<td>Recalculation Method defines how Objective Grid proceeds with recalculation. There are two options: As Needed and Foreground. As Needed indicates that recalculation is highly optimized to recompute the minimum number of cells when updating the sheet. This can result in a significant performance improvement for certain types of sheets, particularly on very large sheets.</td>
</tr>
<tr>
<td>GetRecalcMethod</td>
<td>Returns the recalculation method setting.</td>
</tr>
</tbody>
</table>
14.2.4 Recalculation

The following table lists methods used for recalculation.

Table 13 – Recalculation Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddRecalcRange</td>
<td>Marks a given range of cells so that formula cells that depend on the marked cells can later be reevaluated. The formula engine maintains a recalc range list where all these ranges are added.</td>
</tr>
<tr>
<td>Recalc</td>
<td>Recalculates all cells in the worksheet that depend on cells in the recalc range list.</td>
</tr>
<tr>
<td>ResetError</td>
<td>Resets the error state of the formula engine.</td>
</tr>
<tr>
<td>GetError</td>
<td>Returns the error state of the formula engine. Objective Grid calls this method and when necessary displays a message box with error information based on the return value of this GetError method.</td>
</tr>
</tbody>
</table>

14.2.5 Named Ranges

The following table contains methods used with named ranges.

Table 14 – Named Range Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetRangeName</td>
<td>Adds a named range or replaces an existing named range with the new range settings.</td>
</tr>
<tr>
<td>GetRangeName</td>
<td>Returns the associated range for a given range name.</td>
</tr>
<tr>
<td>DeleteRangeName</td>
<td>Deletes a named range from the range name list.</td>
</tr>
<tr>
<td>GetRangeNameArray</td>
<td>Returns a string array with all named ranges. Each entry in the array is in the form 'name=A1..B5'.</td>
</tr>
<tr>
<td>SetRangeNameArray</td>
<td>Adds a batch of named ranges from a string array. Each entry in the array is in the form 'name=A1..B5'.</td>
</tr>
</tbody>
</table>
14.2.6 Range Specification

The following methods convert range specifications.

Table 15 – Range Specification Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextToRange</td>
<td>Converts a range specification, e.g. ‘A1..C5’, into a CGXRange object, e.g. CGXRange(1,1,5,3).</td>
</tr>
<tr>
<td>RangeToText</td>
<td>Converts a CGXRange object, e.g. CGXRange(1,1,5,3), into a range specification, e.g. ‘A1..C5’.</td>
</tr>
</tbody>
</table>

Keep in mind the following points about these range specification methods:

- The range object must not be a row, column, or table range. Don’t use SetRows(), SetCols(), or SetTable() for specifying the range. Use only plain cell ranges.

```cpp
CGXRange rg;
TextToRange(_T("A1..C5"), rg);
```

- You can use CString functions to transform a string that you get as a result of RangeToText(). For example:

```cpp
CString s;
CGXFormulaSheet::RangeToText(CGXRange(1,1,10,4),
   s.GetBuffer(20), 20);
s.ReleaseBuffer();
```

You can use string functions to convert absolute cell references in formula cells to the appropriate form.

See Section 14.5.5, “Cell Referencing in Objective Grid,” for more information about absolute and relative cell references. See Section 14.3.6, “Ranges,” for more information about ranges.

14.2.7 Expression Evaluation

The following methods are used for parsing expressions.

Table 16 – Expression Evaluation Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>EvaluateExpression</td>
<td>Parses and evaluates the given expression and returns the result as a string.</td>
</tr>
<tr>
<td>ParseExpression</td>
<td>Parses the given expression and returns the parsed formula expression in binary format.</td>
</tr>
</tbody>
</table>
14.2.8 Clipboard Cut/Copy and Paste

Methods associated with the clipboard include:

**Table 17 – Clipboard Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>StoreCopyCells</td>
<td>Copies cells from source range to a destination range. Cell references in formula expressions (unless marked as absolute range) will be adjusted.</td>
</tr>
<tr>
<td>StoreMoveCells</td>
<td>Moves cells from source range to a destination range. Cell references in formula expressions that depend on cells in the moved range will be adjusted.</td>
</tr>
</tbody>
</table>

14.2.9 Cell Operations

The following table lists cell operation methods.

**Table 18 – Cell Operation Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetColCount</td>
<td>Returns the number of columns.</td>
</tr>
<tr>
<td>GetRowCount</td>
<td>Returns the number of rows.</td>
</tr>
<tr>
<td>StoreColCount</td>
<td>Specifies the number of columns.</td>
</tr>
<tr>
<td>StoreRowCount</td>
<td>Specifies the number of rows.</td>
</tr>
<tr>
<td>GetStyleRowCol</td>
<td>Looks up a style in the formula engine. If called with nType = GX_VALUE_EXPRESSION the formula expression (if available) is assigned to the style value. Otherwise, the evaluated cell value (the result) is assigned to the cell value.</td>
</tr>
<tr>
<td>StoreStyleRowCol</td>
<td>Stores a style into the formula engine. If called with nType = GX_VALUE_EXPRESSION, the style value is parsed by calling SetExpressionRowCol. Otherwise, the style value is assigned to the cell as a text label (by calling SetTextRowCol).</td>
</tr>
<tr>
<td>StoreInsertCols</td>
<td>Inserts columns in the formula engine and updates cell references to moved cells.</td>
</tr>
<tr>
<td>StoreInsertRows</td>
<td>Inserts rows in the formula engine and updates cell references to moved cells.</td>
</tr>
<tr>
<td>StoreMoveCols</td>
<td>Moves columns in the formula engine and updates cell references to moved cells.</td>
</tr>
<tr>
<td>StoreMoveRows</td>
<td>Moves rows in the formula engine and updates cell references to moved cells.</td>
</tr>
</tbody>
</table>
### Table 18 – Cell Operation Methods (Continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>StoreRemove-Cols</td>
<td>Removes columns in the formula engine and updates cell references to moved cells.</td>
</tr>
<tr>
<td>StoreRemove-Rows</td>
<td>Removes rows in the formula engine and updates cell references to moved cells.</td>
</tr>
<tr>
<td>StoreMove-DataCols</td>
<td>Rearranges the columns in the grid. The new order of columns is specified through the awColIndex array. Cell references to moved cells are updated.</td>
</tr>
<tr>
<td>StoreMove-DataRows</td>
<td>Rearranges the rows in the grid. The new order of rows is specified through the awRowIndex array. Cell references to moved cells are updated.</td>
</tr>
</tbody>
</table>
14.3 Cells and Ranges

Objective Grid accepts several basic types of cell entries: text, constant numeric values, dates, time, formulas that calculate a value, and graphs. Calculated values can be single numbers, strings, arrays, or tables of values.

14.3.1 Text Cell Values

Text entries are useful for:

- Labeling columns and rows.
- Including comments about data values being calculated.
- Using Objective Grid to manage textual information such as names, addresses, or whatever your application may require.

14.3.1.1 Entering Text

A text entry is usually used for descriptive information such as a column heading, a row label, or an explanatory comment. You can also make text entries of names, addresses, or other non-numeric information. Objective Grid treats text and numeric entries differently, so you can differentiate between text and numeric values when entering cell contents.

To enter most text into a cell, just type the text string as you want it to appear or programmatically call `SetExpressionRowCol()`. If the text you wish to enter would normally be considered a formula, then you must precede the actual text with the `'` (apostrophe) character.

If you make an entry that is not text but which returns an error when treated as a formula, Objective Grid displays an error message. Section 14.8.2, “Error Messages Reference,” contains an alphabetized summary of error messages.

14.3.1.2 Treating Numeric or Date Entries as Text

Note that some numeric entries, such as phone numbers, should really be treated as text. For example, if you enter the phone number 555-1212 in a cell, Objective Grid displays the number -657, the difference of 555 and 1,212. However, if you start the string with the apostrophe (`'555-1212`, for example), Objective Grid treats the phone number as text and does not calculate it.

14.3.1.3 Entering Special Characters

Special characters can be entered as text by using the back-slash convention (`\`). (This technique is compatible with ANSI C escape codes.) The most common use of this is to force explicitly a new line in a text cell by entering `\n`. Other useful ones are `\` which allows the double quote character within a string and `\\` which allows the back-slash character.
14.3.2 Numeric Values

If a cell entry begins with a digit (from 0 to 9), Objective Grid treats the entry as a numeric entry.

Objective Grid also recognizes the following symbols as indicators of numeric entries: ., +, and -. You can format numeric values to be displayed in several ways, including fixed formats, scientific notation, currency, and hexadecimal.

14.3.2.1 Entering Numeric Values

Follow these conventions for entering numeric values:

- To enter a positive number, use the number keys to type the number, with or without a + indicator. If you do not type a plus (+), Objective Grid assumes the number is positive.

- To enter a negative number, type a minus sign (-) and the number. Do not use parentheses to indicate negatives. However, if you change the numeric format to Dollars or Comma, Objective Grid displays negative numbers in parentheses.

- Do not use spaces or commas when entering numbers. You can display commas, if you want to, by changing the format.

- Be careful not to substitute a lower case L for the numeral 1 or the upper case O for the numeral 0.

- You can use scientific notation to enter a number, with the convention that the letter e separates the fraction from the base 10 exponent. 1.23e3 is equivalent to 1230.

14.3.2.2 Changing the Numeric Format

You can choose from a variety of formats for displaying numbers. The display formats do not change the number itself, the way the number is stored internally, or the way it is used in calculations. Formatting just changes the way Objective Grid displays numbers.

See Section 5.14, “Number Formatting,” for information on using the CGXStyle::SetFormat() API. The end-user might change the number formatting with the CGXStyleSheet dialog.

14.3.3 Dates and Times

Objective Grid provides special, built-in features for displaying date entries in the format you choose. Date and time formats include: 24-Oct-99, 24-Oct, 10/24, Oct-99, 10/24/99, 24.10.1999, and 1999-10-24 (ISO 8061). Time is displayed as: 12:00:05.

14.3.3.1 Entering Date and Time Values

If you entered a date or time as a string of numbers separated by hyphens, (for example, 10-31-99) Objective Grid would interpret this string, store a date value in the cell and then set the cell format to date.
If the year (yy) is omitted when entering a date into a cell, the current year is assumed. For example, entering /7/17 is equivalent to entering /7/17/01 when the current year is 2001.

Objective Grid internally stores date and time information using the same convention as other popular spreadsheet programs: Dates are represented as an integer equal to the number of days since December 31, 1899. Times are represented as fractions of a day, starting at midnight. For example, 6:00 AM is stored as 0.25 (a quarter of a 24-hour day). Using this convention, date and time values may be used together. For example, the date/time value 1.25 corresponds to 6:00:00 AM, January 1, 1900.

You can use date values in calculations. For example, subtracting 05-Oct-99 from 10-Oct-99 yields a value of 5 days.

### 14.3.4 Formulas

Formulas establish and calculate mathematical relationships between elements of the spreadsheet. Objective Grid formulas can calculate with numbers, text, logical values, cell references, and other formulas. For example, you can easily calculate the sum of a series of cells, the total of values in a column, or the absolute value of another cell entry.

### 14.3.4.1 Entering Formulas

Formulas are the heart of the Objective Grid spreadsheet, defining relationships between the values in other cells. For example, formulas can be used to sum a column of numbers, compute an average, determine a minimum or maximum value, or round the results of a calculation.

While constant entries remain the same until you change them, cells defined by formulas are automatically changed to reflect changes in referenced cells—even when there are complex interdependencies among cells.

Once entered in a cell, formulas are hidden behind the scenes, performing their work in the background and displaying only the result of their calculation. To view the formula in a cell, move the cell cursor to the cell and switch the cell into edit mode. You can edit the formula or values within the cell.

See also Section 14.1.9, “Display Formula Expression in Cells.”

### 14.3.5 Built-in Functions

The Objective Grid Formula Engine comes with over 240 built-in worksheet functions. Functions can be used alone or in combination with formulas and other functions. Objective Grid was designed for demanding users, so it provides many highly specialized functions for business, engineering, scientific, and statistical applications. Sometimes you might want to provide only a subset of these worksheet functions to the end-user, no worksheet functions at all, or add your own worksheet functions.

The `CGXWorksheetFunctions` class provides an extensible interface to the formula engine where you specify what worksheet functions should get linked into your application and where you can also add your own custom worksheet functions to the engine.
**CGXWorksheetFunctions** itself implements no worksheet functions. This makes sure that none of the built-in functions will be linked into your executable and waste disk space unless you really need them.

In order to enable all built-in functions, you should call `GXEnableWorksheetFunctions()` from your application’s `OnInitInstance()` method. `GXEnableWorksheetFunctions()` instantiates the `CGXDefaultWorksheetFunctions` class. This class has a lookup table with all built-in worksheet functions.

If you want to add your custom functions, you should subclass `CGXWorksheetFunctions` (or `CGXDefaultWorksheetFunctions`) and override `LookupFunction()`. In your `OnInitInstance()` method you can register your class with:

```cpp
GXGetEngineState()->SetWorksheetFunctionClass(new CMyWorksheetFunctions);
```

*CMyWorksheetFunctions* is the class that you derived from `CGXWorksheetFunctions`.

---

Implementing your own custom worksheet functions is only recommended for advanced users. When implementing your own custom worksheet functions you will have to deal directly with the formula engine code, which is not documented at all. Take a look at the existing worksheet functions (e.g. in `src\grid\engine\fnmath.cpp`) for examples of how to implement worksheet functions.

### 14.3.5.1 Tips

When writing functions you can get a pointer to the sheet context, to the `CGXFormulaSheet`, and to the `CGXGridCore` object using the following methods:

- Based from the sheet context id (typically called si when passed to a C-Function) you can get a pointer to the `_gx_sheet_context` object with:
  ```cpp
gxSheetContext * context = _gx_get_context_ptr(si);
```
- Based on this context you can get a pointer to the associated `CGXFormulaSheet` by calling:
  ```cpp
  CGXFormulaSheet* pSheet = context->m_pSheetContext;
  ```
- You can also get a pointer to an associated `CGXGridCore` object by calling:
  ```cpp
  CGXGridCore* pGrid = pSheet()->Grid();
  ```
- You can call `pGrid->GetStyleRowCol` and any other grid functions.

### 14.3.6 Ranges

A **range** is a contiguous, rectangular block of cells that has been referenced as a group in a cell formula or selected to be affected by some editing action, such as Copy Formulas or Move.

#### 14.3.6.1 Creating a Range Object

A range object can either be initialized as an empty range, initialized with another range, or specified by coordinates. These examples show some alternatives for creating a range object.
14.3.6.2 Named Ranges

Individual cells and cell ranges may be referred to by their standard address coordinates (A5..D25, for example) or by pre-assigned names. The Named Range option lets you assign a name to any cell or cell range.

A named range is a range of cells to which a name has been assigned with the Named Range utility. A named cell is a cell to which a name has been assigned. Both named ranges and named cells can be referenced by their names or by their addresses.

Using names can help clarify the logic of your spreadsheet, making it easier to share among users, and easier to update long after it was originally designed. Once defined, names can be used anywhere standard cell or range references are used, including in cell formulas.

The following guidelines apply to named ranges:

- Objective Grid does not differentiate between uppercase and lowercase letters in range names. For example, Objective Grid would consider "Range1" and "range1" to be the same name.

- Names must begin with an uppercase or lowercase alphabetic character. The rest of the name may include any combination of alphabetic characters, numeric characters, and the following characters: $, ., or _.

- When an area containing a named range or cell is moved, Objective Grid automatically updates the definition of the name to reflect the new location.

- Named cells and ranges can have relative or absolute addresses. When cells containing references to named cells with absolute addresses are moved, the references are not updated.

- When cells containing a reference to a named cell or range are copied, Objective Grid converts the reference to an appropriately translated standard reference.

- If you redefine a name, all instances of that name in the spreadsheet are updated as soon as the spreadsheet is recalculated.

- If you remove a name definition, all references to that name in the spreadsheet are converted to appropriate standard references.

- To name a cell or range call the SetRangeName() command:

  ```
  SetRangeName(_T("number"), CGXRange(6,1,10,4));
  ```
14.3.7 Copying and Moving Cells

14.3.7.1 Copying Formulas

The Copy operations, clipboard copy and OLE Drag-and-Drop, duplicate a cell or range of cells in addition to all the formatting and then place these formulas in a new location, overwriting existing data in the destination range. Objective Grid automatically translates relative cell references in the copied formulas to reflect their new locations. For example, if cell A10 contained the formula =$SUM(A1..A9) and you copied it to cell B10, then B10 would contain the formula =$SUM(B1..B9).

To create an absolute cell reference, which is a value that Objective Grid doesn’t translate if the cell contents are copied or moved, insert a dollar sign ( $ ) before each component of the cell reference (e.g., $A$1). For more information, refer to Section 14.5.2, “Formula Values,” and Section 14.5.5, “Cell Referencing in Objective Grid.”

Use the CopyCells() method to perform a copy operation programmatically.

Objective Grid overwrites any existing data in destination ranges, so plan carefully before carrying out Copy and Move operations. If you overwrite data by mistake during a Copy or Move, you can put it back like it was using Undo.

14.3.7.2 Moving Data

The Move operation moves a cell or range of cells to a new location, along with all formulas. Objective Grid clears the source cells and overwrites any existing data in the destination cells. Like the Copy Formulas operation, all cell references are updated to reflect the new cell/range location.

Use the MoveCells() method to perform a move operation programmatically.

The effects of moving cells include:

- When you move a cell that is referenced by a formula, Objective Grid automatically updates the formula for the new location, even if the cell reference is absolute.

- When you move a cell in a range that is referenced by a formula, the formula still references the original range. However, if you move a corner cell of the range, Objective Grid extends or contracts the range to match the new location.

- If you move a cell range that has been named or referenced in a formula, Objective Grid automatically updates the definition of the range name or the formula reference to reflect the new location.

- If you move part of a cell range that has been named or referenced in a formula, it can cause problems. Objective Grid updates range names and references only when you move one or more corner cells of the range. If you move cells in the interior of the range, Objective Grid does not change the range name or any references to it.

If you make a mistake when copying or moving data, you can use Undo to restore the spreadsheet to its state prior to the copy or move operation.
14.3.7.3 Deleting Rows and Columns

If you delete a row or column that contains a cell that is referenced by a formula in another cell, the formula reference will not be modified. Range references will be updated when rows or columns are deleted within a range specification.

14.4 Calculations

This chapter explains Objective Grid calculations, including:

- How Objective Grid calculates,
- How to change calculation options such as automatic recalculation and iteration limit, and
- Elements of formulas.

14.4.1 How Objective Grid Calculates

Objective Grid uses natural order recalculation, which is a method of ordering the computations such that each cell’s value is computed only after the values for all cells on which it depends have been computed. Natural order recalculation guarantees that cells are always computed correctly, regardless of the order in which the cell formulas were entered. Minimized grid windows are recalculated the same as visible sheets.

Objective Grid supports several recalculation options:

- Mode (Manual or Automatic)
- Method (Foreground or As Needed)
- Iteration Limit
- Constraint Checking
- Precision

14.4.2 Mode of Recalculation

Objective Grid supports two modes of recalculation: Manual and Automatic. When Manual is set, formulas are only recomputed when an explicit recalculation operation is requested. When Automatic is set, any action which causes the contents of a cell to change automatically triggers recalculation of any formulas depending on the changed cell.
14.4.3 Method of Recalculation

Recalculation is the process whereby Objective Grid re-computes the mathematics of the worksheet after the user makes changes. Objective Grid lets you designate whether recalculation takes place on demand (manual mode) or after any change is made (automatic mode).

Natural recalculation is a method of calculation that follows a natural hierarchy of the importance of cell dependencies.

Automatic recalculation supports two methods of calculations: Foreground and As Needed.

14.4.3.1 Foreground

When Foreground is set, Objective Grid enforces minimal recalculation, which means that only the cells in the spreadsheet that are potentially affected by an operation that you perform (such as editing a cell, or moving a range of cells) are recalculated. This feature minimizes the work performed during recalculation and thus speeds up your computations. Minimal recalculation is in effect whenever the recalculation is set to Automatic and the method is set to Foreground.

14.4.3.2 As Needed

When As Needed is set, Objective Grid enforces frugal recalculation, which further reduces the number of cells that need to be computed. The performance gain in recalculation can be significant, depending on the size and complexity of the sheet. Frugal recalculation is in effect whenever the recalculation mode in the Recalc Options dialog is set to Automatic and method is set to As Needed.

As Needed does not support Constraint Checking and Iterative Recalculation options. If all of this seems a bit confusing, refer to Figure 126 for clarification.

14.4.4 Iterative Calculations

Normally, a formula in a given cell should not depend on that cell itself, either directly or indirectly. A circular reference is when two or more cells are defined by each other. Such a condition is called a cyclic dependency. When cyclic dependencies exist, the rule for natural order recalculation as described above does not make sense. When you enter a formula that creates a cyclic dependency, the message "Cycle!" is displayed in the cell.

In some cases cyclic dependencies are useful in that they can represent iterative calculations, which Objective Grid supports. Iterative calculation is a method of calculation whereby Objective Grid passes through the worksheet more than once, addressing circular references or performing goal-seeking. Iterative calculation is useful when two or more cells mutually depend on each other such that each time they are recalculated, their values become closer and closer to the desired answer.
When the Iteration Limit is set to a non-zero value and Method is set to Foreground, iterative calculation is enabled. In this mode, Objective Grid will make multiple recalculation passes, still preserving the natural order and minimal recalculation rules described above, until either the iteration limit has been reached or, if constraint checking is enabled, until all constraints are satisfied. The @INIT function helps perform iterative calculations.

A **forward reference** is a cell whose value depends on values calculated later in a spreadsheet.

### 14.4.5 Constraint Checking

**Constraint checking** is a process whereby Objective Grid checks constraint expressions while recalculating the worksheet. If constraint-checking is disabled in the Default Sheet Characteristics dialog box, Objective Grid ignores constraint expressions.

A **constraint expression** is an expression appended to a formula that establishes conditions under which the formula operates or boundaries for valid results of the formula.

Objective Grid formulas may contain **conditional expressions** which establish conditions under which a formula operates or boundaries for valid results of the formula. When calculating the spreadsheet, Objective Grid ignores constraint expressions unless constraint checking is enabled and the recalculation method is set to Foreground for the spreadsheet.

### 14.4.6 Precision

Objective Grid performs all calculations in double-precision. Calculations with logical operators - ! (logical NOT), && (logical AND), || (logical OR), and ?: (conditional) - consider a non-zero value to be True and a zero value to be False. Integer operators - ~ (complement), & (bitwise AND), | (bitwise OR), ^ (bitwise EXCLUSIVE-OR), and % (modulus) convert their operands to 32-bit integers before performing the operation.
14.5 Objective Grid Formulas

Formulas establish and calculate mathematical relationships between elements of the spreadsheet. Whereas numeric entries remain the same until you change them, cells defined by formulas are automatically changed to reflect changes in referenced cells—even where there are complex interdependencies among cells.

Objective Grid formulas can calculate with numbers, text, logical values, cell references, and other formulas. For example, you can easily calculate the sum of a series of cells, the total of values in a column, a minimum or maximum value within a range, the rounded result of another formula, or the absolute value of a cell entry. Formulas can express complex interdependencies among cells, and they can define constraints on the calculation, such as limits on acceptable values or specific conditions under which a calculation should take place.

Once entered in a cell, formulas are hidden behind the scenes, perform their work in the background, and display only the result of their calculation. To view the formula in a cell, simply select the cell.

Objective Grid provides an option that lets you make all formula expression visible (via CGXGridParam::m_nDisplayExpression).

Objective Grid also provides a wide array of functions that perform certain tasks. Functions can be used alone or in conjunction with formulas and other functions. Objective Grid provides many specialized functions in addition to those that are found in typical financial spreadsheets. See Section 14.3.5, “Built-in Functions.”

14.5.1 Formula Syntax

The general form of an Objective Grid formula is:

```
= expression ; constraint expression // comment
```

In the above formula, expression defines the calculations needed to generate the cell’s value, constraint expression places limits on acceptable values or the circumstances under which the calculation should take place, and comment is any text you want to attach to the cell.

The expression part of Objective Grid formulas looks just like an algebraic formula; it contains values and operators that define the relationships between values.

Objective Grid uses the following conventions for formulas:

- A formula must begin with an equal (=) sign. When you begin typing into a cell, Objective Grid automatically assumes that you are typing a formula if you start with one of the following characters:

  0 1 2 3 4 5 6 7 8 9 . - @ =+

- Formulas can have as many as 511 characters. You can type spaces if you wish, but Objective Grid automatically removes them.
14.5.2 Formula Values

Formulas can contain any or all of the following types of values:

- Numbers, such as 123, -123, 12.3.
- Addresses of single cells, such as A1, D5, Z100.
- Addresses of cell ranges such as B12..G29, A1..D5.
- Absolute cell references denoted with dollar signs before the fixed coordinate ($A$1, $A1$, or A$1), which will not be updated when the referencing cell is moved or copied.
- Functions such as @SUM or @RADIANS, with their arguments.
- Text surrounded by double quotation marks, such as "The sum is " or "Total".
- User-defined cell names or cell range names, such as TOTALS or PROJECT1.

14.5.3 Formula Operators

Operators are the characters that establish the relationships between values in a formula, such as +, -, *, and <= .

Objective Grid supports all the arithmetic, Boolean, and logical operators available in the C programming language. **Arithmetic operators** calculate numeric values; **text operators** act on strings of text, and **logical operators** evaluate true/false conditions.

Objective Grid also provides two operators—exponentiation (**) and percent (%)—that are not available in the C language. It does not support the C address operators or the operators that have side effects, such as ++.

Objective Grid formulas can contain the following operators to define relationship between values.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>% 14</td>
<td>Unary percent</td>
</tr>
<tr>
<td>** 13</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>+ 12</td>
<td>Unary plus</td>
</tr>
<tr>
<td>- 12</td>
<td>Unary minus</td>
</tr>
<tr>
<td>~ 12</td>
<td>Bitwise complement (integer)</td>
</tr>
<tr>
<td>! 12</td>
<td>Logical not</td>
</tr>
<tr>
<td>* 11</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/ 11</td>
<td>Division</td>
</tr>
<tr>
<td>% 11</td>
<td>Remainder (integer)</td>
</tr>
<tr>
<td>+ 10</td>
<td>Addition</td>
</tr>
</tbody>
</table>
In formulas with more than one operator, Objective Grid evaluates operators in the order of precedence presented above, with highest precedence first. That is, AND/OR/NOT operators are evaluated after inequality operators in a logical expression, and multiplication/division operations are performed before subtraction/addition operations in an arithmetic expression. Operators at the same precedence level are evaluated from left to right.

The precedence of operators can be overridden by using parentheses to explicitly specify the order of evaluation.

Here are some special notes about Objective Grid operators:

- The operators marked ‘(integer)’ in Table 19 automatically convert their operands to integers.

- The & operator performs double duty: as a bit-wise and if the operands are numbers or as a string concatenation operator joining two strings together if the operands are text.

  Concatenation is linking two strings together. For example, if you concatenate (A1&A2) the string ”John” in cell A1 with the string ”Smith” in cell A2, the result is the value ”John Smith.”

- The % operator also performs double duty: as the percent operator when appended to a number or numeric expression, or as the C-style modulus operator when applied between two integer expressions.
Operators that define equality/inequality relationships (such as == and <) can be used to compare text strings lexically (alphabetically).

In comparing mixed strings lexically, Objective Grid considers string operands to be lower than numeric operands.

The **conditional operator** returns its second operand if its first operand evaluates True (non-zero) and returns its third operand if it evaluates False, (zero).

In formulas with conditional operators, the second and third operands may be any type Objective Grid supports, including ranges. For example, the expression \( \text{@SUM} \left( A1 \ ? \ B1..C20 : C10..D15 \right) \) returns the sum of B1..C20 if A1 evaluates to non-zero; otherwise it returns the sum of C10..D15.

Objective Grid accepts most arithmetic operators used in other spreadsheets like MS Excel, but there are a few differences in syntax and precedence.

### 14.5.4 Referencing Other Cells in Formulas

The real power of Objective Grid lies in its ability to calculate relationships among different cells in the spreadsheet by typing the row/column coordinates, or address, in the formula.

#### 14.5.4.1 To reference a cell by address:

Type the row and column coordinates of the cell in the formula. For example, to reference Row 5 in Column D, type D5.

#### 14.5.4.2 To reference a contiguous group of cells by address:

Type the row and column coordinates of two cells in opposite corners of the block to be referenced, with two periods ( .. ) between the coordinates. For example, to reference the first five columns and the first five rows of the spreadsheet, type A1..E5.

---

This differs from Microsoft Excel. The equivalent Excel syntax would be A1:A5.

### 14.5.5 Cell Referencing in Objective Grid

Objective Grid differentiates between relative, absolute, and indirect (or current cell) references. The latter is unique to Objective Grid.

**Computed cell references** are the result of a function that is itself a cell reference or range reference. See Section 14.6.11.3, “Computed Cell References,” for more information about these.
14.5.5.1 Relative Reference

Relative references are cell or range references that are interpreted relative to the current position of the cell containing the formula. Relative references are updated whenever a cell is copied or moved, to reflect the new position of the cell. By default, Objective Grid considers references to be relative.

Objective Grid tracks the referenced cell by considering its position relative to the formula cell, not by its address. For example, if the formula in cell A1 references cell B2, Objective Grid remembers that the referenced cell is one row down and one column right. If you copy the formula in cell A1 to another location (e.g., D17), the formula will reference the cell one row down and one column right of the new location (e.g., E18).

14.5.5.2 Absolute Reference

Absolute references are references to cells or ranges that remain fixed, no matter where the cell containing the formula is moved or copied. They are preceded with the dollar sign ($) before each coordinate to be fixed, such as $A1, A$1, or $A$1.

Absolute references remain the same, no matter where you move or copy the original formula. For example, if the formula in cell A1 references cell B2, and you copy the formula in cell A1 to another location (e.g. D17), the formula still references cell B2. To specify an absolute cell address, insert a dollar sign ($) before the address coordinate to be fixed, or before both coordinates if both the row and column coordinates are to be fixed. For example: $B$2.

To specify all or part of a cell address to be absolute, insert a dollar sign ($) before the address coordinate to remain fixed. For example:

- $B$5 makes the complete address absolute.
- $B5 makes the column coordinate (B) absolute, the row coordinate (5) relative.
- B$5 makes the column coordinate (B) relative, the row coordinate (5) absolute.

Refer to Section 14.2.6, “Range Specification,” for information about the TextToRange() and RangeToText() methods.

14.5.5.3 Named Ranges

To assign a name to a cell or range of cells, use the SetRangeName() command. To reference a cell or range by name, type the pre-assigned name of the cell or cell block into the formula.

Cell ranges are also relative, so when you move a cell range, references in formulas within that range are updated to reflect their new location. To specify an absolute range reference, insert dollar signs ($) before the coordinates in the formula. For example, to make the range A1..D5 absolute, type the reference as $A$1..$D$5.

To specify part of a cell range to be absolute, insert dollar signs only before the coordinates to remain absolute. For example, $A1..$D5 will fix the column coordinates of cell references but adjust the row coordinates to reflect the new location.

Refer to Section 14.3.6, “Ranges,” for more information.
**14.5.5.4 Indirect (or Current Cell Offset) Reference**

An *indirect reference* is a unique cell referencing technique in Objective Grid that allows you to refer to cells by row and/or column offset values relative to the current cell. This can be used anywhere a cell reference is expected:

- function arguments
- formulas
- constraint expressions

Certain expressions within the context of Objective Grid require a means to express the current cell. Examples include the conditional statistical functions described in [Section 14.7.3, “Conditional Statistical Functions,”](#) and constraint expressions described in [Section 14.5.6, “Constraint Expressions.”](#)

References to cells in the neighborhood of the current cell are made with offset values enclosed in braces ( {} ) following the pound sign # which identifies the current cell. The offsets tell Objective Grid where to look, in relation to the current cell, for the cell being referenced. A negative row offset indicates a row above the current row. A negative column offset indicates a column to the left of the current column. Likewise, positive offset numbers for the row and column indicate a row below and a column to the right of the current cell.

The format is as follows:

`#{column offset, row offset}`

- If you include only one value in the offset, Objective Grid assumes that it is a column offset. For example, the offset reference `#{-1}` tells Objective Grid to look to the column just left of the current cell.
- The offset values may be constants or expressions.

Examples:

- `#{0,-1}` refers to the cell above the current cell.
- `#{-2}` refers to the cell two columns to the left of the current cell.
- `#{1}` refers to the cell to the right of the current cell.
- `#{0,1}` refers to the cell below the current cell.
- `@CSUM(C4..C100, #{-1} == "Joe")` calculates the sum of all the values in the range C4..C100 for which the cell in the column to the left contains the string "Joe".
- `@CCOUNT(C4..C100, #{0,-1})` counts all the cells in the range C4..C100 whose value is greater than the contents of the cell immediately above.
- `@XVALUE("master.xs3", #)` returns the value of the same cell reference in which this function is stored from the sheet indicated.
- `/verb/#-1+2/` adds 2 to the cell value from the cell to the left.
14.5.6 Constraint Expressions

Constraints are limitations or conditions placed on the variables in your spreadsheet. They are expressed as algebraic statements appended to formulas. You can attach a constraint expression to any formula, by typing a semicolon (;) and the constraint conditions after the formula.

Constraint expressions establish conditions under which a formula operates or boundaries for valid results of the formula. Constraint expressions may be simple equality/inequality relationships, or they can be arbitrary formulas. Any valid Objective Grid expression that returns a numeric value is also a valid constraint expression. However, unlike the expression that defines a cell value, a constraint expression can reference the cell in which it resides, using the symbol #.

For example, the formula =A1 + A2 ; #>2 && #<=B5 || #==C7 means "the value of the current cell is the sum of cells A1 and A2, and that value must be either greater than 2 and less than or equal to the value of cell B5, or equal to the value of cell C7".

Constraint expressions are used, for example, in the conditional statistical functions. The benefit of constraint expressions is maximized when combined with current cell reference support (#) as indicated in the above example.

14.5.7 Explicit Dependency

There may be instances where you need to force a recalculation when certain cell values change, when there is no implicit dependency in the formula that would trigger an automatic recalculation. This explicit dependency option is indicated by appending a backslash (\) to the end of the dependent formula. For example, the formula =@SUM(A1..A20)\D50 instructs Objective Grid to recalculate @SUM(A1..A20) whenever the contents of D50 change.

This feature is particularly important when you have a constraint expression containing an offset reference that produces a cell reference outside the cell range referenced in a dependent formula. Under these circumstances, Automatic Recalculation would not necessarily be triggered. In the above example, @CCOUNT(C4..C100, # #{0,-1}) counts all the cells in the range C4..C100 whose value is greater than the contents of the cell immediately above.

In order for C4 to be evaluated, it must be compared to C3 - which is not part of the explicit range, C4..C100. Without indicating an explicit dependency, C4 would never be evaluated properly. So, in this case, we would indicate the dependency as follows:

@CCOUNT(C4..C100, # #{0,-1})\C3..C99

which tells Objective Grid to recalculate whenever any cell in the range C3..C99 changes.

For more information about explicit dependency and computed cell references, see Section 14.6.11.3, “Computed Cell References.”
14.6 Built-in Functions

Objective Grid functions are predefined formulas supplied with the program that perform the work of many formulas or perform special functions that cannot be achieved by formulas, such as manipulating text strings. They offer a shortcut approach to accomplishing the work of long, complex formulas. Mathematical and statistical functions are often used to sum a column of numbers, compute an average, determine a minimum or maximum value, or round the results of a formula. Other functions are used for more specialized purposes such as computing the future value of an investment or the product of multiplying one cell range by another range. Some functions perform calculations that arithmetic operators cannot handle such as text-string manipulations.

An argument is a parameter appended to a function statement, specifying the values that Objective Grid should use in calculating the function. In the syntax statement for a function, the argument list is the list of arguments the function should use for its calculation.

Objective Grid functions fall into the following categories:

- Mathematical function—Performs calculations with numeric values as arguments.

- Statistical function—Performs aggregation and counting operations on a group of values expressed as a list of arguments.

- Conditional Statistical function—Operates much like statistical aggregation functions, except that the last argument is a constraint expression that Objective Grid evaluates for each cell in the argument list. Only cells that meet constraint criteria are included in the calculation.

- String functions—Act on strings of text, rather than on numeric values.

- Logical function—Evaluates conditions on a purely true/false basis, returning the value 0 if the condition is False and the value 1 if the condition is True.

- Digital Logical function—Returns the values 0, 1 or -1 (unknown) based on the value of its arguments. Digital logical functions evaluate the integer portion of a value. Any value not equal to 0 or 1 is considered unknown.

- Financial function—Performs a common financial calculation, such as future value of an annuity at a given interest rate.

- Date and Time function—Returns a value corresponding to the specified date, month, year, hour, minute or second. It can also be the current system time and date.

- Miscellaneous function—Performs a variety of calculations such as returning a reference to a specific cell or range.

- Embedded Tools—Have the ability to return data in a matrix, not just the resident cell.

14.6.1 Mathematical Functions

Mathematical functions perform calculations such as determining absolute value, finding the integer portion of a number, or establishing the value of a constant. Although you could accomplish these tasks with a formula, using a function saves time and trouble.
Objective Grid also provides a full range of trigonometric functions including sine, cosine, tangent, arc sine, hyperbolic sine, hyperbolic arc sine, as well as vector and matrix arithmetic and manipulation.

Mathematical functions perform calculations with numeric values as arguments, returning numeric values.

14.6.2 Statistical Functions

Statistical functions perform aggregation operations such as calculating means, minimums, maximums, and averages.

Objective Grid also provides more sophisticated statistical test functions that perform operations on a group of values expressed as a list of arguments. These include the F-test, t-tests, correlation coefficient, deviations, and all common averages.

Statistical functions return numeric values.

14.6.3 Conditional Statistical Functions

Conditional statistical functions operate much like statistical aggregation functions, except that the last argument is a constraint expression that Objective Grid evaluates for each cell in the argument list. Only cells that meet constraint criteria are included in the calculation. The constraint expression may be any Objective Grid expression that evaluates to a numeric result.

Conditional statistical functions return a numeric value.

14.6.4 String Functions

String functions manipulate and evaluate character strings. For example, string functions can return the length of a string, find the first occurrence of a string in a range, change a string from uppercase to lowercase and vice versa, or replace one string with another.

String functions return strings or numeric values.

14.6.5 Logic Functions

Logic functions return one value if an argument meets certain criteria, another value if it does not. Logic functions are used as an adjunct to conditional statements.

Logic functions return the value 1, 0, or a value.

14.6.6 Digital Logic Functions

Digital logic functions perform digital logic operations such as AND, OR, NOT, etc.
Digital logic functions return the values 0, 1, or -1 (unknown). Any value whose integer portion is not equal to 0 or 1 is considered unknown. Unknown input values may cause unknown output values.

14.6.7 Financial Functions

Financial functions perform common financial calculations, such as calculating the future value of an annuity at a given interest rate, straight-line depreciation, double-declining depreciation, or the payment term for a given investment. The financial functions in Objective Grid cover annuities, cash flows, assets, bonds, and Treasury Bills.

Financial functions are most useful for solving cash flow calculations where you know all but one variable. For example, if you know the present value of an investment, interest rate, and periodic payment, you can use the @FV function to calculate the future value of the investment. If you know the future value and other variables, but need to know the present value, you can use the @PV function.

Many financial functions require specifying a Day Count Basis. A Day Count Basis indicates the way in which the days in a month and the days in a year are to be counted. Most of the financial functions in securities involve four different Day Count Bases: 30/360, actual/actual, actual/360, and actual/365.

30/360 Day Count Basis assumes 30-day months and 360-day years (12 months x 30 days). Objective Grid also follows the "End-of-Month" rule which assumes that a security pays interest on the last day of the month and will always make its interest on the last day of the month. Special rules are followed when calculating the days between two dates on 30/360 Day Count Basis.

For example, let Start_Date = D1/M1/Y1, End_Date = D2/M2/Y2.

- If D1=31, Objective Grid uses 30 for D1.
- If D2=31, Objective Grid uses 31, unless D1=30 or D1=31. In this case, Objective Grid uses 30.
- If D1 is the last day of February (D1=28 or 29 in a leap year), Objective Grid uses 30 for D1.
- If D2 is the last day of February (D2=28 or 29 in a leap year) and D1 is also the last day of February, Objective Grid uses 30 for D2.

The special arguments used by Objective Grid financial functions are defined in Table 20.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>interest rate</td>
<td>The interest rate to be used in the calculations. The rate may be specified as annual, monthly, or quarterly, but it must agree with the increment you use for periods. By default, the interest rate is an annual rate.</td>
</tr>
<tr>
<td>present value</td>
<td>The present value of an investment, representing the amount already received from or committed to an investment.</td>
</tr>
</tbody>
</table>
Functions related to fixed income securities usually require special dates as arguments: issue date, settlement date, first coupon date, last coupon date, maturity date of a security. When specified, the following constraints should be followed:

- issue settlement maturity
- issue first coupon maturity
- issue last coupon maturity

14.6.8 Date and Time Functions

Date and time functions return values corresponding to the specified date, month, year, hour, minute, or second. You can also use date/time functions to enter the current system time and date in a cell. These functions open up many possibilities for managing accounts receivable and calculating test times.

Objective Grid internally stores date and time information using the same convention as other popular spreadsheet programs:

- Dates are represented as an integer equal to the number of days since December 31, 1899, so January 1, 1900 equals 1.
Times are represented as fractions of a day, starting at midnight. For example, 6:00 AM is stored as 0.25 (a quarter of a 24-hour day).

Using this convention, date and time values may be used together. For example, the date/time value 1.25 corresponds to 6:00:00 AM, January 1, 1900.

14.6.9 Miscellaneous Functions

Miscellaneous functions perform a variety of calculations, such as returning a reference to specific cells or ranges or returning the Nth argument from a list of arguments.

14.6.10 Embedded Tools

Embedded tools are a set of functions in Objective Grid that have the ability to return data in a matrix, not just the resident cell. These functions make non-scalar operations such as matrix multiplication and "live" recalculation as easy to use as an ordinary spreadsheet function.

Embedded tools are a powerful feature in Objective Grid. Their power derives in part from their ability to return a set of data, not just a single value. This function makes non-scalar operations such as matrix multiplication and "live" recalculation as easy to use as an ordinary spreadsheet function.

Embedded tools store values in a group of adjacent cells. These adjacent cells are set to constant formulas, with explicit dependencies on their neighboring cells. For example, an embedded tool in cell B2 might produce the formula =1.3459/B2 in cell B3. This formula indicates that the cell currently contains the constant 1.3459 but that its value depends on the contents of cell B2 (the cell containing the embedded tool).

This notion of explicit dependencies is important for recalculation. It guarantees that any cell that references B3 will not be recalculated until after cell B2 is recalculated. This ensures that data generated by the embedded tool is always current.

Embedded tools look like normal functions, and they can be copied, moved and formatted just as any other formula in the spreadsheet. However, you must follow one important guideline: DO NOT combine embedded tools with other embedded tools in a single formula. For example, the formula @INVERT(@MMUL(A1..C4,F1..I3)) is not allowed.

14.6.11 Using Objective Grid Built-in Functions

You enter a function in a cell in the same way you enter a formula or any other entry, with a few additional guidelines.

- Type in the function name. Objective Grid recognizes the string as a function. Function names are abbreviations that indicate what the function does. For instance, ABS computes absolute value, ROUND rounds to the specified number of places, and AVG computes the average of a list of arguments. Function names may be preceded with an '@' sign, but this is not required.
After typing the function name, enter arguments in parentheses. Most functions use one or more arguments to define the task to be performed. For example, the @AVG function averages the value of two or more arguments. The @LENGTH function returns the length of an argument that is a character string.

Use only the arguments required by the function, in the exact order specified in the function syntax. If you enter other arguments or enter them in the wrong order, Objective Grid will misinterpret their meaning or return an error message.

All the function names in this chapter are typed in uppercase letters, but you can enter them in uppercase or lowercase for your entries.

14.6.11.1 Arguments

Arguments specify the values the function should use in its calculations. The number of arguments, their types, and their formats varies from one function to another. Arguments are usually numeric values, cell or range references, or string values. Most functions have at least one argument; a few have none.

The following table shows different types of arguments used in Objective Grid functions.

Table 21 – Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric Value</td>
<td>123</td>
</tr>
<tr>
<td>Address of a cell</td>
<td>A10</td>
</tr>
<tr>
<td>Address of a range</td>
<td>F9..F99</td>
</tr>
<tr>
<td>String Value</td>
<td>“Quarterly Report”</td>
</tr>
</tbody>
</table>

14.6.11.2 Using Operators with Functions

The result of a function depends on the order in which Objective Grid handles the calculations. Please see Section 14.4, “Calculations,” for more information on operators and their precedence.

14.6.11.3 Computed Cell References

Computed cell references are the result of a function that is itself a cell reference or range reference.

Several Objective Grid functions such as @CELLREF and @RANGEREF return a result that is itself a cell reference or range reference. This is a powerful facility, but it must be used with caution because Objective Grid cannot take these indirect references into account when determining the order of recalculation. The same caution applies to constraint expressions used in conditional statistical functions. As a rule, cells that are indirectly referenced by a function are not automatically recalculated. Objective Grid provides a special construct to force a recalculation, referred to as an explicit dependency.
Objective Grid does not recalculate the spreadsheet unless explicit dependencies have been changed, so you may need to force recalculation if you change the value of a cell that is referenced only indirectly through a function.

For example, suppose you want to count the numeric values in the range C3..J100 that fall within the limits specified in cells A1 and A2. The Objective Grid formula to compute this is @CCOUNT(C3..J100,#A1 && #<A2).

This formula will correctly count the numeric values in the range C3..J100. However, if you change the value in A1, Objective Grid will not automatically recalculate the result, because A1 is referenced only indirectly through the constraint expression.

- To force Objective Grid to recalculate the entire spreadsheet you should call the Recalc() command. You should also add Recalculate menu in your application that calls Recalc().

- You can also force Objective Grid to do a partial recalculation with respect to that cell, edit the cell and append a blank and press the [Return] key on the cell containing the @CCOUNT formula.

- You can also use explicit dependencies to circumvent the limitation described above, if you entered the formula below in the form @CCOUNT(C3..J100,#A1 && #<A2)\A1\A2.

  Objective Grid would take into account the dependencies on A1 and A2 and update the spreadsheet just as you expect.

- Another approach is to construct the condition string with an expression that references the cells directly. For example, @CCOUNT(C3..J100, @STRCAT("#",A1,"&_<",A2)).

  In this example, A1 and A2 are directly referenced and thus will properly trigger recalculation.

Explicit Dependency is described in more detail in Section 14.5.7, “Explicit Dependency.”
### 14.7 Quick-Reference Guide to Built-in Functions

#### 14.7.1 Mathematical Functions

The following table lists the mathematical functions that are supported.

Table 22 – Mathematical Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@ABS(X)</code></td>
<td>The absolute value of X.</td>
</tr>
<tr>
<td><code>@ACOS(X)</code></td>
<td>The arc cosine of X.</td>
</tr>
<tr>
<td><code>@ASIN(X)</code></td>
<td>The arc sine of X.</td>
</tr>
<tr>
<td><code>@ATAN(X)</code></td>
<td>The 2-quadrant arc tangent of X.</td>
</tr>
<tr>
<td><code>@ATAN2(X, Y)</code></td>
<td>The 4-quadrant arc tangent of Y/X.</td>
</tr>
<tr>
<td><code>@CEIL(X)</code></td>
<td>The smallest integer greater than or equal to X.</td>
</tr>
<tr>
<td><code>@COS(X)</code></td>
<td>The cosine of X.</td>
</tr>
<tr>
<td><code>@COSH(X)</code></td>
<td>The hyperbolic cosine of X.</td>
</tr>
<tr>
<td><code>@DEGREES(X)</code></td>
<td>Converts the angle expressed in radians to degrees ( ).</td>
</tr>
<tr>
<td><code>@DET(M)</code></td>
<td>The determinant of the matrix range M, which must be a square matrix.</td>
</tr>
<tr>
<td><code>@DOT(R1, R2)</code></td>
<td>The dot product of the vectors R1 and R2.</td>
</tr>
<tr>
<td><code>@EXP(X)</code></td>
<td>e raised to the X power.</td>
</tr>
<tr>
<td><code>@FACT(N)</code></td>
<td>The value of N!.</td>
</tr>
<tr>
<td><code>@FLOOR(X)</code></td>
<td>The largest integer less than or equal to X.</td>
</tr>
<tr>
<td><code>@FRAC(X)</code></td>
<td>The fractional portion of X.</td>
</tr>
<tr>
<td><code>@GAMMA(X)</code></td>
<td>The value of the gamma function evaluated at X.</td>
</tr>
<tr>
<td><code>@GRAND</code></td>
<td>A 12th-degree binomial approximation to a Gaussian random number with zero mean and unit variance.</td>
</tr>
<tr>
<td><code>@INT(X)</code></td>
<td>The integer portion of X.</td>
</tr>
<tr>
<td><code>@LN(X)</code></td>
<td>The natural log (base e) of X.</td>
</tr>
<tr>
<td><code>@LNGAMMA(X)</code></td>
<td>The log base e of the gamma function evaluated at X.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>@LOG(X)</td>
<td>The log base 10 of X.</td>
</tr>
<tr>
<td>@LOG10(X)</td>
<td>The log base 10 of X.</td>
</tr>
<tr>
<td>@LOG2(X)</td>
<td>The log base 2 of X.</td>
</tr>
<tr>
<td>@MOD(X, Y)</td>
<td>The remainder of X/Y.</td>
</tr>
<tr>
<td>@MODULUS(X, Y)</td>
<td>The modulus of X/Y.</td>
</tr>
<tr>
<td>@PI</td>
<td>The value of pi.</td>
</tr>
<tr>
<td>@POLY(X, ...)</td>
<td>The value of an Nth-degree polynomial in X.</td>
</tr>
<tr>
<td>@PRODUCT(X, ...)</td>
<td>The product of all the numeric values in the argument list.</td>
</tr>
<tr>
<td>@RADIANS(X)</td>
<td>Converts the angle expressed in degrees to radians ( ).</td>
</tr>
<tr>
<td>@RAND</td>
<td>A uniform random number on the interval (0,1).</td>
</tr>
<tr>
<td>@ROUND(X, n)</td>
<td>X rounded to n number of decimal places (0 to 15).</td>
</tr>
<tr>
<td>@SIGMOID(X)</td>
<td>The value of the sigmoid function.</td>
</tr>
<tr>
<td>@SIN(X)</td>
<td>The sine of X.</td>
</tr>
<tr>
<td>@SINH(X)</td>
<td>The hyperbolic sine of X.</td>
</tr>
<tr>
<td>@SQRT(X)</td>
<td>The positive square root of X.</td>
</tr>
<tr>
<td>@SUMPRODUCT(R1, R2)</td>
<td>The dot product of the vectors R1 and R2, where R1 and R2 are of equal dimension.</td>
</tr>
<tr>
<td>@TAN(X)</td>
<td>The tangent of X.</td>
</tr>
<tr>
<td>@TANH(X)</td>
<td>The hyperbolic tangent of X.</td>
</tr>
<tr>
<td>@TRANSPOSE(M)</td>
<td>The transpose of matrix M.</td>
</tr>
<tr>
<td>@VECLEN(...)</td>
<td>The square root of the sum of squares of its arguments.</td>
</tr>
</tbody>
</table>
### 14.7.2 Statistical Functions

The following table lists statistical functions that are supported.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>@AVG(...)</td>
<td>The average (arithmetic mean) of its arguments.</td>
</tr>
<tr>
<td>@CORR(R1, R2)</td>
<td>Pearson’s product-moment correlation coefficient for the paired data in ranges R1 and R2.</td>
</tr>
<tr>
<td>@COUNT(...)</td>
<td>A count of its non-blank arguments.</td>
</tr>
<tr>
<td>@F(M, N, F)</td>
<td>The integral of Snedecor’s F-distribution with M and N degrees of freedom from minus infinity to F.</td>
</tr>
<tr>
<td>@ERF(L[, U])</td>
<td>Error function integrated between 0 and L; if U specified, between L and U.</td>
</tr>
<tr>
<td>@ERFC(L)</td>
<td>Complementary error function integrated between L and infinity.</td>
</tr>
<tr>
<td>@FORECAST(...)</td>
<td>Predicted Y values for given X.</td>
</tr>
<tr>
<td>@FTEST(R1, R2)</td>
<td>The significance level ( ) of the two-sided F-test on the variances of the data specified by ranges R1 and R2.</td>
</tr>
<tr>
<td>@GMEAN(...)</td>
<td>The geometric mean of its arguments.</td>
</tr>
<tr>
<td>@HMEAN(...)</td>
<td>The harmonic mean of its arguments.</td>
</tr>
<tr>
<td>@LARGE(R, N)</td>
<td>The Nth largest value in range R.</td>
</tr>
<tr>
<td>@MAX(...)</td>
<td>The maximum of its arguments.</td>
</tr>
<tr>
<td>@MEDIAN(...)</td>
<td>The median (middle value) of the range R1.</td>
</tr>
<tr>
<td>@MIN(...)</td>
<td>The minimum of its arguments.</td>
</tr>
<tr>
<td>@MODE(...)</td>
<td>The mode or most frequently occurring value.</td>
</tr>
<tr>
<td>@MSQ(...)</td>
<td>The mean of the squares of its arguments.</td>
</tr>
<tr>
<td>@PERCENTILE(R, N)</td>
<td>The value from the range R that is at the Nth percentile in R.</td>
</tr>
<tr>
<td>@PERCENRTANK(R, N)</td>
<td>The percentile rank of the number N among the values in range R.</td>
</tr>
<tr>
<td>@PERMUT(S, T)</td>
<td>The number of T objects that can be chosen from the set S, where order is significant.</td>
</tr>
<tr>
<td>@PTTEST(R1, R2)</td>
<td>The significance level ( ) of the two-sided T-test for the paired samples contained in ranges R1 and R2.</td>
</tr>
<tr>
<td>@QUARTILE(R, Q)</td>
<td>The quartile Q of the data in range R.</td>
</tr>
<tr>
<td>@RANK(E, R[, O])</td>
<td>The rank of a numeric argument E in the range R.</td>
</tr>
</tbody>
</table>
### 14.7.3 Conditional Statistical Functions

The following table lists conditional statistical functions that are supported.

Table 24 – Conditional Statistical Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>@CAVG(...)</td>
<td>Conditional average.</td>
</tr>
<tr>
<td>@CCOUN(...)</td>
<td>Conditional count.</td>
</tr>
<tr>
<td>@CMAX(...)</td>
<td>Conditional maximum.</td>
</tr>
<tr>
<td>@CMIN(...)</td>
<td>Conditional minimum.</td>
</tr>
<tr>
<td>@CSTD(...)</td>
<td>Conditional sample standard deviation (N weighting).</td>
</tr>
</tbody>
</table>

### Table 23 – Statistical Functions (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>@SSQ(...)</td>
<td>The sum of squares of its arguments.</td>
</tr>
<tr>
<td>@RMS(...)</td>
<td>The root of the mean of squares of its arguments.</td>
</tr>
<tr>
<td>@SMALL(R, N)</td>
<td>The Nth smallest number in range R.</td>
</tr>
<tr>
<td>@SSE(...)</td>
<td>The sum squared error of its arguments. Its equivalent to @VAR(...) @COUNT(...).</td>
</tr>
<tr>
<td>@STD(...)</td>
<td>The population standard deviation (N weighting) of its arguments.</td>
</tr>
<tr>
<td>@STDS(...)</td>
<td>The sample standard deviation (N-1 weighting) of its arguments.</td>
</tr>
<tr>
<td>@SUM(...)</td>
<td>The sum of its arguments.</td>
</tr>
<tr>
<td>@T(N, T)</td>
<td>The integral of Student’s T-distribution with N degrees of freedom from minus infinity to T.</td>
</tr>
<tr>
<td>@TTEST(R, X)</td>
<td>The significance level of the two-sided single population T-test for the population samples contained in range R.</td>
</tr>
<tr>
<td>@TTEST2EV(R1, R2)</td>
<td>The significance level ( ) of the two-sided dual population T-test for ranges R1 and R2, where the population variances are equal.</td>
</tr>
<tr>
<td>@TTEST2UV(R1, R2)</td>
<td>The significance level ( ) of the two-sided dual population T-test for ranges R1 and R2, where the population variances are not equal.</td>
</tr>
<tr>
<td>@VAR(...)</td>
<td>The sample variance (N weighting) of its arguments.</td>
</tr>
<tr>
<td>@VAR(...)</td>
<td>The sample variance (N-1 weighting) of its arguments.</td>
</tr>
<tr>
<td>@VSUM(...)</td>
<td>The visual sum of its arguments, using precision and rounding of formatted cell values.</td>
</tr>
</tbody>
</table>
14.7.4 String Functions

The following table lists string functions that are supported.

Table 25 – String Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>@CHAR(N)</td>
<td>The character represented by the code N.</td>
</tr>
<tr>
<td>@CLEAN(S)</td>
<td>The string formed by removing all non-printing characters from the string S.</td>
</tr>
<tr>
<td>@CODE(S)</td>
<td>The ASCII code for the first character in string S.</td>
</tr>
<tr>
<td>@EXACT(S1, S2)</td>
<td>Returns true (1) if string S1 exactly matches string S2, otherwise returns 0.</td>
</tr>
<tr>
<td>@FIND(S1, S2, N)</td>
<td>The index of the first occurrence of S1 in S2.</td>
</tr>
<tr>
<td>@HEXTONUM(S)</td>
<td>The numeric value for the hexadecimal interpretation of S.</td>
</tr>
<tr>
<td>@LEFT(S, N)</td>
<td>The string composed of the leftmost N characters of S.</td>
</tr>
<tr>
<td>@LENGTH(S)</td>
<td>The number of characters in S.</td>
</tr>
<tr>
<td>@LOWER(S)</td>
<td>S converted to lower case.</td>
</tr>
<tr>
<td>@MID(S, N1, N2)</td>
<td>The string of length N2 that starts at position N1 in S.</td>
</tr>
<tr>
<td>NUMTOHEX(X)</td>
<td>The hexadecimal representation of the integer portion of X.</td>
</tr>
<tr>
<td>@PROPER(S)</td>
<td>The string S with the first letter of each word capitalized.</td>
</tr>
<tr>
<td>@REGEX(S1, S2)</td>
<td>Returns true (1) if string S1 exactly matches string S2; otherwise returns false (0). Allows &quot;wildcard&quot; comparisons by interpreting S1 as a regular expression.</td>
</tr>
<tr>
<td>@REPEAT(S, N)</td>
<td>The string S repeated N times.</td>
</tr>
<tr>
<td>@REPLACE(S1, N1, N2, S2)</td>
<td>The string formed by replacing the N2 characters starting at position N1 in S1 with string S2.</td>
</tr>
</tbody>
</table>
### 14.7.5 Logic Functions

The following table lists the supported logic functions.

#### Table 26 – Logic Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>@FALSE</td>
<td>The logical value 0.</td>
</tr>
<tr>
<td>@FILEEXISTS(S)</td>
<td>1 if file S can be opened for reading; otherwise 0.</td>
</tr>
<tr>
<td>@IF(X, T, F)</td>
<td>The value of T if X evaluates to 1, or F if X evaluates to 0.</td>
</tr>
<tr>
<td>@ISERROR(X)</td>
<td>Returns 1 if X &quot;contains&quot; an error; otherwise 0.</td>
</tr>
<tr>
<td>@ISNUMBER(X)</td>
<td>1 if X is a numeric value; otherwise 0.</td>
</tr>
<tr>
<td>@ISSTRING(X)</td>
<td>1 if X is a string value; otherwise 0.</td>
</tr>
<tr>
<td>@TRUE</td>
<td>The logical value 1.</td>
</tr>
<tr>
<td>@AND(...)</td>
<td>0 if any arguments are 0; 1 if all arguments are 1; otherwise -1.</td>
</tr>
<tr>
<td>@NAND(...)</td>
<td>0 if all arguments are 1; 1 if any arguments are 0; otherwise -1.</td>
</tr>
<tr>
<td>@NOR(...)</td>
<td>0 if any arguments are 1; 1 if all arguments are 0; otherwise -1.</td>
</tr>
<tr>
<td>@NOT(X)</td>
<td>0 if X=1; 1 if X=0; otherwise -1.</td>
</tr>
</tbody>
</table>
14.7.6 Financial Functions

The following table lists the supported financial functions.

Table 27 – Financial Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@COUPDAYBS(S, M, F[, B])</td>
<td>The number of days between the beginning of the coupon period to the settlement date.</td>
</tr>
<tr>
<td>@ACCRINT(I, F, S, R, P, F[, B])</td>
<td>Accrued interest for a security that pays periodic interest.</td>
</tr>
<tr>
<td>@ACCRINTM(I, S, R, P[, B])</td>
<td>Accrued interest for a security that pays interest at maturity.</td>
</tr>
<tr>
<td>@COUPDAYS(S, M, F[, B])</td>
<td>The number of days in the coupon period that the settlement date is in.</td>
</tr>
<tr>
<td>@COUPDAYSNC(S, M, F[, B])</td>
<td>The number of days between the settlement date and the next coupon date.</td>
</tr>
<tr>
<td>@COUPNCD(S, M, F[, B])</td>
<td>The next coupon date after the settlement date.</td>
</tr>
<tr>
<td>@COUPNUM(S, M, F[, B])</td>
<td>The number of coupon payments between the settlement date and maturity date.</td>
</tr>
<tr>
<td>@COUPPCD(S, M, F[, B])</td>
<td>The previous (most recent) coupon date before the settlement date.</td>
</tr>
<tr>
<td>@TERM(R, FV, PV)</td>
<td>The number of compounding periods for an investment.</td>
</tr>
<tr>
<td>@CUMIPMT(R, NP, PV, S, E, T)</td>
<td>The cumulative interest on a loan between start period S and end period E.</td>
</tr>
<tr>
<td>@CUMPRINC(R, NP, PV, S, E, T)</td>
<td>The cumulative principal paid on a loan between start period S and end period E.</td>
</tr>
<tr>
<td>@DB(C, S, L, P[, M])</td>
<td>Fixed-declining depreciation allowance.</td>
</tr>
<tr>
<td>@DDB(C, S, L, N)</td>
<td>Double-declining depreciation allowance.</td>
</tr>
<tr>
<td>@DISC(S, M, P, R[, B])</td>
<td>The discount rate for a security.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>@DOLLARDE(FD, F)</td>
<td>Converts a dollar amount expressed as a fraction form into a decimal form.</td>
</tr>
<tr>
<td>@DOLLARFR(DD, F)</td>
<td>Converts a dollar amount expressed as a decimal form into a fraction form.</td>
</tr>
<tr>
<td>@DURATION(S, M, R, Y, F[, B])</td>
<td>The Macauley duration of a security assuming $100 face value.</td>
</tr>
<tr>
<td>@EFFECT(NR, NP)</td>
<td>Returns the effective annual interest rate.</td>
</tr>
<tr>
<td>@FV(P, R, N)</td>
<td>Future value of an annuity.</td>
</tr>
<tr>
<td>@FVSCHEDULE(P, S)</td>
<td>The future value of an initial investment after compounding a series of interest rates.</td>
</tr>
<tr>
<td>@INTRATE(S, M, I, R[, B])</td>
<td>The interest rate for a fully invested security.</td>
</tr>
<tr>
<td>@IPMT(R, P, NP, PV, FV[, T])</td>
<td>The interest payment for a specific period for an investment based on periodic, constant payments, and a constant interest rate.</td>
</tr>
<tr>
<td>@IRR(G, F)</td>
<td>The internal rate of return on an investment. (See also @XIRR and @MIRR.)</td>
</tr>
<tr>
<td>@MDURATION(S, M, R, Y, F[, B])</td>
<td>The modified Macauley duration of a security assuming $100 face value.</td>
</tr>
<tr>
<td>@MIRR(CF, FR, RR)</td>
<td>The modified internal rate of return for a series of periodic cash flows.</td>
</tr>
<tr>
<td>@NOMINAL(ER, NP)</td>
<td>The nominal annual interest rate.</td>
</tr>
<tr>
<td>@ODDFPRICE(S, M, I, FC, R, Y, RD, F[, B])</td>
<td>The price per $100 face value of a security with an odd (short or long) first period.</td>
</tr>
<tr>
<td>@ODDFYIELD(S, M, I, FC, R, PR, RD, F[, B])</td>
<td>The yield per of a security with an odd (short or long) first period.</td>
</tr>
<tr>
<td>@PMT(PV, R, N)</td>
<td>The periodic payment for a loan.</td>
</tr>
<tr>
<td>@PPMT(R, P, NP, PV, FV, T)</td>
<td>The payment on the principal for a specific period for an investment based on periodic, constant payments, and a constant interest rate.</td>
</tr>
<tr>
<td>@PRICE(S, M, R, Y, RD, F[, B])</td>
<td>The price per $100 face value of a security that pays periodic interest.</td>
</tr>
<tr>
<td>@PRICEDISC(S, M, D, RD[, B])</td>
<td>The price per $100 face value of a discounted security.</td>
</tr>
<tr>
<td>@PRICEMAT(S, M, I, R, Y[, B])</td>
<td>The price per $100 face value of a security that pays interest at maturity.</td>
</tr>
<tr>
<td>@PV(P, R, N)</td>
<td>The present value of an annuity.</td>
</tr>
</tbody>
</table>
14.7.7 Date and Time Functions

The following table lists the supported date and time functions.

Table 28 – Date and Time Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@DATE(Y, M, D)</td>
<td>The date value for year Y, month M, and day D.</td>
</tr>
<tr>
<td>@DATEVALUE(S)</td>
<td>The corresponding date value for a given string S.</td>
</tr>
<tr>
<td>@DAYS360(S, E)</td>
<td>The number of days between two dates, based on a 30/360 day count system.</td>
</tr>
<tr>
<td>@DAY(DT)</td>
<td>The day number in the date/time value DT.</td>
</tr>
<tr>
<td>@EDATE(S, M)</td>
<td>The date/time value representing number of months (M) before or after start date (S).</td>
</tr>
</tbody>
</table>
The following table lists miscellaneous supported functions.

Table 29 – Miscellaneous Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>@CELLREF(N1, N2)</td>
<td>A reference to the cell in column N1 and row N2.</td>
</tr>
<tr>
<td>@CHOOSE(N, ...)</td>
<td>The Nth argument from the list.</td>
</tr>
<tr>
<td>@COL(C)</td>
<td>The column address of the cell referenced by C.</td>
</tr>
<tr>
<td>@COLS(R)</td>
<td>The number of columns in the specified range R.</td>
</tr>
</tbody>
</table>

14.7.8 Miscellaneous Functions

The following table lists miscellaneous supported functions.

Table 28 – Date and Time Functions (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@EOMONTH(S, M)</td>
<td>The date/time value representing the last day of the month M months after S, if M is positive, or M months before if M is negative.</td>
</tr>
<tr>
<td>@HOUR(DT)</td>
<td>The hour value (0-23) of date/time value DT.</td>
</tr>
<tr>
<td>@MINUTE(DT)</td>
<td>The minute value (0-59) of date/time value DT.</td>
</tr>
<tr>
<td>@MONTH(DT)</td>
<td>The number of the month in date/time value DT.</td>
</tr>
<tr>
<td>@NETWORKDAYS(S, E[, H])</td>
<td>The number of whole working days, starting at S and going to E, excluding weekends and holidays.</td>
</tr>
<tr>
<td>@NOW</td>
<td>The date/time value of the current system date and time.</td>
</tr>
<tr>
<td>@SECOND(DT)</td>
<td>The seconds value (0-59) of the date/time value DT.</td>
</tr>
<tr>
<td>@TIME(H, M, S)</td>
<td>The time value for hour H, minute M, and second S.</td>
</tr>
<tr>
<td>@TIMEVALUE(S)</td>
<td>The corresponding time value for a given string value S.</td>
</tr>
<tr>
<td>@TODAY</td>
<td>The date value of the current system date.</td>
</tr>
<tr>
<td>@WEEKDAY(D)</td>
<td>The integer representing the day of the week on which the day D falls. 1 is Sunday, 7 is Saturday.</td>
</tr>
<tr>
<td>@WORKDAY(S, D[, H])</td>
<td>The day that is D working days after S, if D is positive, or before S, if D is negative, excluding weekends and all holidays specified as dates in range H.</td>
</tr>
<tr>
<td>@YEAR(DT)</td>
<td>The year value of date/time value DT.</td>
</tr>
<tr>
<td>@YEARFRAC(S, E[, B])</td>
<td>The portion of the year represented by the number of days between start date (S) and end date (E).</td>
</tr>
</tbody>
</table>
Chapter 14  The Formula Engine 351

Some Objective Grid functions return a result that is a range or cell reference. Objective Grid does not include these indirect references in determining the pattern of recalculation. Plan carefully before using these functions. See Section 14.6.11.3, “Computed Cell References,” for more information.

14.7.9  Embedded Tools

The following table lists supported embedded tools.

Table 30 – Embedded Tools

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@DFT(R)</td>
<td>The Discrete Fourier Transform of the range R.</td>
</tr>
<tr>
<td>@EIGEN(M)</td>
<td>The eigenvalues of the matrix M.</td>
</tr>
<tr>
<td>@FFT(R)</td>
<td>The Discrete Fourier Transform of the range R using a fast Fourier Transform algorithm.</td>
</tr>
</tbody>
</table>

Some Objective Grid functions return a result that is a range or cell reference. Objective Grid does not include these indirect references in determining the pattern of recalculation. Plan carefully before using these functions. See Section 14.6.11.3, “Computed Cell References,” for more information.
Table 30 – Embedded Tools (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@FREQUENCY(R, B)</td>
<td>Returns a frequency distribution for values R with a set of intervals B.</td>
</tr>
<tr>
<td>@INVDFTR</td>
<td>The inverse of the Discrete Fourier Transform of the range R.</td>
</tr>
<tr>
<td>@INVERT(M)</td>
<td>The inverse of matrix M.</td>
</tr>
<tr>
<td>@INVFFT(R)</td>
<td>The inverse of the Discrete Fourier Transform of the range R using a fast Fourier Transform algorithm.</td>
</tr>
<tr>
<td>@LINFIT(X, Y)</td>
<td>The straight line least squares fit. This function is equivalent to @POLYFIT(X, Y, 1).</td>
</tr>
<tr>
<td>@LLS(A, Y)</td>
<td>The linear least squares solution X to the over-determined system of equations AX=Y.</td>
</tr>
<tr>
<td>@MMUL(M1, M2)</td>
<td>The product of multiplying matrix M2 by matrix M1.</td>
</tr>
<tr>
<td>@PLS(X, Y, d)</td>
<td>Analyzes the least squares polynomial model Y=P(X), where P is a polynomial of degree d.</td>
</tr>
<tr>
<td>@POLYCOEF(X, Y, d)</td>
<td>The least squares coefficients for the polynomial fit Y=P(X), where P is a polynomial of degree d.</td>
</tr>
<tr>
<td>@TRANSPOSE(M)</td>
<td>The transpose of matrix M.</td>
</tr>
<tr>
<td>@TREND(NX, KX, KY)</td>
<td>The y values for new x values given existing x and y values.</td>
</tr>
</tbody>
</table>

Embedded tools should not be contained within other functions or arithmetic operations in a single formula. For example, the formula @INVERT(@MMUL(A1..C4,F1..I3)) is not allowed. You may, however, copy, move and format embedded tools just as any other function.
14.8 Error Messages

Objective Grid checks for a variety of errors. Depending on the error type, the most recent error message is displayed either inside the affected cell(s), on the Message Line, or inside the Objective Grid Message dialog box.

14.8.1 Types of Errors

14.8.1.1 Errors in Functions

Errors that occur inside functions are reported along with the name of the function in which the error occurred.

14.8.1.2 Formula Syntax Errors

These errors occur only when you are typing in a formula. When you finish entering the formula, Objective Grid will attempt to read the formula and convert it to an internal representation. If it is unable to do so, it continues to display the erroneous formula, switches into “edit mode”, places the text cursor at the beginning of the text that it had difficulty parsing, and displays the error message. The problem must be corrected before Objective Grid can continue.

14.8.1.3 Formula Evaluation Errors

Formula evaluation error occurs when Objective Grid reads in a formula and converts it into its internal formula representation, but is not able to evaluate the formula and produce a correct numeric or string formula. In some cases, the formula has been entered incorrectly, for example, an operand or parenthesis is missing. In other cases, an error has occurred as a result of computation that cannot be handled properly by the computer’s floating point hardware, or there is an error condition in a cell or range that is referenced in the context of this formula. Errors can also occur in the evaluation of Objective Grid built-in functions.

14.8.2 Error Messages Reference

Use this handy reference table to look up the meaning of error messages.

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument must be an integer</td>
<td>@FACT has been passed a non-integer argument.</td>
</tr>
<tr>
<td>argument not a cell or range</td>
<td>Function has been passed an argument that is neither a cell nor a range.</td>
</tr>
<tr>
<td>Error Message</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>argument out of range</td>
<td>An argument to a function is not within the correct range for the function and its other arguments.</td>
</tr>
<tr>
<td>arguments must be numeric</td>
<td>The function requires numeric arguments, which may be literal numbers, formulas which return numeric values, or references to cells containing numeric values.</td>
</tr>
<tr>
<td>arguments must be positive</td>
<td>The arguments in this function must be all positive values.</td>
</tr>
<tr>
<td>can not parse condition string</td>
<td>Objective Grid has encountered a malformed conditional expression.</td>
</tr>
<tr>
<td>cannot find interpolation</td>
<td>@INTERP2D or @INTERP3D is unsuccessful in finding interpolated values.</td>
</tr>
<tr>
<td>cash flow series must be a range</td>
<td>@NPV and @MIRR require that their cash flow series must be a range, which must represent a single column or row.</td>
</tr>
<tr>
<td>cash flow series must be single column or row</td>
<td>@NPV and @MIRR require that their cash flow series must be a range, which must represent a single column or row.</td>
</tr>
<tr>
<td>cell operand contains error condition</td>
<td>A cell which is referenced from the cell in which the error occurs contains an error condition.</td>
</tr>
<tr>
<td>cell reference out of range</td>
<td>A cell reference has been made that is outside the range A1..FAN32767</td>
</tr>
<tr>
<td>coefficient matrix has linearly dependent columns</td>
<td>The existence of a unique solution to a linear least squares (LLS) problem, Ax=b, requires that the columns of A are linearly independent.</td>
</tr>
<tr>
<td>column offset out of range</td>
<td>The third argument to the @VLOOKUP function specifies an offset that is less than 0 or is greater than the width of the range specified in the second argument.</td>
</tr>
<tr>
<td>constraint check not supported with &quot;As Needed&quot;</td>
<td>Constraint checking is not supported when the recalculation is set to &quot;As Needed&quot;.</td>
</tr>
<tr>
<td>contains an error indicator</td>
<td>A cell in one or more of the data ranges for a graph contains an error condition. The error condition must be resolved before Objective Grid can plot the graph.</td>
</tr>
<tr>
<td>could not find real root</td>
<td>@IRR could not find a real root. This suggests that the data given to @IRR is probably wrong.</td>
</tr>
<tr>
<td>count less than zero</td>
<td>User has passed a negative argument to a function that requires a count, for example, with @LEFT, it is impossible to take the -2 leftmost characters of a string.</td>
</tr>
<tr>
<td>data set size must be = 3</td>
<td>@LINFIT and @LINCOEF require a data set of size 3 or larger.</td>
</tr>
<tr>
<td>data set size must be = polynomial degree + 2</td>
<td>@PLS, @POLYFIT, and @POLYCOEF require that the data set size be greater than or equal to the polynomial degree + 2.</td>
</tr>
<tr>
<td>Error Message</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>date series must be single column or row</td>
<td>@XIRR and @XNPV require the argument D (date series) to be a single column or single row.</td>
</tr>
<tr>
<td>decimal places out of range</td>
<td>@STRING only takes a decimal place argument between 0 and 15.</td>
</tr>
<tr>
<td>degrees of freedom must be 0</td>
<td>@F and @T require degrees of freedom greater than zero, as &quot;degrees of freedom&quot; is mathematically undefined for zero or less.</td>
</tr>
<tr>
<td>dimension must be power of 2</td>
<td>@FFT and @INVFFT require matrices whose dimensions are powers of two. The somewhat slower functions @DFT and @INVDFT, respectively, are equivalent functions which do not share this requirement.</td>
</tr>
<tr>
<td>divide by zero</td>
<td>An attempt has been made to divide by zero. Note that Objective Grid considers cells that are empty or contain text strings to have the value zero in the context of a numerical calculation.</td>
</tr>
<tr>
<td>does not accept arguments</td>
<td>Several Objective Grid functions, including @PI, @TRUE, @FALSE, @RAND, and @GRAND, do not accept any arguments.</td>
</tr>
<tr>
<td>domain is -1 &lt; x &lt; 1</td>
<td>@ATANH only takes arguments between -1 and 1, exclusive.</td>
</tr>
<tr>
<td>domain is -1 &lt;= x &lt;= 1</td>
<td>@ACOS and @ASIN only take arguments between -1 and 1, inclusive.</td>
</tr>
<tr>
<td>domain is 0 &lt;= x &lt;= 170</td>
<td>@FACT only takes arguments between 0 and 170, inclusive. (Most platforms)</td>
</tr>
<tr>
<td>domain is x 0</td>
<td>@LN, @LOG2, @LOG, @GAMMA, and @LNGAMMA only take arguments greater than zero.</td>
</tr>
<tr>
<td>domain is x = 1</td>
<td>@ACOSH only takes arguments greater than or equal to 1.</td>
</tr>
<tr>
<td>&quot;End Period&quot; must be &gt;= 1</td>
<td>@CUMIPMT and @CUMPRINC require the argument E (end period) to be greater than or equal to 1.</td>
</tr>
<tr>
<td>&quot;End Period&quot; must be &gt;= &quot;Start Period&quot;</td>
<td>@CUMIPMT, @CUMPRINC and @VDB require the argument E (end period) to be greater than or equal to S (start period).</td>
</tr>
<tr>
<td>ending line with a \</td>
<td>The \ is an escape sequence introducer, which should be followed by another character for interpretation, but the string ended prematurely.</td>
</tr>
<tr>
<td>ending line with a superscript command</td>
<td>When displaying text in the context of graphics, a ^ is a superscript introducer. Like y^2 means &quot;y squared&quot;. This message occurs when a ^ occurs at the end of the string.</td>
</tr>
<tr>
<td>ending line with subscript command</td>
<td>When displaying text in the context of graphics, an _ is a subscript introducer. Like y_2 means &quot;y subscript 2&quot;. This message occurs when an _ occurs at the end of the string.</td>
</tr>
<tr>
<td>error in regular expression</td>
<td>An error occurred while parsing the regular expression used in a search or extract operation, or while executing @REGEX or @MATCH.</td>
</tr>
<tr>
<td>expected the right hand side of a range here</td>
<td>The outer range reference is missing.</td>
</tr>
<tr>
<td>Error Message</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>expected to find [something] here</td>
<td>There was a parsing error. The cursor will be placed in the edit window in edit mode. Read the documentation for the function and correct the error.</td>
</tr>
<tr>
<td>expecting a function</td>
<td>There is something wrong with the formula you have entered on the edit line. The parser was expecting to find a function name at the point indicated by the cursor position.</td>
</tr>
<tr>
<td>expecting an operand</td>
<td>There is something wrong with the formula you have entered on the edit line. The parser was expecting to find an operand at the point indicated by the cursor position.</td>
</tr>
<tr>
<td>expecting an operator</td>
<td>There is something wrong with the formula you have entered on the edit line. The parser was expecting to find an operator at the point indicated by the cursor position.</td>
</tr>
<tr>
<td>extraneous operands</td>
<td>There is something wrong with the formula you have entered on the edit line. The parser finds an extraneous operand at the point indicated by the cursor position.</td>
</tr>
<tr>
<td>F must be &gt;= 0</td>
<td>The third argument to @F must be greater than or equal to 0.</td>
</tr>
<tr>
<td>first argument must be numeric</td>
<td>@NPV and @CHOOSE require that their first argument be numeric.</td>
</tr>
<tr>
<td>floating exception</td>
<td>A floating-point arithmetic hardware exception occurred during the computation of the function or expression. This means that the calculations resulted in a number out of the range that the computer hardware is able to represent.</td>
</tr>
<tr>
<td>found something unexpected here</td>
<td>Objective Grid has found something it doesn’t understand in an expression.</td>
</tr>
<tr>
<td>“Fraction” must be &gt;= 1</td>
<td>@DOLLARDE and @DOLLARFR require the argument F (fraction) to be greater than or equal to 1.</td>
</tr>
<tr>
<td>“Frequency” must be 1, 2 or 4</td>
<td>The argument Frequency (number of coupon payment per year) in financial functions is limited to one of the following choices: 1, 2 or 4</td>
</tr>
<tr>
<td>function not installed</td>
<td>This error occurs when Objective Grid encounters an ”@” followed by a function name which it does not recognize as one of its built-in functions, or one that has been installed by a connection program.</td>
</tr>
<tr>
<td>function stack overflow</td>
<td>This error occurs when functions are nested too deeply. Objective Grid supports nesting of functions up to 50 levels deep.</td>
</tr>
<tr>
<td>hex number greater than 32 bits</td>
<td>Objective Grid cannot convert a hex string to a number if the hex string is longer than 8 characters, which translates to 32 bits in the internal binary representation.</td>
</tr>
<tr>
<td>IEEE Floating Exception (Infinity or NaN)</td>
<td>This error means that the formula caused a computation to occur which could not be calculated properly by the computer’s IEEE standard floating point hardware. Most likely, this means that the computation would produce an intermediate or final result outside the range +/-1.8e308.</td>
</tr>
</tbody>
</table>
### Table 31 – Alphabetized Summary of Error Messages (Continued)

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>illegal cell or range reference</td>
<td>It happens when a copy or move operation results in a cell or range reference that is outside the range A1..FAN32767.</td>
</tr>
<tr>
<td>illegal operand of &quot;operator&quot;</td>
<td>This error occurs when one or both of the operands of the specified &quot;operator&quot; are not valid. Most likely, a range name was used as an operand in an arithmetic expression.</td>
</tr>
<tr>
<td>improper argument type</td>
<td>One or more arguments to the function are incompatible with the type of arguments required by the functions.</td>
</tr>
<tr>
<td>improper coefficient type</td>
<td>In the polynomial evaluation function, (@POLY), one or more of the polynomial coefficients are non-numeric.</td>
</tr>
<tr>
<td>improper dimensions</td>
<td>Several Objective Grid matrix functions and embedded tools have certain requirements on the dimensions of their matrix arguments. Check the reference manual, if you are uncertain about those requirements.</td>
</tr>
<tr>
<td>incompatible matrix dimensions</td>
<td>In matrix multiplication (@MMUL), the number of columns in the first matrix must equal the number of rows in the second matrix.</td>
</tr>
<tr>
<td>incompatible range dimensions</td>
<td>The Objective Grid dot product functions (@DOT) requires vectors of equal size. It will also compute the sum-of-products of any two ranges with equal dimensions.</td>
</tr>
<tr>
<td>index column contains empty cell</td>
<td>The first column in the lookup table referenced by @VLOOKUP must not contain empty cells.</td>
</tr>
<tr>
<td>index out of range</td>
<td>In @FIND, the third argument may not be larger than the length of the second argument. In @MID, the second argument may not be larger than the length of the first argument.</td>
</tr>
<tr>
<td>index row contains empty cell</td>
<td>The first row in the lookup table referenced by @HLOOKUP must not contain empty cells.</td>
</tr>
<tr>
<td>integer parameter out of range</td>
<td>An integer parameter greater than 4294967296 or less than -2147483649 has been entered.</td>
</tr>
<tr>
<td>interest rate should be 0</td>
<td>@EFFECT and @NOMINAL require that argument R (interest rate) to be greater than 0.</td>
</tr>
<tr>
<td>interest schedule must be a single column or row</td>
<td>The argument R (array of interest rates) in @FVSCHEDULE must be a single column or row.</td>
</tr>
<tr>
<td>invalid cell reference</td>
<td>User has tried to access a cell with a row that is negative, zero, or greater than 32767, or with a column that is negative or greater than FAN, or 4095.</td>
</tr>
<tr>
<td>invalid date</td>
<td>Objective Grid could not understand the date format. Date values must be in the range 1-73,050, representing the dates January 1, 1900, to December 31, 2099, respectively. This error can also occur when the year, month, and day values passed to @DATE do not represent an actual date within this range (February 31, 1950, or January 1, 2589, for example).</td>
</tr>
<tr>
<td>invalid day count basis</td>
<td>The day count basis in financial functions should be one of the following choices: 0 (30/360), 1 (actual/actual), 2 (actual/360) or 3 (actual/365)</td>
</tr>
</tbody>
</table>
Table 31 – Alphabetized Summary of Error Messages (Continued)

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>invalid range reference</td>
<td>User has tried to make a range reference that references cells beyond the range of the spreadsheet; that is, a row which is negative, zero, or greater than 32767, or a column which is negative or greater than FAN, or 4095.</td>
</tr>
<tr>
<td>invalid table</td>
<td>The table of reference points in @INTERP2D or @INTERP3D contains non-numeric values or blank cells.</td>
</tr>
<tr>
<td>invalid time</td>
<td>Objective Grid cannot parse a time that the user has provided. Time values are fractional values from 0 to 1, representing fractions of a 24-hour period. When interpreting a number as a date/time value, Objective Grid interprets the integer portion of the number as the date and the fractional portion as the time on that date. A negative value is invalid. Also, the @TIME function must have arguments in the range of 0-23 hours, 0-59 minutes, and 0-59 seconds. Any other values are invalid.</td>
</tr>
<tr>
<td>iterative calculation not supported with &quot;As Needed&quot;</td>
<td>To avoid infinite looping, iterative (self-referential) calculations are not supported when the recalculation method is &quot;As Needed&quot;. To use iterative calculations, the user must choose manual recalculation.</td>
</tr>
<tr>
<td>less than 2 arguments</td>
<td>@POLY requires 2 or more arguments.</td>
</tr>
<tr>
<td>&quot;Life&quot; and &quot;Period&quot; must be integers</td>
<td>@DDB requires that &quot;Life&quot; and &quot;Period&quot;, arguments 3 and 4, respectively, be integers.</td>
</tr>
<tr>
<td>&quot;Life&quot; must be &gt; 0</td>
<td>@SLN and @SYD require that &quot;Life&quot; is greater than 0.</td>
</tr>
<tr>
<td>lookup failed to produce a match</td>
<td>@HLOOKUP or @VLOOKUP failed to produce a match. This should only happen with an alphabetic lookup.</td>
</tr>
<tr>
<td>&quot;Lower limit&quot; must be &gt;=0</td>
<td>The argument L (lower limit) should be greater than or equal to 0 in @ERF and @ERFC.</td>
</tr>
<tr>
<td>magnitude too large</td>
<td>@NUMTOHEX requires an argument between 2147483646 and -2147483647, inclusive.</td>
</tr>
<tr>
<td>matrix is singular</td>
<td>It is mathematically impossible to invert a singular matrix.</td>
</tr>
<tr>
<td>matrix must be square</td>
<td>It is impossible to invert, take the eigenvalue of, or take the determinant of a non-square matrix.</td>
</tr>
<tr>
<td>&quot;Match Type&quot; must be 0 for string match</td>
<td>The argument T (type of match) must be 0 if argument V (value to be matched) is text in @MATCH.</td>
</tr>
<tr>
<td>matrix must be symmetric</td>
<td>@EIGEN requires a symmetric matrix.</td>
</tr>
<tr>
<td>modula divide by zero</td>
<td>Mod 0 is an undefined operation.</td>
</tr>
<tr>
<td>must be -15 to +15 places</td>
<td>@ROUND cannot round to greater than 15 places on either side of the decimal point.</td>
</tr>
<tr>
<td>must have &quot;Cost&quot; = &quot;Salvage&quot; &gt;= 0</td>
<td>@DDB, @SLN, @SYD, @DB, and @VDB require that the &quot;Cost&quot; argument be greater than or equal to the &quot;Salvage&quot; argument, which must be greater than or equal to 0.</td>
</tr>
</tbody>
</table>
### Table 31 – Alphabetized Summary of Error Messages (Continued)

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>must have issue &lt; first coupon &lt; maturity</td>
<td>The values of argument I (issue date), FC (first coupon date) and M (maturity date) must satisfy the following condition: I &lt; FC &lt; M</td>
</tr>
<tr>
<td>must have issue &lt; last coupon &lt; maturity</td>
<td>The values of argument I (issue date), LC (last coupon date) and M (maturity date) must satisfy the following condition: I &lt; LC &lt; M</td>
</tr>
<tr>
<td>must have &quot;Life&quot; = &quot;Period&quot; &gt;= 1</td>
<td>@DDB, @DB, and @VDB all require that the &quot;Life&quot; argument be greater than or equal to the &quot;Period&quot; argument, which must be greater than or equal to 1.</td>
</tr>
<tr>
<td>must have N 0, K 0 and N &lt; K</td>
<td>The arguments N (number of objects to choose from) and K (Number of objects to be chosen) in @PERMUT must follow the following condition: N0, K0 and N&lt;K.</td>
</tr>
<tr>
<td>need at least 2 cash flow values</td>
<td>A single data point does not make a cash flow series; it takes two to trend. Computing the internal rate of return (@IRR) is undefined for only one value.</td>
</tr>
<tr>
<td>operand equal to 0</td>
<td>@HMEAN does not take arguments whose value is 0.</td>
</tr>
<tr>
<td>operand larger than 32 bits</td>
<td>Integers in Objective Grid cannot take more than 32 bits to express. This restricts integers to the range 2147483647 to -2147483648, or 4294967295 to zero, depending on whether the operand is only positive or can be negative.</td>
</tr>
<tr>
<td>no duplicate number found</td>
<td>The @MODE can not find the most frequently occurring number because all numbers appear only once in the argument list.</td>
</tr>
<tr>
<td>no match was found</td>
<td>@MATCH is unsuccessful in finding a match.</td>
</tr>
<tr>
<td>non hex digits in string</td>
<td>@HEXTONUM requires that its argument be a string containing only hex digits, 0-9 and A-F.</td>
</tr>
<tr>
<td>non-numeric operand</td>
<td>An expression of some sort has a non-numeric operand where a numeric operand is required, making the result of the expression undefined.</td>
</tr>
<tr>
<td>non-numeric value in ...</td>
<td>Doing arithmetic on alphabetic entities is undefined.</td>
</tr>
<tr>
<td>not enough arguments to function</td>
<td>User has entered too few arguments to the function.</td>
</tr>
<tr>
<td>&quot;Number&quot; is not in the reference list</td>
<td>The number to be ranked is not in the reference list in @RANK.</td>
</tr>
<tr>
<td>number is too [large</td>
<td>small]</td>
</tr>
<tr>
<td>number of compounding periods should be &gt;=1</td>
<td>@EFFECT and @NOMINAL require that argument C (number of compounding periods) to be greater than or equal to 1.</td>
</tr>
<tr>
<td>one argument must be non-zero</td>
<td>@ATAN2 requires that one of its arguments be non-zero.</td>
</tr>
<tr>
<td>operand contains error condition</td>
<td>Some cell referenced by the operand is in an error condition, or contains a reference to a cell which is in an error condition, etc.</td>
</tr>
</tbody>
</table>
### Table 31 – Alphabetized Summary of Error Messages (Continued)

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>operand less than or equal to 0</td>
<td>@GMEAN does not take arguments that are 0 or negative.</td>
</tr>
<tr>
<td>operand out of range</td>
<td>@CHAR only takes integers between 1 and 255</td>
</tr>
<tr>
<td>operands of &quot;&amp;&quot; must be same type</td>
<td>The &quot;&amp;&quot; operator serves a dual purpose: if its operands are numeric, it performs a bitwise AND operation; if its operands are text strings, it concatenates the two strings. If the operands are neither numeric nor both strings, this error occurs.</td>
</tr>
<tr>
<td>operands of &quot;.&quot; must be cell reference</td>
<td>The .. operator can only join two cell references to create a range. It cannot join integers to make a range of integers, or do anything else.</td>
</tr>
<tr>
<td>&quot;Payment&quot; and &quot;FV&quot; must have the same sign</td>
<td>@TERM requires that Payment and Future Value have the same sign.</td>
</tr>
<tr>
<td>&quot;Payment&quot; must be non-zero</td>
<td>@TERM requires that Payment be non-zero.</td>
</tr>
<tr>
<td>&quot;Period&quot; must be &gt;= 0</td>
<td>@SYD requires that Period be greater than or equal to 0.</td>
</tr>
<tr>
<td>&quot;Period&quot; must be an integer 0</td>
<td>@FV, @PMT, @PV, and @RATE require that Period be an integer greater than 0.</td>
</tr>
<tr>
<td>polynomial degree must be between 1 and 10</td>
<td>@PLS, @POLYFIT, and @POLYCOEF require that the polynomial degree be between 1 and 10.</td>
</tr>
<tr>
<td>pooled sample size less than 3</td>
<td>@TTTEST2EV requires a pooled sample size greater than 2 to be mathematically defined.</td>
</tr>
<tr>
<td>population less than 1</td>
<td>@CVAR, @CSTD, @SSE, @VAR, and @STD require a population greater than or equal to 1.</td>
</tr>
<tr>
<td>&quot;PV&quot; and &quot;FV&quot; must be non-zero</td>
<td>@CTERM and @RATE require that Present and Future Values be non-zero by definition.</td>
</tr>
<tr>
<td>&quot;PV&quot; and &quot;FV&quot; must have the same sign</td>
<td>@CTERM and @RATE require that Present and Future Values have the same sign.</td>
</tr>
<tr>
<td>ranges must be same dimensions</td>
<td>@PTTEST and @CORR require that both their arguments be ranges of equal dimensions, since they work with pairs of values, one value from each range.</td>
</tr>
<tr>
<td>&quot;Rate&quot; must be greater than -1</td>
<td>@CTERM, @FV, @PMT, @PV, @TERM, @NPV, @XNPV, and @XIRR require that their Rate argument be greater than -1.</td>
</tr>
<tr>
<td>&quot;Rate&quot; must be non-zero</td>
<td>@CTERM requires that its Rate argument be non-zero.</td>
</tr>
<tr>
<td>rate found is less than -1</td>
<td>@IRR has found a rate less than -1 after iterating the maximum number of times.</td>
</tr>
<tr>
<td>recursion too deep</td>
<td>This error will occur if Objective Grid encounters &quot;a condition string within a condition string&quot;. For example, it happens with a conditional statistical formula whose condition string calls another conditional statistical function which in turn contains its own condition string.</td>
</tr>
<tr>
<td>Error Message</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>result of expression is a range</td>
<td>Some Objective Grid functions, such as @CELLREF and @RANGEREF, return cell references or range references as a result. Cell and range references cannot be the final result of a formula.</td>
</tr>
<tr>
<td>resultant string too long</td>
<td>A string generated by a formula is too long (greater than 512 characters).</td>
</tr>
<tr>
<td>row offset out of range</td>
<td>The third argument to the @HLOOKUP function specifies an offset that is less than 0 or is greater than the depth of the range specified in the second argument.</td>
</tr>
<tr>
<td>sample missing from pair</td>
<td>The two input ranges to the paired t-test (@PTTEST) and Pearson product-moment correlation (@CORR) functions contain paired values. If a value appears at a given position in the first range, there must also be a value in the corresponding position of the second range.</td>
</tr>
<tr>
<td>sample size less than 2</td>
<td>@CVARS, @CSTDS, @VARS, @STDS, @TTEST, @PTTEST, @TTEST2UV, and @FTEST require a sample size greater than 1.</td>
</tr>
<tr>
<td>searching NULL list</td>
<td>searching list with a NULL function.</td>
</tr>
<tr>
<td>selector out of range</td>
<td>The first argument to @CHOOSE must be 0 or more and be less than or equal to the number of the rest of the arguments - 1.</td>
</tr>
<tr>
<td>settlement date should be &lt; maturity date</td>
<td>Settlement date should be earlier than maturity date in financial functions.</td>
</tr>
<tr>
<td>settlement date should be = issue date</td>
<td>Settlement date should not be earlier than the issue date.</td>
</tr>
<tr>
<td>showing NULL list</td>
<td>showing list with a NULL function</td>
</tr>
<tr>
<td>&quot;Start Period&quot; must be = 1</td>
<td>@CUMIPMT and @CUMPRINC require the argument S (start period) to be greater than or equal to 1.</td>
</tr>
<tr>
<td>starting date should be at beginning of &quot;Dates&quot;</td>
<td>The number in argument D (dates) should not precede the starting date in @XIRR and @XNPV.</td>
</tr>
<tr>
<td>substring longer than string</td>
<td>@FIND cannot find an instance of the pattern string within a shorter target string, since it is impossible to embed a string in a string shorter than itself.</td>
</tr>
<tr>
<td>substring not found</td>
<td>@FIND could not find an instance of the pattern string in the target string.</td>
</tr>
<tr>
<td>token buffer overflow</td>
<td>This error can only occur when a formula is entered that is more complex than Objective Grid can accept. Objective Grid can accept up to 200 operators, numbers, function calls, and text strings in a single formula, which is more than any human can reasonably deal with.</td>
</tr>
<tr>
<td>too few arguments</td>
<td>The function requires more arguments.</td>
</tr>
<tr>
<td>too many arguments to function</td>
<td>User has provided too many arguments to the function. No function can take more than 100 arguments.</td>
</tr>
</tbody>
</table>
Table 31 – Alphabetized Summary of Error Messages (Continued)

<table>
<thead>
<tr>
<th>Error Message</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>too many arguments</td>
<td>@NOT only takes one argument, unlike the rest of the digital logic functions. @ROW and @COL take 1 argument, @ANNOTATE takes 3-5 arguments.</td>
</tr>
<tr>
<td>Treasury Bill should not be outstanding more than 1 year</td>
<td>The period between the settlement date and maturity date of a Treasury bill should not exceed one year.</td>
</tr>
<tr>
<td>unable to parse extract filter</td>
<td>Happens when you are doing an Extract operation and you specify an invalid boolean expression; e.g., #==/5.</td>
</tr>
<tr>
<td>unable to parse search condition</td>
<td>Happens when you are doing a numeric search and you specify an invalid boolean expression; e.g., #==/5.</td>
</tr>
<tr>
<td>undefined symbolic name</td>
<td>This error occurs when Objective Grid encounters a symbolic range or cell reference that has not been defined. To use a symbolic name to refer to a cell or range, you must first define it using the SetRangeName command.</td>
</tr>
<tr>
<td>unexpected question mark</td>
<td>Objective Grid supports C-language compatible condition expressions, which use the operator pair &quot;?&quot; and &quot;:&quot;. If one of these operators appears without the other, an error occurs.</td>
</tr>
<tr>
<td>unresolved name in expression</td>
<td>A name that is not a valid function or named range has been used in the expression.</td>
</tr>
<tr>
<td>&quot;Upper limit&quot; must be &gt;=0</td>
<td>The argument U (Upper limit) should be greater than or equal to 0 in @ERF.</td>
</tr>
<tr>
<td>&quot;values&quot; and &quot;dates&quot; series must have the same dimension</td>
<td>@XIRR and @XNPV require the argument V (cash flow series) and the argument D (date series) to have the same dimension.</td>
</tr>
<tr>
<td>&quot;Values&quot; must have at least one inflow and one outflow</td>
<td>@MIRR requires the value range contains at least one income (positive value) and one payment (negative value)</td>
</tr>
<tr>
<td>wrong number of arguments</td>
<td>The number of arguments passed to the function is incorrect. Check Section 14.7, “Quick-Reference Guide to Built-in Functions,” to determine the correct number of arguments that the function expects.</td>
</tr>
</tbody>
</table>
15.1 Objective Grid DLL Architecture

In this chapter, we will take a look at the distribution architecture of the Objective Grid class library. We will also go into details of how Dynamic Link Libraries (DLLs) in general, and MFC extension DLLs in particular, integrate with MFC applications. Objective Grid is primarily built and linked to as an MFC extension DLL. It is also possible to link to the Objective Grid library statically.

Please refer to the samples located at:

- `<stingray-installdir>\Samples\Grid\Grid\Dll\DllHusk` (Extension DLL)
- `<stingray-installdir>\Samples\Grid\Grid\Dll\OGRegDll` (Regular DLL)

15.1.1 Dynamic Link Libraries

DLLs are a very important part of Windows. DLLs are pieces of program code that are separate from the main program. These segments of code can be loaded by the main program on demand and used in a transparent manner. All of the support required for doing this is built into the Windows operating system.

15.1.2 MFC DLLs

MFC in its present form supports three types of DLLs. These are:

1. Regular DLLs
2. Regular DLLs dynamically linked to MFC
3. Extension DLLs

Understanding how these DLL types function is essential to understanding the DLL architecture that is used by the Objective Grid class library.
15.1.2.1 Regular DLLs

These are the MFC DLL variants that are closest to regular Win32 DLLs. They use MFC internally and statically link to it. These DLLs have a `CWinApp`-derived object and can be called by any Win32 application. The calling application does not have to be an MFC application. Regular DLLs cannot (normally) export MFC objects. They are normally used to export C-style functions. These functions can be called by external programs. For example, consider the following function:

```
extern "C" void _declspec(dllexport) _stdcall Foo()
{
    CString str;
    str = _T("This is a regular dll");
    // perform any additional operations
}
```

This function can be called from any application that links to this DLL.

As can be seen from above, these DLLs can use MFC just like any other MFC application, but cannot export MFC objects.

15.1.2.2 Regular DLLs Linked Dynamically to MFC

These are not very different from the regular DLLs explained above. The only difference is that they link dynamically to MFC. This is a considerable advantage, since it can make the DLL considerably smaller. Yet another unique possibility is available with this DLL. MFC extension DLLs (explained below) cannot be used with non-MFC applications. As a workaround, it is possible to create a regular DLL that links dynamically with MFC. This regular DLL would in turn link to the extension DLL. This lets us use the extension DLL from non-MFC applications. This approach is particularly useful for building COM servers with MFC (Objective Grid CCE uses this approach). With extension DLLs, management of run time classes and class factories is very easy. The functionality is then exported for use by other applications through a regular DLL COM server.

Regular DLLs dynamically linked to MFC also have `CWinApp`-derived application objects.

15.1.2.3 Extension DLLs

These DLLs are the most interesting DLLs from an MFC programmer’s perspective. They can be used to export cool MFC derived functionality in a seamless manner that is otherwise not possible. Let us say that we have a cool `CView`-derived class that we would like to share among programs (and perhaps among programmers). We can simply put this `CView`-derived class in an MFC extension DLL, and it can be shared in seamless manner from any MFC application! To understand how this works, let us take a look at the initialization code in an MFC extension DLL.

MFC extension DLLs do not have a `CWinApp`-derived object. Their `DllMain()` function is very similar to what you would normally expect from a WIN32 DLL.

```c
static AFX_EXTENSION_MODULE TestDll1DLL = { NULL, NULL };
extern "C" int APIENTRY DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
{
    // Remove this if you use lpReserved
```
UNREFERENCED_PARAMETER(lpReserved);

if (dwReason == DLL_PROCESS_ATTACH)
{
    TRACE0("TESTDLL1.DLL Initializing!\n");
    // Extension DLL one-time initialization
    if (!AfxInitExtensionModule(TestDll1DLL, hInstance))
        return 0;
    // comments deleted
    new CDynLinkLibrary(TestDll1DLL);
}
else if (dwReason == DLL_PROCESS_DETACH)
{
    TRACE0("TESTDLL1.DLL Terminating!\n");
    // Terminate the library before destructors are called
    AfxTermExtensionModule(TestDll1DLL);
}
return 1;  // ok

We can see that the DLL makes a call to AfxInitExtensionModule().

    // Extension DLL one-time initialization
    if (!AfxInitExtensionModule(TestDll1DLL, hInstance))
        return 0;

This function just saves data that is used by MFC, the DLL module handle, the run-time class list head, and also the class factory list head. This is the first part of the initialization.

    // save the current HMODULE information for resource loading
    ASSERT(hModule != NULL);
    state.hModule = hModule;
    state.hResource = hModule;

    // save the start of the run-time class list
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    state.pFirstSharedClass = pModuleState->m_classList.GetHead();
    pModuleState->m_classList.m_pHead = pModuleState->m_pClassInit;

The extension DLL acts as part of the application that hosts it. To make this possible, an object of class CDynLinkLibrary is created on the heap. This is the second part of the initialization.

new CDynLinkLibrary(TestDll1DLL);

Creating an instance of CDynLinkLibrary causes the data that is inherent to the DLL to become part of a global linked list that MFC maintains.

The first part of the CDynLinkLibrary saves state information:

    // copy info from AFX_EXTENSION_MODULE struct
    ASSERT(state.hModule != NULL);
    m_hModule = state.hModule;
    m_hResource = state.hResource;
    m_classList.m_pHead = state.pFirstSharedClass;
The second part of the constructor adds this `CDynLinkLibrary` object to the linked list that is maintained by the MFC framework.

```cpp
// insert at the head of the list
//(extensions will go in front of core DLL)
AfxLockGlobals(CRIT_DYNLINKLIST);
m_pModuleState->m_libraryList.AddHead(this);
AfxUnlockGlobals(CRIT_DYNLINKLIST);
```

Adding this `CDynLinkLibrary` to the linked list that is maintained by the framework ensures that the newly loaded MFC extension DLL functions as a seamless part of the application. Whenever MFC locates a resource, run-time class, or a class factory, it looks up this list and basically iterates through it. This ensures that there is no delineation between the calling application and this DLL. To illustrate this, let us look at the code for `CRuntimeClass::Load()`.

```cpp
CRuntimeClass* PASCAL CRuntimeClass::Load(CArchive& ar, UINT* pwSchemaNum)
// loads a run-time class description
{
    WORD nLen;
    char szClassName[64];
    CRuntimeClass* pClass;

    WORD wTemp;
    ar >> wTemp; *pwSchemaNum = wTemp;
    ar >> nLen;

    if (nLen >= _countof(szClassName) ||
        ar.Read(szClassName, nLen*sizeof(char)) != nLen*sizeof(char))
    {
        return NULL;
    }
    szClassName[nLen] = '\0';

    // search app specific classes
    AFX_MODULE_STATE* pModuleState = AfxGetModuleState();
    AfxLockGlobals(CRIT_RUNTIMECLASSLIST);
    for (pClass = pModuleState->m_classList; pClass != NULL;
        pClass = pClass->m_pNextClass)
    {
        if (lstrcmpA(szClassName, pClass->m_lpszClassName) == 0)
        {
            AfxUnlockGlobals(CRIT_RUNTIMECLASSLIST);
            return pClass;
        }
    }
    AfxUnlockGlobals(CRIT_RUNTIMECLASSLIST);
    #ifdef _AFXDLL
    // search classes in shared DLLs
    AfxLockGlobals(CRIT_DYNLINKLIST);
    for (CDynLinkLibrary* pDLL = pModuleState->m_libraryList;
        pDLL != NULL;
        pDLL = pDLL->m_pNextDLL)
```
for (pClass = pDLL->m_classList; pClass != NULL; 
    pClass = pClass->m_pNextClass) 
    { 
        if (lstrcmpA(szClassName, pClass->m_lpszClassName) == 0) 
            { 
                AfxUnlockGlobals(CRIT_DYNLINKLIST); 
                return pClass; 
            } 
    } 
AfxUnlockGlobals(CRIT_DYNLINKLIST); 
#endif 

TRACE1("Warning:Cannot load %hs from archive. Class not defined.\n", 
    szClassName); 
return NULL; // not found

First, the above code loads the run-time class name. It then attempts to look up the run-time class in 
the application. It then looks at the classes in extension DLLs simply iterating through them as 
highlighted above. This explains how MFC extension DLLs work like a part of the calling 
application.

### 15.1.2.4 Exporting Functions from Extension DLLs

To understand the mechanism for importing and exporting symbols (functions and variables) that 
are used in Stingray Studio libraries, refer to Section 2.5, “Extending Stingray Libraries,” of the 
Stingray Studio Getting Started Guide.

### 15.1.3 Module State

Now that we have covered the various important aspects about MFC DLLs that concern an Object-
ive Grid programmer, let us look at yet another related important topic, the concept of module 
state.

In its simplest sense, the module state is state information that is used by the MFC library for 
everything from loading resources to creating ActiveX objects. Each application that links to the 
MFC DLL has a unique module state that separates itself from every other user of that DLL. You 
can think of discrete module state cubes as separate MFC processes. To understand a little better, let 
us look at the definition of AFX_MODULE_STATE (shown with some data omitted for brevity).

// AFX_MODULE_STATE (global data for a module)
class AFX_MODULE_STATE : public CNoTrackObject 
{ 
public: 
#ifdef _AFXDLL 
    AFX_MODULE_STATE(BOOL bDLL, WNDPROC pfAfxWndProc, DWORD dwVersion); 
    AFX_MODULE_STATE(BOOL bDLL, WNDPROC pfAfxWndProc, DWORD dwVersion, BOOL 
    bSystem); 
#else 
    AFX_MODULE_STATE(BOOL bDLL); 
#endif 
}; 

// entry point
It is clear that this data is what defines much of how an extension DLL behaves as part of a chain of DLLs. We can readily see that if a DLL is registered in one module state, it will not be found in another module state list. Even within the same process, we can see that lists of run-time classes, ActiveX class factories etc., are visible only from within the correct module state.

Now that we understand the basics of \texttt{AFX\_MODULE\_STATE}, let us proceed to look at situations where distinct module states come into play and where they don’t. In general, the most important points to remember are:

1. Regular DLLs have their own module state.
2. Extension DLLs do not have their own module state. They take on the state of the calling application or DLL.
3. Applications have their own module state.

We can see from the above that for a MFC extension DLL to function normally, it will have to be in the module state context that it was initialized in. In any other context, calls into the DLL will fail in a cryptic manner, with \texttt{ASSERT\_VALID} triggering or resources failing to load. To gain a better understanding, let us look at a few real life scenarios and attempt to analyze them in terms of what we have seen so far.
15.1.3.1 Application Calls Grid Extension DLL

There is no problem with this approach, since the module state context in which the DLL is initialized is the same as that which is used when calls are made into the DLL. This is the situation that most users will find themselves in. You don’t have to worry much about the module state in this case.

15.1.3.2 Your Regular DLL Links to the Grid Extension DLL. The Application Uses this DLL

In this case we can readily discern, based on what we know, that there are two distinct module states—one that pertains to the application, and one that pertains to the regular DLL. Since this regular DLL is what makes calls into the grid, this is the module that should initialize the grid. So we just initialize the grid in terms of the regular DLLs module state. To do this, we have to make a special call to an initialization function that is exported by the grid extension DLL:

\[ \text{GXInitDll()} \]

All that this function does is to create a `CDynLinkLibrary` on the heap. Why is this not done implicitly, as with all extension DLL usage? The reason is that when the DLL is loaded, the module state that is active will be the state of the calling application. If we initialize in this context, then we will be in for some surprises, as seen in the comments that AppWizard generates when we generate an extension DLL project.

```c
if (dwReason == DLL_PROCESS_ATTACH) {
    TRACE0("TESTDLL1.DLL Initializing!\n");

    // Extension DLL one-time initialization
    if (!AfxInitExtensionModule(TestDll1DLL, hInstance))
        return 0;

    // Insert this DLL into the resource chain
    // NOTE: If this extension DLL is being implicitly linked to by
    // an MFC Regular DLL (such as an ActiveX control)
    // instead of an MFC application, then you will want to
    // remove this line from DllMain and put it in a separate
    // function exported from this extension DLL. The regular DLL
    // that uses this extension DLL should then explicitly call
    // that function to initialize this extension DLL. Otherwise,
    // the CDynLinkLibrary object will not be attached to the
    // regular DLL's resource chain, and serious problems will
    // result.

    new CDynLinkLibrary(TestDll1DLL);
}
```

When we move this one line of code into a separate function (in this case `GXInitDll()`), everything falls in place. When the extension DLL calls this function, the module state that is active is that of the regular DLL. Thus, the extension goes into the correct module state, and everything works as expected.
15.1.3.3 Regular DLL Links to Grid Extension DLL. Application Tries to Use the Grid Directly

This is an unpleasant situation that should be avoided. You can readily see the reason. When the application calls the grid DLL, the module state is incorrect. This also leads to another situation that is not as easily seen. Suppose one of the exported methods that the regular DLL has calls into the grid. Will that cause problems? The answer is yes. The reason is that when calls are made into the regular DLL, the module state is not automatically set to that of the DLL. MFC has no way of knowing which calls you export. So the burden is on you to make sure that all exported calls are protected with the AFX_MANAGE_STATE macro. This macro simply ensures that the correct module state is set for the duration of that call. This switch is automatically made if this is an MFC exported COM interface method or a message handler. MFC knows and takes care of these entry points. For others, it is our responsibility to do so.

Almost all other cases can be explained in terms of the simple rules and situations that we have seen above. This stuff is not so confusing after all!

While we are talking at length about module states, we might as well explain why we use module states in relation to global data in the grid. Previously, all grid state information was maintained in the process’s address space. With version 6.1, we made the switch to module state local data. The required implementation is in gxproc.h for the interested, but in essence this approach lets us integrate our global data with MFC’s least common denominator, the module state.

15.1.3.4 Module State and Cleanup

In versions of Objective Grid prior to 6.1, it was necessary to call the GXTerminate() function during grid cleanup in order to clean up some grid state information that was maintained in the process’s address space. This was true whenever the grid was used in a regular application or any kind of DLL, whether it was a regular DLL, an MFC extension DLL, or an ActiveX control (also known as an OCX).

For Objective Grid versions 6.1 and later, cleanup is handled differently because grid state information is now maintained in the module state local data. If the grid is being used inside of an ActiveX control, you will need to call the GXForceTerminate() function to correctly reset the module state as part of cleaning up your ActiveX control. If you are not using the grid in an ActiveX control, you do not need to do anything at all. The grid will do all necessary cleanup automatically in these cases.

If you are using Objective Grid 6.1 or later, you may safely remove all calls to GXTerminate() that were added when using a previous version of the grid. (Remember, if the grid is inside of an ActiveX control, you must call GXForceTerminate() upon cleanup.) In some cases, leaving the old GXTerminate() calls in will not cause any harm, but in other cases, this will cause problems. To be safe, we recommend that you remove any old GXTerminate() calls from your code.

The method GXTerminate() is unsupported. As a convenience, GXTerminate() has been rewritten in terms of GXForceTerminate(). However, we recommend replacing calls to GXTerminate() with calls to GXForceTerminate().
15.1.4 Manage State

You will need to protect calls across module states with the `AFX_MANAGE_STATE` macro. The `AFX_MANAGE_STATE` macro should be the first call you make in a function, before any memory is assessed or allocated. This a requirement for MFC in general; it has nothing to do with Objective Grid. If you neglect to call `AFX_MANAGE_STATE` first or if you fail to call it altogether, you might experience unpredictable behavior. For example, the control will assert when the mouse cursor is moved across any of the headers because it won’t get the correct resource handle.

If you call mfc42.dll from a regular MFC DLL (instead of from an MFC program or extension DLL), because the global variables in which it stores data aren’t synchronized, you’ll need to insert the following as the first line of all exported functions in your regular DLLs:

```c
AFX_MANAGE_STATE(AfxGetStaticModuleState());
```

The `AFX_MANAGE_STATE` macro won’t have any effect if the code is statically linked to MFC.
15.2 Multithreaded Applications

Objective Grid can be used in a multithreaded application but it cannot be updated from multiple threads. In general, all interaction with the grid has to be through the thread that created it. You can do very easily by using Windows messages and calling `PostMessage()` or `PostThreadMessage()` as appropriate. This will ensure that calls are always made from the context of the correct thread.

15.3 Sharing Extended Functionality Across Classes

To have common source that can be shared between `CWnd`-derived grids and `CView`-derived grids, we recommend using the template approach. This approach is used in the Objective Grid Designer source. An explanation of the template approach follows.

15.3.1 Shared Source

It is often beneficial to have the same code body for a view and a window. This can typically be done by isolating all the code that goes into implementing the control logic into a separate class and then deriving your window or view from this class. The Grid library uses this approach. It derives multiply from `CWnd`- or `CView`-based classes and `CGXGridCore`-based classes.

Figure 127 – Inheritance diagram for the Objective Grid Library

![Inheritance diagram for the Objective Grid Library](image)
Considering that we implement a lot of functionality in our derived class, we can see one possible complication. Our implementation can extend either a window-based class or a view-based class at one time. In many cases, the code that we extend (add) will be common to both views and windows and can be put to good use with both.

To achieve this end, we used the handle view/window classes in earlier versions of the library. These worked on the principle of containment. They would contain a CGXGridCore-derived object and delegate to it. With increasingly better compiler support for templates, we decided that it was time to approach this problem with templates.

With templates, the inheritance hierarchy looks like this:

![Inheritance diagram when using templates for sharing code](image)

Note that we can now extend the template class irrespective of the base class. Once the template class is complete, it can be used either as a window or a view by simply switching the template instantiation parameter. As you can see, this approach is very flexible.

There are some important issues that need to be taken care of before we can implement this approach with the MFC framework. The primary of these issues is with the message map. The default message maps cannot take template arguments. We changed these macros to accommodate message maps.

The code for an elementary template based window is shown below:

```cpp
// Implementation of the template based grid
template<class T>
class TWnd:public T
{
    // Do not use GRID_DECLARE_DYNCREATE
```
/// GRID_DECLARE_DYNCREATE(TWnd)
protected:
// grid overrides:
virtual BOOL GetStyleRowCol(ROWCOL nRow, ROWCOL nCol,
   CGXStyle& style, GXModifyType mt = gxCopy,
   int nType = 0);

// implementation
///{(AFX_MSG(TWnd)
   afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
   afx_msg void OnAppAbout();
   ///</AFX_MSG
GRID_DECLARE_MESSAGE_MAP()
});

/////////////////////////////////////////////////////////////////////
// CTemplateWindow

// special message map macro (ref macro.h)
CCEBEGIN_MESSAGE_MAP(TWnd, T)
///{(AFX_MSG_MAP(TWnd)
   ON_WM_LBUTTONDOWN()
   ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
   ///</AFX_MSG_MAP
END_MESSAGE_MAP()

// you can use ClassWizard to add handlers
template<class T>
void TWnd<T>::OnLButtonDown(UINT nFlags, CPoint point)
{
   T::OnLButtonDown(nFlags, point);
}

template<class T>
void TWnd<T>::OnAppAbout()
{
   AfxMessageBox(_T("Template Window About\n"));
}

template<class T>
BOOL TWnd<T>::GetStyleRowCol(ROWCOL nRow, ROWCOL nCol, CGXStyle& style,
      GXModifyType mt, int nType)
{
   BOOL b = T::GetStyleRowCol(nRow, nCol, style, mt, nType);
   if(nType != -1 && nCol == 1 && nRow !=0)
    {
      style.SetValue(_T("Template grid!"));
    }
   else if(nType != -1 && nCol == 2 && nRow ==1)
    {
      style.SetValue(_T("Common validation code!"));
    }
   return b;
}
You will notice that, apart from the normal template syntax, the only change is with the message map. The syntax for the new message map is very simple; only the first line is changed. The logic is the same. Derived class maps to Base class.

\CCEBEGIN_MESSAGE_MAP(TWnd, T)
    ///(AFX_MSG_MAP(TWnd)

Please take a look at the file \texttt{gxmesmac.h} if you are interested in the actual macros.

Now that you have the template-based common code in a template class you can instantiate your actual derived class without much effort:

\begin{verbatim}
class CTemplateView : public TWnd<CGXGridView>
{
    protected: // create from serialization only
        CTemplateView();
        GRID_DECLARE_DYNCREATE(CTemplateView)
}
\end{verbatim}

Now that the class that you are deriving from is a regular class, there is no need for the usage of any special macros in this derived class. The only exception is with the \texttt{GRID_IMPLEMENT_DYNCREATE} macro.

You will have to use this macro instead:

\begin{verbatim}
\verbatimcode{Format}
\verbatimcode{// Format}
\verbatimcode{// <this class, template base class, ultimate base grid/window class>}
\verbatimcode{CCEIMPLEMENT_DYNCREATE(CTemplateView, TWnd<CGXGridView>, CGXGridView)}
\end{verbatim}

The message maps (etc.) are the same as with the regular grid.

Please refer to the Template sample available from the Rogue Wave Web site, as explained in Section 3.6.1, “Location of Sample Code,” in the \textit{Stingray Studio Getting Started Guide}. 


15.4 Plug-in Components

15.4.1 Motivation

MFC has its limitations when it comes to encapsulating different window functionality into separate objects. When you program with MFC, you often have to implement different window actions based on user events. For example, when the user presses a mouse button, the MFC window is set into a special context. In subsequent mouse messages, you check the context of the window and give graphical feedback to the user based on the mouse movements. Once the user releases the mouse button, you reset the window context and perform the specified user action. Suppose you want to add support for more user actions. The easiest way is to add if-statements for each context in your message handlers. This approach has severe disadvantages: each event handler is responsible for handling a variety of actions that are not related to each other. In short, it ignores encapsulation.

As an experienced C++ developer, you want to get rid of these if-statements and provide individual objects for each user action. The goal of this article is to show you a new approach for encapsulating user actions into separate objects that support MFC message maps. These special objects, called Plug-in components, can be reused among different window and view objects without code duplication. To give you a real world example for using this plug-in approach, this article includes a convenient reusable MFC class called CMSJIntelliMousePlugin that can be attached to any CWnd or CView class. The component provides support for IntelliMouse scrolling, zooming, and panning. No code change is necessary in the component source code to use its functionality with different window and view classes.

15.4.2 Existing Approaches

There are existing approaches that encapsulate user actions into separate objects and do not use if-statements, but these solutions lack support for the MFC message map. Consequently, most MFC developers avoid these approaches.

15.4.2.1 Approach One

The first approach is to add message handlers to the window class and forward each of these messages to the attached component object that is responsible for handling the user actions. The following code snippet demonstrates how to delegate the WM_MOUSEMOVE message to an attached object:

```cpp
void CMyView::OnMouseMove(UINT nFlags, CPoint point)
{
    // forward this event to an attached object
    m_pObject->OnMouseMove(nFlags, point);

    CView::OnMouseMove(nFlags, point);
}
```
The disadvantage of this approach is obvious. There is a tight coupling between the window class and the user action component. Whenever you need to process a new message in the component, you need to add a message handler in the parent window class and forward it to the component. You might try to solve the problem by providing pre-defined message handlers for each window message, but this approach has the disadvantage that it results in a large amount of messages and relatively few of them will be used.

15.4.2.2 Approach Two

The second approach is to override the `WindowProc()` method that is the entry point for all window messages sent to a window. In the overridden method, you can forward each window message to the attached user action component object. In the attached component, you implement a switch statement that provides handlers for the window messages you want to handle. The following code shows what a typical event handler looks like:

```cpp
void CUserActionComponent::HandleMessage(UINT nMessage, WPARAM wParam, LPARAM lParam)
{
    switch (nMessage)
    {
    case WM_MOUSEMOVE:
        OnMouseMove(wParam, CPoint(LOWORD(lParam), HIWORD(lParam));
        break;
    }
}
```

This approach allows you to add messages in the user action component without changing the parent window class, however, it is a step backwards, akin to early C-like Windows SDK development. This approach requires you to perform the tedious task of decoding the `WPARAM` and `LPARAM` parameters into useful information. After decoding a few of these parameters, you will wish you could still use the ClassWizard to add new messages.

15.4.2.3 Summary of Existing Approaches

Both approaches are insufficient because they lack support for MFC message maps in the user action components. Given these alternatives, the if-statement approach is the most attractive alternative even if it entails copying and pasting code from one window class to another to provide the same functionality for different views or windows.

Using plug-in components lets you override `WindowProc()` and forward all windows messages to a user action component that fully supports MFC message maps. ClassWizard lets you add message handlers, as with any other MFC window class.

15.4.3 The Plug-in Approach

Before demonstrating how to implement the plug-in approach, let us describe the requirements for this approach and show how to encapsulate the solution in one common base class for plug-in components.
Here are the requirements for the plug-in approach:

1. Determine one point of entry for searching the MFC message map and dispatching any window messages to the correct message handler in a derived class.
2. Ensure that source code for user actions in existing window classes can be reused without making major changes.
3. Avoid redundant calls to the default window procedure. Only the parent window object should call this method.

Here is a more detailed discussion for each of these requirements and its solutions:

### 15.4.3.1 Message Dispatching

MFC message dispatching is implemented by `CWnd`'s `OnWndMsg()` member function. `OnWndMsg()` searches the window’s message map and calls the correct message handler in a derived class. One interesting feature of `OnWndMsg()` is that it correctly dispatches messages whether or not a valid windows handle is attached to the `CWnd` object. `OnWndMsg()` is completely independent of the `CWnd`'s `m_hWnd` attribute. It works correctly even if you never called `CWnd::Create()`.

Armed with this knowledge, we could derive the plug-in component base class from `CWnd` that does not need to be attached to a windows handle and provides an entry point for each window message. The entry point for window messages is the plug-in component’s `HandleMessage()` method.

The following code shows how `HandleMessage()` is implemented. The method calls the protected `CWnd::OnWndMsg()` member, which then searches the message map and calls the correct message handler. The meaning of the attributes `m_bExitMessage` and `m_bSkipOtherPlugins` is discussed later in this article.

```cpp
BOOL CMSJPluginComponent::HandleMessage(UINT message, WPARAM wParam, LPARAM lParam, LRESULT* pResult)
{
    m_bSkipOtherPlugins = FALSE;
    m_bExitMessage = FALSE;

    return CWnd::OnWndMsg(message, wParam, lParam, pResult);
}
```

The next code snippet shows how messages are forwarded from the parent window class to the plug-in component. `m_pPlugin` is a pointer to a plug-in component object.

```cpp
LRESULT CMyView::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
    if (m_pPlugin)
    {
        LRESULT lResult;
        m_pPlugin->HandleMessage(message, wParam, lParam, &lResult);
        if (m_pPlugin->m_bExitMessage)
            return lResult;
    }
    return CScrollView::WindowProc(message, wParam, lParam);
}
```
15.4.3.2 Reuse of Existing Code

*CWnd* is a thin wrapper class for a window handle and provides many member functions that rely on the *m_hWnd* attribute. For example, *CWnd::Invalidate()* is a wrapper to the equivalent Windows SDK method and passes *m_hWnd* as the window handle. The member function is declared as an inline method in *afxwin.inl* using the following code:

```cpp
AFXINLINE void CWnd::Invalidate(BOOL bErase)
{
    ASSERT(::IsWindow(m_hWnd));
    ::InvalidateRect(m_hWnd, NULL, bErase);
}
```

Many other *CWnd* member functions are implemented in exactly the same way. If you port existing code to a plug-in component and call a *CWnd* member function, your application asserts because *m_hWnd* is not a valid window handle. To solve this problem, we need to provide a valid windows handle for the plug-in component’s *m_hWnd* attribute.

Consider the following two issues:

1. *CWnd::OnWndMsg* disregards the value of the *m_hWnd* attribute, so we can assign any value to it.

2. A plug-in component is not a real window object. The plug-in component should operate directly on the parent window object. It receives the same messages that the parent window object receives and any window operations that are executed in the plug-in component need to affect the parent window.

Assigning the parent’s window handle to the plug-in component’s *m_hWnd* attribute is the ideal solution. Using the parent’s window handle lets you port existing code to a plug-in component without changing any existing calls to *CWnd* member functions. All *CWnd* member functions now operate directly on the parent window.

If you are an experienced MFC developer, you may question the legality of assigning the same windows handle to different *CWnd* objects. In the case of *CWnd::Attach()* , you cannot assign the same windows handle to different *CWnd* objects. If you attempt to do this, MFC will assert. Internally, MFC only allows one window object for each window handle. The window handles and *CWnd* objects are maintained in the window handle map. However, the *Plug-In Approach* does not require us to call *CWnd::Attach()* . Instead, we only assign the windows handle to *m_hWnd*, which is safe. However, you should be aware that whenever you call *CWnd::FromHandle(m_hWnd)*, MFC returns a pointer to the parent *CWnd* object, because this is the window that is registered in the MFC window handle map.

15.4.3.3 Default Window Procedure

As mentioned earlier, another requirement is to avoid redundant calls to the default window procedure of the parent window. The solution to this problem is to override the virtual *DefWindowProc()* method for the plug-in component class and return immediately, as shown in the following code snippet below. Then, only the parent window is calling the default window procedure.
LRESULT CMSJPluginComponent::DefWindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
    // do nothing - this makes sure that calls to Default() 
    // will have no effect (and thus make sure that the same 
    // message is not processed twice).
    return 0;
}

15.4.4 The CGXPluginComponent Class

The CGXPluginComponent class is the base class for plug-in components.

Here is a short overview on member functions and attributes.

- **Plugin()** — Call this method to attach the component to a window object. The method assigns the window’s handle to the plug-in component’s m_hWnd attribute.

- **m_bExitMessage** — If you set m_bExitMessage equal to TRUE, the window procedure should return after the plug-in component has processed the message. See the source code for WindowProc() earlier in this chapter to see how to process this attribute in the override of the WindowProc() method in your parent window class.

- **m_bSkipOtherPlugins** — Use this attribute to coordinate several plug-ins. If you want to attach several plug-ins to a window object, check this attribute in the WindowProc() method of the parent window class.

15.4.5 Implementing a Sample Plug-in Component

To show you how easy it is to use the plug-in approach, this section documents the development steps for implementing a sample auto-scroll component. The auto-scroll component checks if the user presses the left mouse button. In response to this event, a timer is started and WM_VSCROLL messages are sent to the parent window. When the user moves the mouse up or down, the parent window scrolls into the given direction. Once the user releases the mouse button, the timer is killed and the auto-scroll operation ends. Other events such as the WM_CANCELMODE message or when the user presses the ESC key stop the operation. The component can easily be reused and attached to any view or window class without changing its source code.

The major steps for implementing the sample component are:

1. Create a small MFC application with AppWizard using CView as the main window’s base class. For instance, create a class called CMyView.

2. Derive a class called CAutoScrollPlugin from the CGXPluginComponent class, where you add the functionality that should be encapsulated in the plug-in component.

3. In the CMyView class, override WindowProc() and then call the OnWndMsg() method of the plug-in component. To do this, call CGXPluginComponent::PlugIn(this) in your CMyView::OnInitUpdate() override.
Here is a more detailed explanation:

1. When you create the project using the MFC AppWizard, please derive the view class from \texttt{CScrollView}. We recommend you name the view class \texttt{CMyView}. After you generate the project, enlarge the scroll range specified in \texttt{OnInitialUpdate()}. For example:

   \begin{verbatim}
   CSize sizeTotal;
   sizeTotal.cx = sizeTotal.cy = 15000;
   SetScrollSizes(MM_TEXT, sizeTotal);
   \end{verbatim}

2. Next, create the \texttt{CAutoScrollPlugin} class. Use ClassWizard to derive a class from a generic \texttt{CWnd} and name it \texttt{CAutoScrollPlugin}. After you generate the class, you can derive it from \texttt{CGXPluginComponent}. To do this, edit the header and implementation file and replace all occurrences of \texttt{CWnd} with \texttt{CGXPluginComponent}. If you remove the existing ClassWizard (.clw) file from the project directory and press <CTRL>+W, the ClassWizard file is regenerated. You can then add message handlers to the \texttt{CAutoScrollPlugin} class with ClassWizard.

The following listing shows the final implementation of the \texttt{CAutoScrollPlugin} component.

\begin{verbatim}
// AutoPlug.cpp : implementation file
#
#include "stdafx.h"
#include "autoscrl.h"
#include "AutoPlug.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

const int nScrollTimer = 991;

/////////////////////////////////////////////////////////////////////
CAutoScrollPlugin
CAutoScrollPlugin::CAutoScrollPlugin()
{
    m_bIsAutoScrolling = FALSE;
    m_nTimer = 0;
}

CAutoScrollPlugin::~CAutoScrollPlugin()
{
}

BEGIN_MESSAGE_MAP(CAutoScrollPlugin, CGXPluginComponent)
   //{{AFX_MSG_MAP(CAutoScrollPlugin)
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONUP()
    ON_WM_TIMER()
    ON_WM_KEYDOWN()
    ON_WM_CANCELMODE()
   //}}AFX_MSG_MAP
END_MESSAGE_MAP()
\end{verbatim}
// CAutoScrollPlugin message handlers

void CAutoScrollPlugin::OnLButtonDown(UINT nFlags, CPoint point)
{
    m_ptMouseDown = point;
    ClientToScreen(&m_ptMouseDown);

    m_bIsAutoScrolling = TRUE;
    SetCapture();

    m_nTimer = SetTimer(nScrollTimer, 10, NULL);

    m_bExitMessage = TRUE;
}

void CAutoScrollPlugin::OnLButtonUp(UINT nFlags, CPoint point)
{
    if (m_bIsAutoScrolling)
        AbortScrolling();

    CGXPluginComponent::OnLButtonUp(nFlags, point);
}

void CAutoScrollPlugin::OnTimer(UINT nIDEvent)
{
    CPoint pt;
    GetCursorPos(&pt);

    UINT nSBCode = SB_LINEUP;
    if (pt.y > m_ptMouseDown.y)
        nSBCode = SB_LINEDOWN;

    SendMessage(WM_VSCROLL, MAKEWPARAM(nSBCode, 0), NULL);

    CGXPluginComponent::OnTimer(nIDEvent);
}

void CAutoScrollPlugin::AbortScrolling()
{
    if (m_bIsAutoScrolling)
    {
        m_bIsAutoScrolling = FALSE;
        ReleaseCapture();
        KillTimer(m_nTimer);
    }
}

void CAutoScrollPlugin::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    if (nChar == VK_ESCAPE)
        AbortScrolling();

    CGXPluginComponent::OnKeyDown(nChar, nRepCnt, nFlags);
}
void CAutoScrollPlugin::OnCancelMode()
{
    AbortScrolling();
}

3. The next step is to add a pointer to the plug-in object in your view class. To do this, add this code to the class declaration:

    class CMyView: public CScrollView
    {
        ...
        CGXPluginComponent* m_pPlugin;
    }

    In myview.cpp, instantiate the auto-scroll component and call its Plugin() method in the OnInitialUpdate() routine:

    m_pPlugin = new CAutoScrollPlugin;
    m_pPlugin->PlugIn(this);

    Don’t forget to include the header file for the CAutoScrollPlugin class in myview.cpp!

4. Finally, override WindowProc() and call the HandleMessage() method of the plug-in component.

    The following listing shows the implementation of the CMyView class:

    // myView.cpp : implementation of the CMyView class
    //
    #include "stdafx.h"
    #include "autoscrl.h"
    #include "MyDoc.h"
    #include "myView.h"
    #include "autoplug.h"

    #ifdef _DEBUG
    #define new DEBUG_NEW
    #undef THIS_FILE
    static char THIS_FILE[] = __FILE__;
    #endif

    ///////////////////////////////////////////////////////////////////////////// CMyView

    GRID_IMPLEMENT_DYNCREATE(CMyView, CScrollView)

    BEGIN_MESSAGE_MAP(CMyView, CScrollView)
       //{{AFX_MSG_MAP(CMyView)
        //}
        AFX_msg_map(CMyView)
    END_MESSAGE_MAP()

    ///////////////////////////////////////////////////////////////////////////// CMyView

    construction/destruction
15.4.6 Using the IntelliMouse Plug-in Component

A real world example for using this plug-in approach is the convenient and reusable CGX IntelliMousePlugin component that can be attached to any CWnd or CView class. The component provides support for IntelliMouse scrolling, zooming and panning. No code change is necessary in the component source code to use its functionality with different window and view classes.

Here is a short overview of the functionality implemented with the component. The implementation is very similar to MS Excel and Internet Explorer 4.0. The following features are provided:

- **Scroll** by rolling the mouse wheel.
- **Scroll horizontally** by pressing and holding SHIFT while rolling the mouse wheel.
- **Zoom in and out** by pressing and holding CTRL while rolling the mouse wheel.
Auto-Scroll by pressing the mouse wheel button down while dragging the mouse up, down, left, or right.

Click-Lock for the mouse wheel button: Just press and hold down the mouse wheel button for a moment to lock your click. With Click-Lock, you can scroll through the grid easily by simply dragging the mouse. Its functionality is identical to Auto-Scroll, except you don’t need to hold the mouse wheel button. Press (click) the mouse wheel button again to release Click-Lock.

The CGXIntelliMousePlugin class integrates into any advanced view that may itself process many window messages.

15.4.6.1 Using the IntelliMouse Component with any View or Window

Check out the ogmouse sample (samples/grid/plugin/ogmouse) to see how easy it is to add IntelliMouse support to any existing view. All you have to do is follow these steps from the previous section:

1. Add a pointer to the plug-in object in your view class. Allocate a CGXIntelliMousePlugin object and call CGXPluginComponent::Plugin() at initialization time.

15.4.6.2 Using the IntelliMouse Component with CGXGridCore

With CGXGridCore, using IntelliMouse support is even easier. CGXGridCore already provides internally the necessary dispatching of messages in the WindowProc() method. All you need to do is register the IntelliMouse component. To do this, simply call EnableIntelliMouse(); at initialization time of your grid (for example in OnInitialUpdate()).

EnableIntelliMouse() calls CGXGridCore::AddPlugin(). CGXGridCore::AddPlugin() registers the plug-in component within the grid. This registration is necessary to let the grid know that it should dispatch WindowProc() events to this component. When the grid goes out of scope, the plug-in object will be automatically destroyed.

Check out the GridApp sample for using IntelliMouse within a grid.
15.5 Extending the Undo/Redo Mechanism

The Objective Grid Undo/Redo mechanism is fully extensible. You can add custom support for Undo/Redo using self-written commands.

The *CGXCommand* class is an abstract base class for storing Undo information for commands. If you want to support Undo/Redo for your commands, you should derive a class from *CGXCommand* and store all information necessary for undoing the operation into the command.

Objective Grid maintains *CGXCommand* objects in an Undo and Redo list. When the user wants to undo the last operation, Objective Grid removes the first *CGXCommand* object in the list and calls the object's *Execute()* method. In your *CGXCommand*-derived class, you should override the *Execute()* method and embed the necessary steps to undo the command.

Basic steps for adding Undo/Redo support for a specific command are:

1. Derive a class from *CGXCommand*.
2. Insert the *GRID_DECLARE_DYNAMIC*/*GRID_IMPLEMENT_DYNAMIC* and the *GRID_DECLARE_COMMAND*/*GRID_IMPLEMENT_COMMAND* macros (see the example).
3. Provide a constructor.
4. Override the *Execute()* method.
5. In your command method, create the *CGXCommand*-derived object and add it to the Undo/Redo list. This is done by calling *AddCommand()*.

Here is a complete example:

// header file:

```cpp
class CGXSetFrozenRowsCmd: public CGXCommand
{
  GRID_DECLARE_DYNAMIC(CGXSetFrozenRowsCmd);
  GRID_DECLARE_COMMAND(CGXSetFrozenRowsCmd);

public:
  // Construction
  CGXSetFrozenRowsCmd(ROWCOL nFrozenRows, ROWCOL nHeaderRows);

  // Operation
  virtual BOOL Execute(CGXGridCore* pGrid, GXCmdType ctType);

  // Data
  ROWCOL  m_nFrozenRows;
  ROWCOL  m_nHeaderRows;
};
```

// implementation file (.cpp)

```cpp
GRID_IMPLEMENT_DYNAMIC(CGXSetFrozenColsCmd, CGXCommand);
GRID_IMPLEMENT_COMMAND(CGXSetFrozenColsCmd, GX_IDM_SETFROZENCOLS);
 // GX_IDM_SETFROZENCOLS is a string resource id with
 // the description of the command.

CGXSetFrozenRowsCmd::CGXSetFrozenRowsCmd(ROWCOL nFrozenRows,
                                      ROWCOL nHeaderRows)
```
BOOL CGXSetFrozenRowsCmd::Execute(CGXGridCore* pGrid,
                                      GXCmdType ctCmd)
{
    return pGrid->SetFrozenRows(m_nFrozenRows, m_nHeaderRows,
                                 GX_UPDATENOW, ctCmd);
}

BOOL CGXGridCore::SetFrozenRows(ROWCOL nFrozenRows,
                                 ROWCOL nHeaderRows, UINT flags, GXCmdType ctCmd)
{
    ASSERT(nHeaderRows <= nFrozenRows);
    // ASSERTION-> Rows with headers must be frozen ->END

    ROWCOL nOldFrozenRows = GetFrozenRows();
    ROWCOL nOldHeaderRows = GetHeaderRows();
    if (StoreFrozenRows(nFrozenRows, nHeaderRows))
    {
        UpdateFrozenRows(nOldFrozenRows, nOldHeaderRows,
                          flags, TRUE);

        if (ctCmd != gxRollback && m_pParam->m_bUndoEnabled)
            AddCommand(new CGXSetFrozenRowsCmd(nOldFrozenRows,
                                                 nOldHeaderRows), ctCmd);

        return TRUE;
    }

    return FALSE;
}
15.6 Print, Print Preview

The shared library Print.dll encapsulates the print and print preview functionality and can be linked to any type of application in which CGXGridWnd-based classes are used. Source code for this regular DLL is located in Samples\Grid\General\PrintPreview\Print.

15.6.1 Implementation

MFC provides print support for CView-derived classes only. To support CWnd-derived classes, use the Print.dll, which encapsulates print and print preview functionality. This DLL is an encapsulation of an SDI application containing a CGXGridView object that takes a CGXGridParam object as input.

Use these functions to access the print and preview capabilities in Print.dll:

```cpp
BOOL Preview(HWND hWndParent, CGXGridParam* Param, UINT WM);
BOOL PrintWnd(HWND hWndParent, CGXGridParam* Param, UINT WM);
```

Set the value of the following Windows Message WM as follows:

- For applications with an MFC dialog, use WM_CLOSE.
- For applications that directly use a CGXGridWnd-based class on WinForms, use WM_USER.
- For applications that use a Grid-based .NET control, use WM_USER+1.

The MainFrame object with CGXGridView is hidden if PrintWnd() is called.

15.6.2 Setting Print and Print Preview Functionality in a CGXGridWnd-Derived Grid

Include the header file from the DLL:

```cpp
#include "..\Print\export.h"
```

15.6.3 Examples

Objective Grid supplies full MFC print and print preview support for CView-derived grids and additional regular DLL for print and print preview support for CWnd-derived grids. A dialog is supplied for manipulating page header and footer text. The Print DLL uses Print or Print Preview features in a similar manner for a CGXGridWnd-derived grid. However, in .NET applications, you'll need to incorporate additional message handling to close the application from the Print Preview UI. For more details, please refer to the Microsoft “Technical Note 30: Customizing Printing and Print Preview.”

Three samples, GridDlgPrint, GridFormPrint and WinFormControlPrint, demonstrate the various uses of Print.dll. Please review the source code of each of these samples to view the similar, but slightly different, source code implementations of Print.dll functionality.
The following example code demonstrates print preview functionality in a dialog, WinForm and .NET control on a WinForm.

**Dialog**

```cpp
void CTestDlg::OnPrintPreview()
{
    this->ShowWindow(SW_HIDE);
    Preview( this->m_hWnd, m_wndGrid.GetParam(), WM_CLOSE);
}

void CTestDlg::OnPrint()
{
    PrintWnd( this->m_hWnd, m_wndGrid.GetParam(), WM_CLOSE);
}
```

**WinForm**

```csharp
System::Void Preview_Click(System::Object^ sender,
System::EventArgs^ e)
{
    this->Hide();
    Preview((HWND)Handle.ToInt32(), m_pGX->GetParam(), WM_USER);
}

System::Void Print_Click(System::Object^ sender,
System::EventArgs^ e)
{
    PrintWnd((HWND)Handle.ToInt32(), m_pGX->GetParam(), WM_USER);
}
```

```csharp
virtual void WndProc( Message &m ) override
{
    if (m.Msg == WM_USER)
    Application::Exit();
    __super::WndProc (m);
}
```

**.NET control**

```csharp
System::Void PrintPreview(IntPtr handle)
{
    Preview((HWND)handle.ToInt32(),
    m_pGX->GetParam(),
    WM_USER + 1);
}

System::Void Print(IntPtr handle)
{
    PrintWnd((HWND)handle.ToInt32(),
    m_pGX->GetParam(),
    WM_USER + 1);
}
```

// This is called on a WinForm containing
// .NET control with CGXGridWnd
protected override void WndProc (ref Message m )
{
if (m.Msg == WM_USER + 1)
    Application.Exit();
base.WndProc (ref m);
}

15.6.4 Related Samples

Use of Print.dll is demonstrated in the sample <Install_dir>\Samples\Grid\Print Preview.

The solution file, Print.sln, includes the following projects:

- **Print** – DLL with Print and Print Preview functionality
- **GridDlgPrint** – Modified application from shipped sample
  <Install_dir>\Samples\Grid\Tutorial\DlgGrid.
- **WinFormPrint** – Modified application from shipped sample
  <Install_dir>\Samples\Grid\Integration with .NET\GridForm
- **GridControl** and **WinFormControlPrint** – Modified .NET control and application
  from the shipped sample <Install_dir>\Samples\Grid\Integration with .NET\GridControl

To test the sample, launch Visual Studio by clicking on the Print.sln build solution, then set one of application projects as the startup project and run the sample.

**Note:** By default, Print.dll is configured for dynamic linking only. The preprocessor definition, _PRINT, must be added to your application to properly link and use Print.dll functionality.
15.7 Using This Product Outside MFC

Most of our GUI components for building Microsoft Windows applications are class libraries built on top of MFC classes, and are intended to provide complete integration with MFC. But what about companies who use other frameworks, like Zugg Software’s zApp, or need to maintain legacy Windows SDK code?

Here’s an approach that lets non-MFC applications access MFC components using Stingray Objective Grid, a library of MFC extension classes for implementing grid controls. Instead of rewriting a whole project, you can subclass and customize Objective Grid classes with MFC and export an interface to your existing application. This approach works well even when using Objective Grid from .NET languages like C# or VB.NET. You can find more details in the Knowledge Base article "Stingray Studio MFC Projects Integration with .NET". To get to the Knowledge Base, go to kb.roguewave.com/kb/ and click on the Stingray link.

It is possible to convert into a regular DLL not just a dialog-based application, but also SDI or MDI applications with rich functionality. You can find an additional sample with an SDI application converted into a DLL in `<stingray-installdir>\Samples\Grid\General\PrintPreview`.

In this section, we create a regular DLL that registers a customized grid as a window class in the Microsoft Windows environment. This way you can incorporate Objective Grid into a dialog the same way as a regular Windows SDK control. You can then instantiate a grid window through a call to the Windows SDK CreateWindow() method, or embed the grid as a custom control into a dialog template with a resource editor. Here are the steps:

1. Create a DLL to export the grid window.
   a. First, create a regular DLL with Visual Studio by selecting File | New | Project, and select **MFC DLL**.

Figure 129 – New Project
b. Specify ogregdll as the **Project name** and click OK.

c. In the dialog that appears, select **Regular DLL with MFC statically linked**.

**Figure 130 – MFC DLL Wizard**

![MFC DLL Wizard](image)

- Click Finish to close the window.

2. **Add the initialization code for Objective Grid.**

a. To do this, open the file `stdafx.h` and insert:

   ```
   #include <grid\gxall.h>
   ```
   at the end of the file.

b. Bring up the resource tree by selecting **Resource View** from the **View** menu.

c. Right-click on the root node (`ogregdll.rc`) and select **Resource Includes** from the context menu.

d. In Resource Includes, add

   ```
   #include "grid\gxresrc.h"
   ```
   to the **Read-Only Symbol Directives**, and

   ```
   #include "grid\gxres.rc"
   ```
   to the end of **Compile-time Directives**, and click OK.
e. A warning dialog box saying, “Directive Text will be written into your resource script and may render it un compilable.” pops up. Discount the warning and click OK to save the resource file.

f. Now, select **Class View** from the **View** menu.

g. Right click on **COgreddllApp** and select **Properties** from the context menu.

   All the overridable functions will be shown at the bottom.

h. Click on **Overrides** and override **InitInstance** and **ExitInstance** if they are not already overridden.

Figure 131 – ClassWizard

![ClassWizard Image]

i. Add a call to **GXInit()** in **InitInstance()**.

   Normally no termination code is necessary. However, if you are dynamically loading and unloading a DLL or an OCX that uses Objective Grid, you should add a call to **GXForceTerminate()** in the DLL’s or OCX’s **ExitInstance()**.

j. In each case, select **<Edit Code>** in the drop down box to jump to the code.

k. After editing, save and exit the code window.

3. **Add a customized class for** Objective Grid.

   a. Open **Class View** again, right click on **ogregdll** and select **Add Class** from the **Add** menu.

   b. Select **Visual C++|MFC|MFC Class** to add a MFC Class to the project.

   c. In the New Class dialog box specify **CWnd** as the base class and name it **CMyGridWnd**.

   d. Click Done in the dialog.
e. Now open MyGridWnd.cpp and MyGridWnd.h in the editor and replace all occurrences of CWnd with CGXGridWnd.

f. Add the Objective Grid macro GRID_DECLARE_REGISTER() to the class declaration in MyGridWnd.h and GRID_IMPLEMENT_REGISTER(CMyGridWnd, CS_DBLCLKS | CS_GLOBALCLASS, 0, 0, 0) to MyGridWnd.cpp.

You can also use ClassWizard to add methods to this class, such as OnLButtonDblClk() to customize the standard grid behavior.

4. Register and Compile.

a. Add a call to CMyGridWnd::RegisterClass in InitInstance() and include CMyGridWnd to the stdafx.h file.

```cpp
BOOL CogregdllApp::InitInstance()
{
    GXInit();
    CMyGridWnd::RegisterClass();
    CWinApp::InitInstance();

    return TRUE;
}
```

b. Compile as usual.

5. Use.

a. To use CMyGridWnd outside MFC, just call LoadLibrary("ogregdll.dll") from your application.

You can instantiate a grid by calling:

```cpp
CreateWindow("CMyGridWnd", ...)
```

with your usual parameters, or by using it in a dialog template.
15.8 Internationalization

15.8.1 Fonts

Objective Grid automatically uses the system font as the default font. If you assign fonts to cells, be sure to use fonts that support international characters.

15.8.2 Far Eastern Languages

Objective Grid supports both Unicode and DBCS. That means you can either develop Unicode applications that support Japanese and Chinese or you can develop DBCS applications for these languages that run under supported platforms.

Unicode is not supported in this version of the Excel read/write classes. When building Grid libraries with Unicode, an alternative to the read/write classes is Excel Automation, discussed in Section 26.7, “Excel Automation with Objective Grid.”

All edit cells support the IME window and can handle double byte characters.

Please, see Section 5.15.4, “Character Sets.” for a workaround for enabling support for the Japanese locale.

Also, if Grid is built with ANSI-DBCS and is used with the Japanese locale, it is more effective to use CGXRichEditCtrl instead of CGXEditControl in cells where Japanese text is displayed.

15.8.3 Resources

Objective Grid provides resource files in English, French, Dutch, and German. By default, English resources are loaded during initialization. To use an alternate language, you need to do one of the following:

- Include grid\gxresdeu.rc in your application’s resource file. Then there is no need to ship gx##deu.dll.

- Build gx##deu.dll and call GXInit("Deu") in your InitInstance() method. Calling GXInit("Deu") only works when you are using Objective Grid as a shared DLL.

Be sure not to include gxres.rc in your applications resource file when using GXInit("Deu"). Otherwise, gxres.rc will be used for looking up resources.
15.9 Struct Member Alignment

The Objective Grid libraries have been built with struct member alignment set to 8 bytes. Please make sure your application also uses this calling convention.

To set struct member alignment to 8 bytes in Microsoft Visual Studio:

1. Select Solution Explorer from the View menu.
2. Right click on the root node and select Properties.
3. Once the dialog appears, click the C/C++ tab and select Code Generation from the Category combo box.

The struct member alignment combo box will appear.

If you require another struct member alignment, you will have to rebuild the Objective Grid libraries with the same calling convention. Otherwise you will get unresolved externals when linking to Objective Grid.

To rebuild the Objective Grid libraries with a 1-byte alignment:

1. In gxall.h change pshpack8 to pshpack1—as shown in the code below.

   ```c
   #ifndef _GXALL_H_
   #define _GXALL_H_

   // OG uses 8 byte packing for structures
   #if _MFC_VER >= 0x0300
     // pshpackX.h saves the current #pragma pack value
   #include /* */ "pshpack1.h"
   #else
     #pragma pack(1)
   #endif

   #if _MFC_VER >= 0x0300
     #include <afxcmn.h> // MFC support for Windows 95 Common Controls
     #include <afxole.h> // MFC support for OLE
   #endif

   2. Compile the Objective Grid library with the /Zp1 option.
16.1 Grid Architecture Review

As you try to optimize the performance of your grid, it is useful to keep the Objective Grid architecture in mind.

The CGXGridParam class encapsulates all the data needed to persist a grid. It is commonly referred to as the Param object, which maintains attributes and objects being shared by several grid views of one document. It has pointers to the styles map with the grid-wide base styles, the properties object, the data object with the cell contents, the Undo/Redo list, selected ranges, and covered cells. By default, a CGXGridParam object is created in the initialization code for Objective Grid.

The CGXData class holds the grid’s data. An instance of this class is stored in the Param object. This object is not used for database bound grids, which do not actually store data values inside the grid. Additionally, formula grids use a specially-derived CGXData class, CGXFormulaSheet, to hold data used by the formula engine.

The CGXStyle class stores all the state information for a particular cell—including things like its value, its font, its background color, the type of border it has, etc. Each cell can have an individual style object (stored in the CGXData).

CGXFormulaSheet, CGXData, and CGXGridCore have some of the same methods. When you call CGXGridCore methods, validation and other processes take place. If you use CGXData methods when possible, you can avoid some processing.

16.2 Initializing Formula Grids

16.2.1 Using Formula Sheet Methods

Speed can be a problem in initializing large formula grids. If you are using the grid with the formula engine, the fastest way to fill the grid is to use CGXFormulaSheet methods instead of CGXGridCore methods to do the assignments. If you call SetNumberRowCol() and SetTextRowCol() directly, the speed improvements will be enormous compared to calling SetValueRange() or SetExpressionRowCol() for each cell. Be aware that cells are not checked for read-only status, cell objects are not notified, and no Undo information is created. The code for this sample is located at <stingray-installdir>\Samples\Grid\General\Formula.

16.2.1.1 SetFormulaRowCol()

```cpp
void CMyView::OnTestTestusingformulasheet()
{
    // Performance tests: Using the formulasheet methods instead of the
    // gridcore level calls will speed things up. The code below calls
    // the formulasheet method SetFormulaRowCol directly.
    // NOTE: Directly calling this method will bypass the
    // notification of the associated cell type object for a cell
    // (CGXControl::StoreStyle will not be called) and
    // the read-only state of the cell will also not be checked.

    GetParam()->EnableUndo(FALSE); //Turn off Undo, if it’s on.
    LockUpdate(TRUE);
    DWORD ti = GetTickCount();
    CGXFormulaSheet* pSheet = GetSheetContext();
    for (ROWCOL nRow = 2; nRow < 300; nRow++)
    {  for (ROWCOL nCol = 1; nCol < 10; nCol++)
        {  pSheet->SetFormulaRowCol(nRow, nCol, "=A1 + B1");     }
    }
    LockUpdate(FALSE);
    GetParam()->EnableUndo(TRUE); //Turn Undo back on.

    Redraw();
    CString msg;
    msg.Format("%d Ticks", GetTickCount()-ti);
    AfxMessageBox(msg);
}

void CMyView::OnTestTestusinggridmethods()
{
    // The code below uses the call to the GridCore SetExpressionRowCol.

    GetParam()->EnableUndo(FALSE); //Turn off Undo, if it’s on.
    LockUpdate(TRUE);
    DWORD ti = GetTickCount();
    for (ROWCOL nRow = 2; nRow < 300; nRow++)
    {  for (ROWCOL nCol = 1; nCol < 10; nCol++)
        {  SetExpressionRowCol(nRow, nCol, "=A1 + B1");     }
    }
```
LockUpdate(FALSE);
GetParam()->EnableUndo(TRUE); //Turn Undo back on.
Redraw();
CString msg;
msg.Format("%d Ticks", GetTickCount()-ti);
AfxMessageBox(msg);
}

16.2.1.2 SetNumberRowCol() and SetTextRowCol()

// Performance tests:

// Using SetNumberRowCol/SetTextRowCol directly instead of
// SetValueRange or SetExpressionRowCol will speed up the
// initialization of the grid enormously (just as fast as filling
// an array).
//
// Check it out below!
//
// NOTE: Directly calling these methods will bypass the
// notification of the associated cell type object for a cell
// (CGXControl::StoreStyle will not be called) and the readonly
// state of the cell will also not be checked.
//
DWORD ti = GetTickCount();
CGXFormulaSheet* pSheet = GetSheetContext();
CGXStyle style;
for (; nRow < 300; nRow++)
{
    for (ROWCOL nCol = 1; nCol < 10; nCol++)
    {
        CString s;
        s.Format("%d/%d", nRow/100, nCol);
        // style.SetValue("Hello");
        // StoreStyleRowCol(nRow, nCol, &style, gxOverride, 0);
        // pSheet->SetNumberRowCol(nRow, nCol, (double) nRow+nCol);
        pSheet->SetTextRowCol(nRow, nCol, _T("Hello"));
    }
}

CString msg;
msg.Format("%d Ticks", GetTickCount()-ti);
AfxMessageBox(msg);

16.2.2 Copying Cells

Using a combination of SetExpressionRowCol() and CopyCells() also provides fast initialization. Here is an example of this technique:

SetRowCount(10000);
SetColCount(50);
COleDateTime t, t1;
t = COleDateTime::GetCurrentTime();
CString s;
for (ROWCOL i = 1; i >=1000 ; ++i)
\{ 
    s.Format(="A%d-B%d", i, i);
    SetExpressionRowCol( i, 3, s );
\}
CGXRange range( 1, 3, 1000, 3);
for ( i = 1001; i <= 9001; i+=1000 )
{ 
    CopyCells(range, i, 3);
}
t1 = COleDateTime::GetCurrentTime();
s.Format("%f seconds\n", (t1-t) * 24. * 60. * 60.);
AfxMessageBox(s);

If this is a standard initialization, using an OGF file might be a way to quickly load it. (OGF is the file format for formula grids, while OGL is the file format for a standard grid.)
16.3 Populating the Grid

16.3.1 Improving Performance

If populating a grid takes too long, consider these techniques to speed things up. To improve performance:

- Don’t insert rows one at a time.
  
  In many cases this is the primary reason that things seem to slow down after the insertion of many rows. If possible, before you call your code that fills cells, go ahead and call `CGXGridCore::SetRowCount()` with the total number of rows that you need.

  While you are at it you can go ahead and call `SetColCount()` with the total number of columns you need, although it isn’t really a performance issue.

- Use the Data Object methods, e.g. `CGXData::StoreValueRowCol()`, to change the value of cells.

  Don’t call `CGXGridCore::SetValueRange()` to actually place the value in the grid. If you directly call `CGXData::StoreValueRowCol()` instead, this will bypass several pieces of grid code (`CGXGridCore` methods) that tend to affect performance. Using `CGXData::StoreValueRowCol()` can really speed things up. You would use a call something like this:

  ```
  GetParam()->GetData()->StoreValueRowCol(...)
  ```

  Use `CGXData::StoreStyleRowCol()` instead of `CGXGridCore::StoreStyleRowCol()`.

- Call `CGXGridCore::LockUpdate(TRUE)` to prevent redraws during multiple grid update calls.

  Normally each command will update the grid (or invalidate the affected rectangle) immediately, which will result in substantially slowing down performance.

  As you make a series of value changes, using `LockUpdate()` to prevent redraws will save significant time because subsequent commands need not invalidate the grid (and compute cell/window coordinates). If you have several subsequent commands that will change the grid, call `LockUpdate(TRUE)`. After your series of commands, call `LockUpdate(FALSE)` and `Redraw()` to refresh the whole grid.

  The following example illustrates how to use `LockUpdate()` while maintaining its original setting:

  ```
  void CMyGridView::OnFormatResizecols()
  {
      // if no cells are selected, copy the current cell's
      // coordinates
      CGXRangeList selList;
      if (!CopyRangeList(selList, TRUE))
          return;
```
CGXLongOperation theOp;

ResetCurrentCell( );

BeginTrans(GXGetAppData( )->strmResizeCols);

// stop updating the display for subsequent commands
BOOL bLockOld = LockUpdate(TRUE);

TRY
{

// series of commands
// (ResizeColWidthsToFit will do a lot of SetColWidth
// commands)
POSITION pos = selList.GetHeadPosition( );
while (pos)
{
    if (!ResizeColWidthsToFit(selList.GetNext(pos)))
        AfxThrowUserException( );
}

CommitTrans( );
}
CATCH(CUserException, e)
{
    Rollback( );
}
END_CATCH

// Now, refresh the whole grid
LockUpdate(bLockOld);
Redraw( );

◆ Don't generate Undo information while populating your grid.

As you populate your grid, Objective Grid may be generating and maintaining Undo information that will allow actions to be undone by the user. Generating and maintaining this Undo information can be time consuming. If you do not want your user to be able to undo the work you are doing to populate the grid, then you should disable the grid Undo mechanism. To do so, call GetParam()->EnableUndo(FALSE); before the code that populates the grid. If you want to enable the grid’s Undo support after your grid has been populated, call GetParam()-> EnableUndo(TRUE); after it has been populated.

◆ Set the grid to read-only.

Use CGXGridCore::SetReadOnly() and CGXGridParam::SetLockReadOnly() to manage the read-only status of the grid.

◆ Use virtual mode with large data sets.

If you need to display a very large amount of data and find that populating the grid takes too long, instead of storing the data in the grid, consider using virtual mode for better performance. Virtual mode is ideally suited for displaying grids with many rows.

The great thing about virtual mode is that the grid only needs data for cells that are currently in view (not every row). This means that you supply values to the grid (from your data structure) only when the grid requests them. You do this by overriding
GetStyleRowCol() and supplying the values there on demand. If this is a read-only grid, you would not have to override StoreStyleRowCol(), which is the second piece of a virtual grid.

Use GetStyleRowCol() instead of SetStyleRange(), as in this example:

```cpp
BOOL CMyGridWnd::GetStyleRowCol(ROWCOL nRow, ROWCOL nCol, CGXStyle &style, GXModifyType mt, int nType)
{
    BOOL ret= CGXGridCore::GetStyleRowCol( nRow, nCol, style, mt, nType);
    if(nType >= 0)
    {
        if (nCol>1&&nRow>1&&nCol<367&&nRow<151)
        {
            style.SetValue(value);
            style.SetInterior(color);
        }
    }
    return ret;
}
```

Take a look at the VirtGrid tutorial that ships with Objective Grid to see how you might use virtual mode. Refer to Section 5.9, “Virtual Grids,” and Chapter 10, “VirtGrid Tutorial,” for more information on Virtual Grids.

Keep code out of GetRowCount() and GetColCount().

GetRowCount() and GetColCount() are used extensively to validate values within routines. In addition, because HitTest() is called repeatedly on every mouse move and because this method touches a lot of grid code, including GetRowCount(), it is not unusual for GetRowCount() to be called thousands of times in a short period of time. Because GetRowCount() and GetColCount() are called quite often, your overrides of these methods are not an appropriate place to do extensive calculation.

See Section 5.5.3 for more information on Hit-Testing.
16.4 Speeding Up Computations

Minimal recalculation, in effect whenever the recalculation mode is set to Automatic and the method is set to Foreground, means that only the cells that are potentially affected by an operation (such as editing a cell, or moving a range of cells) are recalculated. This feature minimizes the work performed during recalculation and thus speeds up your computations.

For more information about different calculation modes and methods, please refer to Section 14.4, “Calculations.”

16.5 Using Profiling in Visual Studio

If you are still experiencing slow performance, use profiling in Visual Studio to analyze which function is slowing down the program. Refer to Visual Studio documentation on profiling applications.

16.6 Reducing Application Size

The simplest way to build smaller libraries is to run the Objective Grid Build Wizard from the Start menu, and (on the components selection screen) only include the components you need for your application. You can specify your own name for the library. Continue with the Build Wizard and Visual Studio to the last screen, and then use the Build | Batch Build menu option to build the libraries you need. On the last screen of the Build Wizard, you will see a description of how to include your custom library in your projects (instead of the default library).

For more information on reducing the size of your application, please refer to Chapter 6, “Reducing the Size of Your Application.”
CSliderCtrl Tutorial

17.1 Introduction

The goal of this tutorial is to demonstrate how to embed pre-existing controls into the grid. The following pages give an overview of the project, explain how to execute the program, discuss how to verify the program output, and explain the relevant parts of the source code in detail.

17.2 CSliderCtrl Overview

Periodically, programmers encounter situations where they want to embed a pre-existing C++ control into the grid. It could be an MFC control or a custom control they wrote from scratch. The following tutorial tells you how to create a grid control from a CSliderCtrl.

The grid control (CGXSilderCtrl) is created by multiply inheriting from CGXControl and CSliderCtrl and overriding the key virtual functions in CGXControl. Also, following the control-sharing paradigm of the grid architecture you need to map all the necessary data attributes of the CSliderCtrl (such as range and position) to appropriate cell style’s attributes. In this tutorial the position of the slider is stored as the value attribute in the style and the minimum and maximum positions are stored as custom user attributes in the style. If you need to use more attributes of the CSliderCtrl, then you could create more custom user attributes to store their values in the style. The tutorial assumes that you are familiar with the control architecture of the grid as explained in Chapter 2, “Design Overview.”

Please refer to the sample located at <stingray-installdir>\Samples\Grid\Tutorial\CSliderCtrl. 
17.3 The CSliderCtrl Program

This section examines the tutorial code in detail.

17.3.1 The Declaration

class CGXSliderCtrl : public CGXControl, public CSliderCtrl {
    GRID_DECLARE_CONTROL(CGXSliderCtrl)
public:
    BOOL m_bIsActive;
    BOOL m_bModify;
    // Constructor and destructor
    CGXSliderCtrl(CGXGridCore* pGrid);
    virtual ~CGXSliderCtrl();
    BOOL CreateControl(int nID);
    virtual BOOL Store();
    //Operations
    virtual void Draw(CDC* pDC, CRect rect, ROWCOL nRow,
        ROWCOL nCol, const CGXStyle& style,
        const CGXStyle* pStandardStyle);
    virtual void Init(ROWCOL nRow, ROWCOL nCol);
    virtual void SetValue(LPCTSTR pszRawValue);
    virtual BOOL GetValue(CString& strResult);
    //virtual BOOL GetControlText(CString& strResult,
    //    ROWCOL nRow, ROWCOL nCol, LPCTSTR pszRawValue,
    //    const CGXStyle& style);
    virtual BOOL GetModify();
    virtual void SetModify(BOOL bModified);
    // Status
    virtual void SetActive(BOOL bActive);
    virtual BOOL IsActive();
    virtual void Hide();
    // Mouse hit
    virtual BOOL LButtonUp(UINT nFlags, CPoint pt, UINT nHitState);

protected:
    CSliderCtrl* m_pStaticWnd;  // For drawing static cells

    // Generated message map functions
protected:
    AFX_MSG_MAP(CGXSliderCtrl)
    //}AFX_MSG
    GRID_DECLARE_MESSAGE_MAP()
};

As stated earlier, the grid control CGXSliderCtrl derives multiply from CGXControl, the abstract interface that the grid understands, and the CSliderCtrl. There is a second instance of CSliderCtrl as this class’s member m_pStaticWnd. The reason we have two instances of CSliderCtrl is to aid control sharing among different cells. While, the multiply inherited control is always used to draw an active cell, the second instance m_pStaticWnd is used to draw all the inactive cells. The
CGXSliderCtrl is created using the construct and create paradigm, similar to the CSliderCtrl. We create both the inherited CSliderCtrl and the member CSliderCtrl in the CreateControl() function.

We will examine the rest of the functions with their implementation.

### 17.3.2 Activating and Deactivating the Control

```cpp
// Operations
void CGXSliderCtrl::Init(ROWCOL nRow, ROWCOL nCol)
{
    ASSERT(::IsWindow(m_hWnd));
    // ASSERTION-> Did you forget to call Create? ->END

    // Stores the cell coordinates, resets the style and
    // sets the window text
    CGXControl::Init(nRow, nCol);

    SetActive(FALSE);

    // SetValue will set the window text for the current cell
    NeedStyle();
    SetValue(m_pStyle->GetValueRef());                                  //1

    SetModify(FALSE);
}
void CGXSliderCtrl::SetValue(LPCTSTR pszRawValue)
{
    int nPos = _ttoi(pszRawValue);
    if( nPos < 0)
        nPos = 50;
    SetPos(nPos);
    OnModifyCell();
}
```

Init() is a virtual function in the CGXControl class that will be called every time a cell becomes a current cell so that you can initialize the intrinsic state of the control before it becomes active.

```cpp
//1 initializes the intrinsic state for this control.

// Mouse hit
BOOL CGXSliderCtrl::LButtonUp(UINT nFlags, CPoint pt, 
    UINT nHitState)
{
    // Unreferenced:
    nFlags, pt, nHitState;

    SetActive(TRUE);                      //1
    Refresh();                             //2

    CRect r = GetCellRect(m_nRow, m_nCol);
    pt.x -= r.left;
    pt.y -= r.top;

    PostMessage(WM_LBUTTONDOWN, MK_LBUTTON, MAKELONG(pt.x, pt.y));}//3
```
PostMessage(WM_LBUTTONDOWN, MK_LBUTTON, MAKELONG(pt.x, pt.y));

// check child buttons
CGXControl::LButtonUp(nFlags, pt, nHitState);

return TRUE;
}

BOOL CGXSliderCtrl::Store()
{
    // Calls SetStyleRange() and resets the modify flag
    ASSERT(m_pStyle);

    CString sValue;
    if (m_pStyle && GetModify() && GetValue(sValue)) //1
    {
        SetActive(FALSE);                              //2
        SetModify(FALSE);

        return Grid()->SetValueRange(
            CGXRange(m_nCol, m_nCol),
            sValue,
            gxOverride,
            0, GX_INVALIDATE);
        // Cell will be automatically redrawn inactive
    }

    return TRUE;
}

LbuttonUp() is another virtual in CGXControl that will be called by the grid whenever the user clicks on an inactive cell.

//1 Changes the state of the control to be active.
//2 Invalidates the cell rectangle and forces the cell to be drawn in the new active state.
//3 Posting the button down and up messages to the active control ensures immediate response from the active control.

Note that the active state is something you should maintain in your derived control. The member m_bActive and the virtual overrides SetActive() and GetActive() take care of this.
return Grid()->SetValueRange(
    CGXRange(m_nRow, m_nCol),
    sValue,
    gxOverride,
    0, GX_INVALIDATE);                           //3
    // Cell will be automatically redrawn inactive
}

return TRUE;
}

Store() is another CGXControl virtual function that will be called by the grid when the changes in
the cell made by the user has to be stored in response to a user action. Note that you will have to
maintain the dirty state of the control in the derived class. The member m_bModify and the virtuals
GetModify() and SetModify() take care of this.

//1 Check if the cell is dirty; if it is, get the current position of the slider as a string to be stored
as the value. GetValue() performs the conversion to a string.

//2 Clear the active state and the dirty flag.

//3 Store the new value into the grid using SetValueRange(). Note that you could instead use
SetStyleRange() to make changes in other attributes of the cell’s style.

17.3.3 Drawing the Active and Inactive State

void CGXSliderCtrl::Draw(CDC* pDC, CRect rect, ROWCOL nRow,
ROWCOL nCol, const CGXStyle& style,
const CGXStyle* pStandardStyle)
{
    ASSERT(::IsWindow(m_hWnd));
    // ASSERTION-> Did you forget to call Create? ->END

    ASSERT(pDC != NULL && pDC->IsKindOf(RUNTIME_CLASS(CDC)));
    // ASSERTION-> Invalid Device Context ->END
    ASSERT(nRow <= Grid()->GetRowCount() && nCol <=
    Grid()->GetColCount());
    // ASSERTION-> Cell coordinates out of range ->END

    ASSERT_VALID(pDC);
    // Draw the decorated window

    // Erase Frame around the cell
    DrawFrame(pDC, rect, style);

    // Font
    rect = GetCellRect(nRow, nCol, &rect, &style);

    CSliderCtrl* pWnd;
    CFont font;

    if (Grid()->IsCurrentCell(nRow, nCol) && nRow == m_nRow
    && nCol == m_nCol)
    {

// current cell -
pWnd = this;
int nPosMin =
    style.GetUserAttribute(GX_IDA_SLIDER_MIN).GetLongValue();
int nPosMax =
    style.GetUserAttribute(GX_IDA_SLIDER_MAX).GetLongValue();
pWnd->SetRange(nPosMin, nPosMax);                // 2
}
else
{
    // any other cell - use m_pStaticWnd
    // use static window and set format and initialize
    // window with cell settings
    pWnd = m_pStaticWnd;
    int nPosMin =
        style.GetUserAttribute(GX_IDA_SLIDER_MIN).GetLongValue();
    int nPosMax =
        style.GetUserAttribute(GX_IDA_SLIDER_MAX).GetLongValue();
pWnd->SetRange(nPosMin, nPosMax);
    if(style.GetIncludeValue())
    {
        int nPos = _ttoi(style.GetValue());
        if(nPos < 0)
            nPos = 50;
        pWnd->SetPos(nPos);
    }
}
pWnd->MoveWindow(rect, FALSE);                      //3
pWnd->Invalidate();
pWnd->ShowWindow(SW_SHOW);
if (nRow > Grid()->GetFrozenRows() &&
    (Grid()->GetTopRow() > nRow || nCol > Grid()->GetFrozenCols() &&
     Grid()->GetLeftCol() > nCol))
    // Ensure that the window cannot draw outside the
    // clipping area!
    {       // 4
        CRect rectClip;
        if (pDC->GetClipBox(&rectClip) != ERROR)
        {
            CRect r = rect & Grid() -> GetGridRect();
            GridWnd()->ClientToScreen(&r);
            pWnd->ScreenToClient(&r);
            GridWnd()->ClientToScreen(&rectClip);
            pWnd->ScreenToClient(&rectClip);
            pWnd->ValidateRect(r);
            pWnd->InvalidateRect(rectClip);
        }
    }
pWnd->UpdateWindow();
// if (nRow == m_nRow && nCol == m_nCol && m_bIsActive)
// initialize CWnd, make it visible and set focus
if (nRow == m_nRow && nCol == m_nCol && IsActive() && !Grid()->IsPrinting() && !(Grid()->GetTopRow() > nRow || Grid()->GetLeftCol() > nCol))
{
    // 5
    Grid()->SetIgnoreFocus(TRUE);
    if (pWnd->GetFocus() == GridWnd())
        pWnd->SetFocus();
    Grid()->SetIgnoreFocus(FALSE);
    SetModify(TRUE);
}
else
{
    pWnd->SetWindowPos(0,32000,32000,0,0,
    SWP_NOSENDCHANGING|SWP_NOCOPYBITS|SWP_NOACTIVATE|SWP_NOREDRAW);//6
    pWnd->ShowWindow(SW_HIDE);
    GridWnd()->ValidateRect(rect);
}

// Avoid overdrawing with BitmapDC in CGXGridCore::OnDraw
ExcludeClipRect(rect);

// Unreferenced:
pStandardStyle;
}

//1 If this was called for the active cell, initialize the range for the active cell.
//2 If this was called for the inactive cell, initialize the range and position of the inactive cell.
//3 Move the corresponding control to this rectangle, invalidate it, and show it.
//4 Update the window so that it will be redrawn with its new settings.
//5 If called for the active state, set the focus to the active control.
//6 If called for the inactive state, hide the static control and validate the rectangle so that the grid will not try to redraw it.
18.1 Introduction

The goal of this tutorial is to demonstrate binding the grid to an external data source using the browser grid classes CGXBrowserView/Wnd. The following pages will give you an overview of the project and explain the relevant parts of the source code in detail.

18.2 BrwsGrid Overview

This tutorial addresses the same issue of binding the grid to an external data source as the VirtGrid tutorial. But, the grid in this tutorial will provide extensive editing capabilities like basic field editing, adding rows and deleting rows, and enable the extensive list of grid features like floating cells, covered cells, hide rows/columns, sort rows, move columns, etc.

Take a look at the CGXBrowserGrid class description for detailed information on the browsing functionality it provides.

The explanation in this tutorial assumes that you are familiar with the “virtual mode” architecture and the browser architecture of the grid. If not, please take a look at Section 2.5, “Browser Architecture,” and Chapter 5, “Using Objective Grid.”

For illustration purposes, we are going to use an instance of CGXData, m_data, as the external data source to which the grid will be bound. Note that in your application you might have your own data structure or stubs that talk to an external data source.

Most of the functionality of this example is implemented inside overridden virtual functions.

Please refer to the sample located at <stingray-installdir>/Samples/Grid/Tutorial/BrwsGrid.
18.3 The BrwsGrid Program

This section examines the BrwsGrid program you just ran. The grid in this program displays and allows you to edit the data stored in the data member m_data of the document.

In the following sections, code modules are followed by the explanations of the comments.

18.3.1 Initialization and Setting Column Count

```cpp
void CBrwsGridView::OnInitialUpdate()
{
    CGXBrowserView::OnInitialUpdate();

    LockUpdate(TRUE);
    GetParam()->EnableUndo(FALSE);

    InitBrowserSettings(); //1
    GetParam()->EnableTrackRowHeight();
    SetColCount(
        GetDocument()->m_data.GetColCount()); //2
    SetStyleRange(CGXRange().SetCols(4),
        CGXStyle().SetControl(GX_IDS_CTRL_CHECKBOX)); //3
    SetStyleRange(CGXRange().SetCols(1),
        CGXStyle().SetInterior(RGB(150,150,150)));
    SetStyleRange(CGXRange().SetCols(2),
        CGXStyle().SetTextColor(RGB(0,0,255)));
    for(ROWCOL col = 1; col <=
        GetDocument()->m_data.GetColCount(); col++) //4
    {
        SetValueRange(CGXRange(0, col),
            GetDocument()->m_data.GetValueRowCol(0, col));
    }

    GetParam()->SetSortRowsOnDblClk(TRUE);

    GetParam()->EnableUndo(TRUE);
    LockUpdate(FALSE);
    Redraw();
}
```

//1 Initializes the default browser settings.

//2 Set the column count in the grid based on the number of columns in your external data source. This allows you to set the base styles for the columns in the following lines.

//3 The following lines change the column base style in the corresponding columns, thereby affecting all the cells in those columns. Note that the browser grid does not allow you to set the styles for individual cells.

//4 Initialize the column names here.
18.3.2 Supplying the Row Count Dynamically

// Supply the record count to the grid.
long CBrwsGridView::OnGetRecordCount()
{
    return GetDocument()->m_data.GetRowCount();        //1
}

//1 Supply the row count from your external data source here. This function will be called
every time the grid has to be redrawn.

18.3.3 Managing Sorted Rows and Moved Columns

short CBrwsGridView::GetFieldFromCol(ROWCOL nCol)
{
    return (short)
    (GetColIndex(nCol) - GetHeaderCols());       //1
}
long CBrwsGridView::GetRecordFromRow(ROWCOL nRow)
{
    return (long) GetRowIndex(nRow) -
    (long) GetHeaderRows();                        //2
}

//1 The browser grid enables moving columns by maintaining a map which translates the cur-
rent moved column position into the original column before moving. This function converts
the current column to the corresponding field in your data source. If your data source has
columns that are zero-based, then you should make this function return the appropriate
field number.

//2 The above discussion holds true for sorted rows as well.

18.3.4 Supplying the Value To the Grid and Storing the Changes

BOOL CBrwsGridView::OnLoadCellStyle(ROWCOL nRow, ROWCOL nCol, CGXStyle & style,
    LPCTSTR pszExistingValue)
{
    int nField = GetFieldFromCol(nCol);                   // 1

    if (pszExistingValue == NULL && nRow >= GetFirstRow())// 2
    {
        if (!m_bNoValueNeeded && nField != -1 && nField <= (short)
            GetDocument()->m_data.GetColCount())  // 3
            {
                style.SetValue(GetDocument()->m_data.GetValueRowCol(
                    GetRecordFromRow(nRow), nField));// 4
            }
    }
    else
        style.SetEnabled(FALSE);
    return TRUE;
}
```cpp
void CBrwsGridView::OnFlushCellValue(ROWCOL nRow, ROWCOL nCol, LPCTSTR pszChangedValue) // 5
{
    // cannot handle this.
    if (pszChangedValue == NULL)
        return;

    int nField = GetFieldFromCol(nCol);

    if (nField <= 0 || nField > (short)GetDocument()->m_data.GetColCount()) // 6
    {
        SetWarningText(_T("You have tried to modify the value of an unbound field!"));
        AfxThrowNotSupportedException();
        return;
    }

    GetDocument()->m_data.StoreValueRowCol(GetRecordFromRow(nRow), nField, pszChangedValue, gxOverride);    //7
}
```

//1 Translate the column number to the field count in your data source.

//2 If pszExistingValue == NULL, then the grid already has the value in its internal buffer and you should only supply specific formatting here.

//3 If m_bNoValueNeeded == TRUE, then you do not have to supply the value here. This flag will be set while composing styles for tooltips which do not require the value.

//4 Get the value from your data source and set it in the composed style.

//5 This function will be called when the user edits one or more fields in a row and moved to another row and for each edited field in that row. Store the change back into your data source in this function.

//6 nField is greater than the field count if you had unbound columns in your grid. If that is the case, make sure that the unbound column is disabled. (This is usually the case.)

//7 Store the value back into your data source.

18.3.5 Deleting Rows

Override DeleteRows(), as shown in the tutorial sample code. The code is not discussed here line by line, since it is very elaborate. Please refer to the source code in the sample project for detailed information.

Basically, this code deals with the issue of deleting multiple non-contiguous rows. The problem is that as soon as you delete a row, the index of any row that is to be deleted (and that is below the current deleted row) becomes invalid.

It resolves the issue by deleting rows one at a time from bottom up, so that all the indexes for the rows to be deleted remain valid.

It also updates the internal data structure that is a map of the current moved rows to their original position before they were moved.
18.4 Conclusion

Note that the example data source that we use in this tutorial provides APIs that allow you to access the entire table at any moment. But, if you need to bind to a data source that exposes its data through a cursor object with a current row state, you should at our ODBC/ADO grid implementations in `<stingray-installdir>\Samples\Grid\Database` for more details.
Chapter 19

Dynamic Splitter Window Support

19.1 Introduction

The CGXDTabWnd and CGXDSplitWnd classes in the Objective Grid library work together to provide dynamic splitter window support. Using these classes, you can create a grid application that behaves like a workbook in MS Excel.

19.2 Process

1. Generate a default SDI doc/view or MDI doc/view based skeleton application with the MFC AppWizard.

2. Include grid\gxall.h in stdafx.h.

3. Add "grid\gxresrc.h" and "grid\gxres.rc" to the Resource Includes dialog.

4. Add a call to the GXInit() function call in the InitInstance() function of the CWndApp-derived class of the application.

   BOOL CGridDSplitApp::InitInstance()
   {
     AfxEnableControlContainer();
     GXInit();
     ...
   }

5. Add grid-related data to your document class. Remember to initialize the m_pParam pointer to NULL in the constructor of the document class.

   class CGridDSplitDoc : public CDocument
   {
     public:
       CGXGridParam* GetParam()
       { return m_pParam; };

       void SetParam(CGXGridParam* pParam)
       { m_pParam = pParam; };
   protected:
6. Change the base class for your view class from `CView` to `CGXGridView`. The following steps are specific to adding dynamic splitter support.

7. Add a member variable of `CGXDTabWnd` and `CGXDSplitWnd` to the `CMainFrame` class.

```
protected:  // control bar embedded members
    CStatusBar  m_wndStatusBar;
    CToolBar    m_wndToolBar;
    CGXDSplitWnd      m_wndSplitter;
    CGXDTabWnd      m_wndTab;
```

8. Override the `OnCreateClient()` virtual function of the `CMainFrame` class for SDI application, or override the `CChildFrame` class for MDI applications and implement the function as shown below. In addition, include the header file for our view class, `GridDSplitView.h`, at the top of the `mainfrm.cpp` file. After you add the include, add a forward declaration of the `CGridDSplitDoc` class to the top of `CGridDSplitView` class.

```cpp
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext)
{
    // Specify the run-time class for the first sheet.
    // Don't forget to register the document template
    // in InitInstance!
    pContext->m_pNewViewClass=RUNTIME_CLASS(CGridDSplitView);

    // creates the tab window
    VERIFY(m_wndTab.Create(this, _T("What's New?")), pContext));

    // each view should associated with the same document.
    m_wndSplitter.m_pOwnerWnd = &m_wndTab;

    pContext->m_pNewViewClass=UNTIME_CLASS(CGridDSplitView);
    m_wndSplitter.Create(&m_wndTab,
        2, 2, // TODO: adjust the number of rows, columns
        CSize(10, 10), // TODO: adjust the minimum pane size
        pContext);
    m_wndSplitter.ShowWindow(SW_HIDE);
    m_wndTab.AttachWnd(&m_wndSplitter, "Splitter Tab");
    return TRUE;
}
9. Now, you need to add some code to the view class to support the dynamic splitter window. Override the `OnInitialUpdate()` virtual function in the view class.

```cpp
void CGridDSplitView::OnInitialUpdate()
{
    if(GetDocument()->GetParam() == NULL)
        SetParam(new CGXGridParam(), TRUE);
    else
        SetParam(GetDocument()->GetParam(), FALSE);
    DoInitialUpdate();
    GetDocument()->SetParam(GetParam());
    GetParam()->EnableUndo(FALSE);
    SetRowCount(100);
    SetColCount(20);
    SetValueRange(CGXRange(1,1), _T("Dynamic!"));
    EnableHints();
    //GetParam()->SetSmartResize(FALSE);
    GetParam()->EnableUndo(TRUE);
    UpdateScrollbars();
}
```

10. Implement the `DoInitialUpdate()` function used in `OnInitialUpdate()`. The implementation of this function is almost an exact copy of the `OnInitialUpdate()` method in the `CGXGridView` class, which is a class in the Objective Grid library. Notice that in the preceding implementation, we used some protected member variables of the `CGXGridCore` class. To give your view class access these variables, you need to make your view class a friend of `CGXGridCore`.

Let's add an empty derived class `class_core` to the top of the `DoInitialUpdate()` function.

```cpp
class _core: public CGXGridCore
{
    friend class CGridDSplitView;
};
void CGridDSplitView::DoInitialUpdate()
{
    SetGridWnd(this, GetDocument(), TRUE);
    CWnd* pWnd = DXGetParentSplitter(this, TRUE);
    if(pWnd && pWnd->IsKindOf(RUNTIME_CLASS(CGXDSplitWnd)))
        m_pSplitterWnd = (CSplitterWnd*) pWnd;
    else
        m_pSplitterWnd = NULL;
```
// Comment: I have move the following lines before
// CGXGridCore::OnGridInitialUpdate() so that scrollbars
// can be initialized correctly.
if (m_pSplitterWnd || GetSharedScrollbarParentWnd())
    SetScrollBarMode(SB_BOTH, gxnShared | gxnEnhanced);
else
    SetScrollBarMode(SB_BOTH, gxnAutomatic |
                     gxnEnhanced);
OnGridInitialUpdate();
CGXView::OnInitialUpdate();

CGXGridParam* pParam = GetParam();

// check print device
if (GetPrintDevice() == NULL)
{
    // is print device in parameter object?
    if (pParam->GetPrintDevice() == NULL)
        pParam->SetPrintDevice(new CGXPrintDevice);

    SetPrintDevice(pParam->GetPrintDevice(), FALSE);
}

// if this is a new pane in a dynamic splitter window
// I will initialize top row or left column

if (m_pSplitterWnd != NULL)
{
    ASSERT(m_pSplitterWnd-> IsKindOf(RUNTIME_CLASS(CGXDSplitWnd)));
    VERIFY(m_pSplitterWnd->IsChildPane(this, m_nSplitRow,
                                        m_nSplitCol));

    ASSERT(m_nSplitRow < 2);
    ASSERT(m_nSplitCol < 2);

    if (m_nSplitRow > 0 || m_nSplitCol > 0)
    {
        // copy settings from other pane
        CGXGridView *pView;
        pView = (CGXGridView *) m_pSplitterWnd->GetPane(0, 0);

        ASSERT(pView != NULL);
        if (pView->IsKindOf(RUNTIME_CLASS(CGXGridView)))
        {
            if (m_nSplitRow > 0)
            {
                pView = (CGXGridView *) m_pSplitterWnd->GetPane(0,
                                                                  m_nSplitCol);

                ASSERT(pView != NULL);
            }
        }
    }
}
if (pView->IsKindOf(RUNTIME_CLASS(CGXGridView)))
{
    m_nLeftCol = pView->m_nLeftCol;
    m_nTopRow = pView->m_nTopRow;
    m_bDisplayHeaderRow = FALSE;

    // disable smart redrawing of WM_SIZE message
    ((_core*)pView)->m_cxOld = 0;
    (_core*)pView)->m_cyOld = 0;
}

if (m_nSplitCol > 0)
{
    pView = (CGXGridview *)
        m_pSplitterWnd->GetPane(m_nSplitRow, 0);
    ASSERT(pView != NULL);
    if (pView->IsKindOf(RUNTIME_CLASS(CGXGridView)))
    {
        m_nTopRow = pView->m_nTopRow;
        if (m_nSplitRow == 0)
            m_nLeftCol = pView->m_nLeftCol;
        m_bDisplayHeaderCol = FALSE;

        // disable smart redrawing of WM_SIZE message
        ((_core*)pView)->m_cxOld = 0;
        (_core*)pView)->m_cyOld = 0;
    }
    m_pSplitterWnd = NULL;
    UpdateScrollbars();
}

11. Implement the DXGetParentSplitter() function that is used in the DoInitialUpdate() function. This function is implemented as static.

CWnd* CGridDSplitView::DXGetParentSplitter(const CWnd *pWnd, 
    BOOL bAnyState)
{
    CWnd* pSplitter = pWnd->GetParent();
    if (!pSplitter->IsKindOf(RUNTIME_CLASS(CGXDSplitWnd)) 
        && !pSplitter->IsKindOf(RUNTIME_CLASS(CGXDTabWnd)))
        return NULL; // not a splitter
    if (!bAnyState)
        return pSplitter;
    // ignore splitters in minimized (iconic) windows
    while ((pWnd = pWnd->GetParent()) != NULL)
        if (pWnd->IsIconic())
            return NULL;
        return pSplitter;
12. Now, delegate the `OnDraw()` function to that of `CGXGridView` class.

```cpp
type CGridDSplitView::OnDraw(CDC* pDC) {
    CGXGridView::OnDraw(pDC);
}
```

13. If you compile and run the application now, you will have a grid application with dynamic splitter support. If you choose the Splitter tab, you will see our dynamic splitter window at work. Two sets of scrollbars will be displayed. To hide one set of the scroll bars, you need to make a small change to your code.

14. Override the `GetScrollBarCtrl()` function of the `CGXGridView` class.

```cpp
CScrollBar* CGridDSplitView::GetScrollBarCtrl(int nBar) const
{
    #define _AfxGetDlgCtrlID(hWnd) ((UINT)(WORD)::GetDlgCtrlID(hWnd))
    
    ASSERT(nBar == SB_HORZ || nBar == SB_VERT);
    if (GetStyle() &
        ((nBar == SB_HORZ) ? WS_HSCROLL : WS_VSCROLL))
    {
        // it has a regular windows style scrollbar
        // (no control)
        return NULL;
    }

    CWnd* pParent = DXGetParentSplitter(this, TRUE);
    if (pParent == NULL)
        return NULL;            // no splitter

    UINT nID = _AfxGetDlgCtrlID(m_hWnd);
    if (nID < AFX_IDW_PANE_FIRST || nID >
        AFX_IDW_PANE_LAST)
        return NULL;            // not a standard pane ID

    // appropriate PANE id - look for sibling (splitter, or
    // just frame)
    UINT nIDSscroll;
    if (nBar == SB_HORZ)
        nIDSscroll = AFX_IDW_HSCROLL_FIRST +
        (nID - AFX_IDW_PANE_FIRST) % 16;
    else
        nIDSscroll = AFX_IDW_VSCROLL_FIRST +
        (nID - AFX_IDW_PANE_FIRST) / 16;

    // return shared scroll bars that are immediate
    // children of splitter
    return (CScrollBar*)pParent->GetDlgItem(nIDSscroll);
}
```
else
    nIDScroll = AFX_IDW_VSCROLL_FIRST;
    return (CScrollBar*) pParent->GetDlgItem(nIDScroll);
}
CScrollBar* pBar = (CScrollBar*) pParent->GetDlgItem(nIDScroll);

// check one parent up
if(pBar == NULL)
    pBar = (CScrollBar*) pParent->GetParent()->GetDlgItem(nIDScroll);

    return pBar;
}

15. Now, the dynamic splitter window should only have one set of scrollbars. You can use mouse to drag the splitter box to split the window in the Splitter tab window.
20.1 Introduction

The CGXPageBreakPlugin class implements the page-break UI and scaled printing support for Objective Grid-based applications. It allows users to visually set page breaks for their grid using mouse drag-and-drop. It also allows users to scale and fit their grids to a specified number of pages wide and tall. These two behaviors are controlled by the m_bFitToPages variable.

- **If m_bFitToPages is TRUE**, then the grid will be scaled to fit the specified number of pages wide and tall, using the SetPageGrid API. In other words, users will have control over how many pages to divide the grid into but will not have control over where the page break should be. Scaling will be done in this mode only when the specified page break dimensions are different from the inherent page break dimensions of the grid.

- **If m_bFitToPages is FALSE**, users will be able to use the mouse to drag the existing page break line to a new position and to add new page break lines. However, if users drag page break lines farther than a printed page apart, the grid will automatically insert new page break lines in between. In this mode, users get an approximate WYSIWYG effect in the printing printout. No grid scaling is done in this mode.

Please refer to the sample located at `<stingray-installdir>\Samples\Grid\General\PageBreakUI`. 
20.2 Adding Page Break and Scaled Printing Support

You can add a page break and/or scaled printing support to an Objective Grid-based application quickly and easily.

1. After you create an Objective Grid application, add a pointer to the `CGXPageBreakPlugin` class variable in your view class. Ensure that your view class is derived from the `CGXGridView` class.

```cpp
class CPageBreakUIView : public CGXGridView
{
public:
    CGXPageBreakPlugin* m_pPlugin;
    ...;
}
```

2. Create the plugin in the constructor for your view class.

```cpp
CPageBreakUIView::CPageBreakUIView()
{
    m_pPlugin = new CGXPageBreakPlugin;
    ...;
}
```

3. Destroy the plugin in the destructor method of the view:

```cpp
CPageBreakUIView::~CPageBreakUIView()
{
    delete m_pPlugin;
}
```

4. Call the `Plugin()` method of the plugin during the `OnInitialUpdate()` method of the view, before any other method calls:

```cpp
void CPageBreakUIView::OnInitialUpdate()
{
    if(m_pPlugin != NULL)
        m_pPlugin->Plugin(this);
    ...
}
```

5. Override the `OnBeginPrinting()` method of your view class. In the overriding method, you need to call the `DoBeginPrinting()` method and then call `CGXGridView::OnBeginPrinting()` if `DoBeginPrinting()` returns `FALSE`. The `DoBeginPrinting()` method prepares the mapping mode for the given `pDC`. After you edit this method, the device context is ready for use in printing or in the print preview.

```cpp
void CPageBreakUIView::OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo)
{
    BOOL bRet = FALSE;
    if( m_pPlugin != NULL )
        bRet = m_pPlugin->DoBeginPrinting(pDC, pInfo);
    if( !bRet )
        CGXGridView::OnBeginPrinting(pDC, pInfo);
}
```
6. Override the `OnGridPrepareDC()` method of `CGXGridView` class in your view class. In this method, you need to delegate the call to the `DoPrepareDC()` in our plugin.

```cpp
void CPageBreakUIView::OnGridPrepareDC(CDC* pDC, CPrintInfo* pInfo)
{
    if( m_pPlugin != NULL )
        m_pPlugin->DoPrepareDC(pDC, pInfo);
    else
        CGXGridView::OnGridPrepareDC(pDC, pInfo);
}
```

7. Override the `WindowProc()` method of your view class. Generally, this is how you add a plugin.

```cpp
LRESULT CPageBreakUIView::WindowProc(UINT message, WPARAM wParam, LPARAM lParam)
{
    LRESULT lResult;
    if(m_pPlugin != NULL)
    {
        m_pPlugin->HandleMessage(message, wParam, lParam, &lResult);
        if( m_pPlugin->m_bExitMessage )
            return lResult;
    }
    return CGXGridView::WindowProc(message, wParam, lParam);
}
```

8. In the `OnDraw()` method of your view class, add plugin drawing code similar to the following code. Generally, you should draw the page break line last, so the `DoDraw()` method of the plugin is called last.

```cpp
void CPageBreakUIView::OnDraw(CDC* pDC)
{
    CGXGridView::OnDraw(pDC);
    if( m_pPlugin != NULL )
        m_pPlugin->DoDraw(pDC);
}
```

9. You may need to add a menu item in your project to show or hide the page break lines.

```cpp
void CPageBreakUIView::OnChangePageBreakMode()
{
    if (m_pPlugin != NULL )
    {
        m_bIsPageBreakMode = !m_bIsPageBreakMode;
        m_pPlugin->SetPageBreakMode(m_bIsPageBreakMode);
    }
}
10. For scaled printing, you could provide a dialog to your users where they can specify their desired page break dimensions. You could then use the SetPageGrid API to scale the grid to the specified dimensions.

```cpp
void CPageBreakUIView::OnFitToPages()
{
    if( m_pPlugin != NULL )
    {
        CPageConfigDlg dlg;
        dlg.m_bFitToPages = m_pPlugin->m_bFitToPages;
        dlg.m_nColPages = m_pPlugin->m_nColPages;
        dlg.m_nRowPages = m_pPlugin->m_nRowPages;
        if( dlg.DoModal() == IDOK )
            m_pPlugin->SetPageGrid(dlg.m_nRowPages,
                                            dlg.m_nColPages, dlg.m_bFitToPages);
    }
}
```

11. To change the color of the page label text and page break lines, you can assign new COLOR-REF values to the following public members of CGXPageBreakPlugin:

- `m_clrLabel` (Color for the page label text.)
- `m_clrDefaultBreakLine` (Color for default break lines.)
- `m_clrCustomBreakLine` (Color for custom break lines.)

12. To change the font, size, or visibility of the page label text, override CGXPageBreakPlugin::DrawPageBreak(). Currently, the font is hardcoded and the visibility of the page labels and page break lines are toggled simultaneously when CGXPageBreakPlugin::SetPageBreakMode() is called.

13. For scaled printing, you could provide a dialog to your users where they can specify their desired page break dimensions. You could then use the SetPageGrid API to scale the grid to the specified dimensions.

```cpp
void CPageBreakUIView::OnFitToPages()
{
    if( m_pPlugin != NULL )
    {
        CPageConfigDlg dlg;
        dlg.m_bFitToPages = m_pPlugin->m_bFitToPages;
        dlg.m_nColPages = m_pPlugin->m_nColPages;
        dlg.m_nRowPages = m_pPlugin->m_nRowPages;
        if( dlg.DoModal() == IDOK )
            m_pPlugin->SetPageGrid(dlg.m_nRowPages,
                                            dlg.m_nColPages, dlg.m_bFitToPages);
    }
}
```
Database Engine Architecture

21.1 Introduction to Database Engine Architecture

The Database Engine Architecture is a new scalable, extensible framework that enables you to bind Objective Grid to different data sources. The details of database access are decoupled from the grid in a separate DBEngine class (a CGXDBEngine subclass). The browser-like layout and update logic are in a layout-manager class (CGXBrowserLayoutManager).

21.2 Database Engine Architecture

The three main classes comprising the Database Engine framework are CGXDBEGrid, CGXDBEngine, and CGXBrowserLayoutManager.

Figure 132 – Database Engine Class Architecture
NOTE: The dotted lines in the graphic above indicate that the class delegates functionality to another.

21.2.1 DBEngine

This architecture introduces the interface DBEngine (abstract base class, CGXDBEngine), which exposes an interface through which a grid can access any kind of datasource. You can bind an implementation of this abstract base class to a specific data access technology, such as Rogue Wave’s SourcePro DB (version 5.1).

The APIs exposed by the engine are divided into three groups: Initializing, Fetching, and Updating.

21.2.2 Initializing APIs

There is no initializing API in the base CGXDBEngine class, because the parameters required for initializing typically vary in count and type depending on how you are accessing the database. The concrete implementation of the CGXDBEngine class specifies the APIs for initializing. The concrete implementation provides one-time initialization through the constructors and/or special functions to initialize/re-initialize the engine. The engine can re-initialize in three different ways.

- Requery with the same parameters, usually to get the most current version of the underlying data.
- Requery with modified parameters but the result set would have the same schema.
- Re-initialize with different parameters, with the result set having a new schema. This amounts to closing the engine and re-opening it.

The concrete implementation needs to support one or more of the techniques for reinitializing the engine. The technique you use to re-initialize the engine affects the bound grid. If you requery the grid, it redraws to refresh its state and display. If you re-initialize the grid with different parameters, a new schema is created and InitFromEngine() is called to re-initialize the grid.

21.2.3 Fetching APIs

Most of the database driver APIs expose the result of a query in the form of records, so you can fetch the data in the records one at a time. Although this is suitable for data-processing applications, data-presentation applications like the grid prefer the resultant data in a tabular form. Accordingly, the CGXDBEngine interface exposes a table-like API.

```cpp
virtual BOOL GetValue(long nRecord, long nField, CString& strValue) = 0; // nRecord and nField are 0 based...

virtual long GetRecordCount() = 0; // 1 based...
virtual long GetFieldCount() = 0;    // 1 based...
```

Because the grid ultimately renders a string to the screen, it is optimal for GetValue() to return a string.
You need to fetch records for the grid incrementally. When you are working with large tables, you should fetch a minimal number of records initially and then fetch more records as necessary. Typically, more records need to be fetched when the user scrolls down to see more records. The CGXDBEngine exposes a convenient API to handle fetching records.

```cpp
// Enables incremental fetching...
virtual void FetchRecords(long nRecords) = 0;
```

A side effect of fetching incrementally is that we do not know if the count returned by GetRecordCount reflects the total record count or only the number of records fetched so far. The following API solves that problem.

```cpp
virtual BOOL IsEOFSeen();
```

### 21.2.4 Updating APIs

Unlike the fetch data, the grid likes to see update data as records rather than in tabular form. This is because the grid subscribes to the popular database editing technique of editing and updating one record at a time. The following update APIs allow such a technique.

```cpp
virtual BOOL Edit(long nRecord) = 0;
    // Call Edit before editing a record via SetValue...

virtual BOOL SetValue(long nRecord, long nField, CString* strValue);

virtual BOOL Update(CString* strError = NULL) = 0;
    // Complete the editing on a record by calling Update...

virtual long AddRow() = 0;
    // The returned value is the new record no. that should
    // be used in SetValue...

virtual BOOL DeleteRows(long nFrom, long nTo) = 0;
```

Calling `Edit()` begins an edit mode on a particular record. After a record is in edit mode, you can call `SetValue()` on that record only. After you finish editing the record, call `Update()` to store the changes inside the datasource. Complete the same steps to add a new row.

### 21.2.5 Miscellaneous APIs

There are also APIs that allow you to cancel a current edit mode, ensure that the engine is initialized, ensure that the result set can be updated, requery the result set, and more. For a comprehensive listing of the APIs exposed, see CGXDBEngine in the Objective Grid Class Reference.

### 21.2.6 Summary of Database Architecture

Essentially, the logic involved in communicating with the datasource is decoupled from the grid in an implementation of CGXDBEngine. You can easily bind the grid with your datasource by implementing the simple interface specified in the abstract CGXDBEngine.
# 21.3 Layout Manager

The layout manager, `CGXBrowserLayoutManager`, is a utility class for the database engine grid. It encapsulates the logic for initializing the grid with a browser-like layout, so you can take advantage of browser-like editing behavior (Edit/Modify/Update). It also communicates with the `DBEngine` to fetch and store data. With this utility class, you can easily customize the browser-like layout and the browser-like editing behavior by subclassing and overriding the virtuals.

## 21.3.1 Browser-Like Layout

The grid styles and properties that the layout manager modifies fall into two categories: datasource-independent and datasource-dependent.

The datasource-independent properties include an arrow row header control that indicates the current row with the edit mode and the append row at the bottom that allows you to add new records.

There are other properties that are dependent on the datasource and the result set’s schema, like the column name (based on the field name), column width (based on the field type), and the column base style (again based on the field type). The grid needs to determine certain styles for the column based on the corresponding field’s data type.

Sometimes you may encounter proprietary, nonstandard data types with different data access APIs. Ideally, the `DBEngine` would convert the proprietary data types from different data access APIs to a predefined data type enumeration that the layout manager understands. However, our classes do not change proprietary data types to a predefined data type enumeration, because this conversion is inefficient, restrictive, and not easily scalable.

Instead, the layout manager delegates conversion to a function with a pre-defined signature through a member-function pointer that you can initialize during run-time. The signature for this function is defined as `GXInitDBEGridFromEngine`. Each implementation of `CGXDBEngine` needs to have an implementation of the `GXInitDBEGridFromEngine` function signature to avoid translating proprietary data-types inside the `DBEngine` and to keep the layout manager abstract so it can work with all data access APIs.

## 21.3.2 Browser-Like Editing

The layout manager supplies the data to be displayed in the grid cells on demand and stores any changes the user makes to the datasource via the `DBEngine`. It uses the `DBEngine`’s edit mode to cache the user’s changes in the current row. When the user moves to a new row, it asks the `DBEngine` to store the changes into the datasource by calling `Update()`. `DBEngine` handles adding recording in the same fashion.

Note that the grid has a reference to the `DBEngine`. It always uses the layout manager to fetch/store data from the engine so it can remain independent of the data access logic.
21.3.3 Hierarchical Browser Layout Manager

The CGXHierGridLayoutManager template derives from the CGXBrowserLayoutManager template and implements the logic involved in initializing a CGXRegularGrid-type hierarchical grid.

It handles binding the child grids to the appropriate child engine by querying for it from the parent’s engine.

Use this kind of layout manager in conjunction with a hierarchical DBEGrid.

21.4 Database Engine Grid

The database engine grid (CGXDBEGrid) holds a reference to the CGXDBEngine and the CGXBrowserLayoutManager. It delegates all of its operations to the layout manager, which communicates with the DBEngine.

The CGXDBEGrid class also exposes a set of APIs that you can use directly on the datasource to query or modify it. See the CGXDBEGrid class members in the Objective Grid Class Reference. These APIs are based on the assumption that one database record corresponds to one row in the grid. This is the default layout set by the CGXBrowserLayoutManager. If you use a custom layout manager that wraps a database record between multiple rows, the meaning of the following functions may change or functions might become irrelevant. A better approach is to operate on the CGXDBEngine associated with the grid when you need to get/set data programmatically.

21.4.1 Hierarchical Database Engine Grid

The CGXHierDBEGrid derives from CGXDBEGrid and CGXRegularGrid in the following manner:

\[
\text{template <class GRID>}
\text{class CGXDBEHierGrid : public CGXDBEGrid < CGXRegularGrid<GRID> >}
\]

It holds a reference to the CGXHierGridLayoutManager and CGXDBEngine. Like CGXDBEGrid, this delegates all of its operations to the CGXHierGridLayoutManager, which initializes the grid.

21.4.2 Initializing the Grid with CGXDBEngine

You need to bind the database engine classes together. Consider the following initialization code:

```c++
void CGXDBEToolsGridView::OnInitialUpdate()
{
  ...
  // Create and initialize a concrete implemenatation of CGXDBEngine...
  CGXDBToolsEngine* engine = new CGXDBToolsEngine(aTable,
         updateCols, conn);
  // Grid takes a pointer to the abstract interface CGXDBEngine. 
  // whereas the object itself is a CGXDBToolsEngine an 
  // implementation of CGXDBEngine.
  SetEngine(m_pDBEngine);
  // Implementation(DBTools) specific initialization function.
```

Chapter 21   Database Engine Architecture  435
This code binds the grid to a DBEngine through the SetEngine() call. It also creates a new instance of the layout manager in the grid. Now you can make the layout manager's initialization function pointer point to the data access specific initialization function. The grid is bound to the DBEngine.

21.5 ADOEngines

21.5.1 CGXADOEngine

The CGXADOEngine implements CGXDBEngine and binds it to an ADORecordset.

Pass an ADORecordset via the constructor while creating the engine.

CGXADOEngine(ADORecordset* pSet, BOOL bIsOpen = TRUE);

There are also member-function variants for this constructor that you can use to re-initialize the engine at a later time.

void Init(ADORecordset* pSet, BOOL bIsOpen = TRUE);

If any of these re-initializing calls change the schema of the result set, then you will need to re-initialize the grid to reflect this new schema with a call to InitFromEngine().

21.5.2 CGXHierADOEngine

The CGXHierADOEngine derives from CGXADOEngine and creates a child during a call to GetChildAt(). It checks if there are any adChapter type fields available. If it finds any, it creates a child off the recordset to which it points. It also keeps the child recordset synchronized with the child engines. It is fairly complex, because all the children share the same underlying child recordset.

Use this engine when binding it to a hierarchical DBEGrid.

Please refer to the sample located at <stingray-installdir>\Samples\Grid\Database\AdoHier.
21.6 Database connectivity with Objective Grid

21.6.1 Microsoft Access

The Objective Grid distribution includes Microsoft Access .mdb files for testing data binding with ODBC and ADO in the samples built with Win32.

21.6.1.1 Enabling ODBC Source on x64

On an x64 machine, you have to use the 32-bit version of ODBC Data Source Administrator, located at \C:\Windows\SysWOW64\odbcad32.exe, to set the data sources.

21.6.2 Microsoft SQL Server

The Grid samples for both Win32 and x64 provide for connectivity to SQL Server. To test the ODBC samples, you need to set the data source with an available SQL Server driver. We recommend installation of Microsoft’s standard sample database, Northwind.

To test the ADO samples, download and install SQL Server x64 drivers on the test machine. You may obtain Microsoft’s SQL Server updates from http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=ceb4346f-657f-4d28-83f5-aae0c583d52.

For testing ADO with SQL Server, you can override CreateRecordSet in the Grid sample ADO-Query as follows:

In adoquvw.h, add the following function prototype:

```cpp
CComPtr<ADORecordset> CAdoqueryView::CreateRecordset(CComBSTR bstrSource, CComBSTR bstrSQL);
```

In adoquvw.cpp, add the following function implementation:

```cpp
CComPtr<ADORecordset> CAdoqueryView::CreateRecordset(CComBSTR bstrSource, CComBSTR bstrSQL)
{
    bstrSource = "Provider=SQLNCLI10; Data Source=<data source name>; InitialCatalog=Northwind; User Id=<user ID>; Password=<password>";
    return CMyAdoRecordView::CreateRecordset(bstrSource, bstrSQL);
}
```

When using the sample Northwind database, use the "Select* from Customers" query.

21.6.3 Oracle

For testing ODBC samples built with Win32, set DSN with Microsoft ODBC for Oracle. To test ADO samples from an x86 machine, be sure msdaora.dll is installed. There are two options available: connection through ODBC or directly.

1. Use following connection string to connect to ODBC through ADO:
bstrSource = "Driver={Microsoft ODBC for Oracle};
DSN = <data source name>; Uid=<user ID>; Pwd=<password>*;"

2. For a direct connection through ADO, use the following connection string:

bstrSource = "Provider=msdaora; Data Source=<data source name>;
User Id=<user ID>; Password=<password>*;"

Then, use the following override:

```
CComPtr<ADODAORecordset> CAdoqueryView::CreateRecordset(CComBSTR bstrSource,
CComBSTR bstrSQL)
{
    HRESULT hr;
    COleVariant vNull;
    CComPtr<ADODAORecordset> piRecordSet;
    CComPtr<ADODAOConnection> piConnection;
    //Select a source
    bstrSource = "Driver={Microsoft ODBC for Oracle}; DSN = OraTest;
    Uid=dbtest1;Pwd=zebco5";
    //bstrSource = "Provider=msdaora; Data Source=kettle; User Id=dbtest1;
    //Password=zebco5";
    hr = CoInitialize(NULL);
    if (FAILED(hr))
    {
        AfxMessageBox(_T("Unable to Initialize COM"));
        goto ErrorExit;
    }
    //Intialize Connection
    hr = CoCreateInstance(CLSID_CADOConnection, NULL, CLSCTX_INPROC_SERVER,
    IID_IADOConnection, (LPVOID *)&piConnection);
    if(FAILED(hr))
    {
        AfxMessageBox(_T("Unable to create connection object"));
        goto ErrorExit;
    }
    if (bstrSource == NULL || bstrSQL == NULL)
    {
        AfxMessageBox(_T("Invalid(NULL) Source name or SQL string"));
        goto ErrorExit;
    }
    //Initialize Recordset
    hr = CoCreateInstance(CLSID_CADORecordset, NULL, CLSCTX_INPROC_SERVER,
    IID_IADORecordset, (LPVOID *)&piRecordSet);
    if(FAILED(hr))
    {
        AfxMessageBox(_T("Unable to create Recordset"));
        goto ErrorExit;
    }
    BOOL bSuccess;
    hr = piConnection->Open( bstrSource, NULL, NULL, -1 );
    if (FAILED(hr))
    {
        AfxMessageBox(_T("Unable to make connection to Source: ")
```
21.6.4 Re-Enabling DAO in Objective Grid

Although Stingray Studio dropped support for DAO when that support was dropped by Micro-
soft, the DAO code was not dropped from the code base. So you can still use DAO by making the
Objective Grid Build Wizard aware of this code. For information on how to do this, please see this
article in the Stingray Forums: http://forums.roguewave.com/showthread.php?136-
Enabling-DAO-in-OG-10.4.
Chapter 22

Grid XML Serialization Architecture

22.1 Overview

The Objective Grid XML Serialization Architecture builds upon the core Stingray XML Serialization Architecture to allow persistence of Objective Grid’s data components in XML format. Please refer to the Stingray Foundation Library XML Serialization Architecture documentation (in sflug.chm or sflug.pdf) before reading through this chapter.

You can write and read Objective Grid as XML and convert its XML format into any other format (or vice versa) using XSL style sheets. We encourage you to take a look at Objective Grid’s XML format by viewing the serialized file via an XML file viewer like Internet Explorer. Use one of the samples mentioned in this guide to create an XML file from Objective Grid.

If you are using Microsoft Visual Studio, the latest version of the Microsoft® Platform SDK is a requirement for building the Stingray Studio libraries with support for XML serialization functionality. In order to properly build XML support as part of the Objective Grid library or the Objective Chart library, our XML classes require you to install the latest version of the Platform SDK on your system. Moreover, the Platform SDK header files must precede any other include files and/or lib files in your development environment path. We tested our components against MSXML.dll version 6.10.1200.0 (MSXML 6.0 SP1). Later versions of this DLL will also work. You can download the Platform SDK from:


22.2 Extensions

22.2.1 Formatters for Objective Grid Components

In order to participate in the XML Serialization Architecture we provide pre-built formatters corresponding to each and every data component in Objective Grid. For example, CGXGridParamFTR and CGXStyleFTR will allow you to serialize CGXGridParam and CGXStyle objects, respectively.
22.2.2 Tag Names for Data Components

To facilitate internationalization, the pre-defined tag names corresponding to each data component are stored in a separate resource file and will be automatically included in your build. All the formatters allow you to optionally specify a custom tag name in their constructors.

22.2.3 Selective Linking via Build-Wizard

For optimal library size, Objective Grid’s build-wizard allows you to include or ignore this feature when you build the Objective Grid libraries.

22.3 Usage Example

To start using this feature in your Objective Grid applications, you have to modify your document class to make it derive from `SECXMLDocAdapter_T` and override the `XMLSerialize()` virtual. From inside the overridden `XMLSerialize()` virtual you can simply use the pre-built formatters to serialize all or some of the Objective Grid data components. For example, to serialize the whole grid, the override will look like this:

```c++
void CGridSampleDoc::XMLSerialize(SECXMLArchive& ar)
{
    if(ar.IsStoring())
    {
        CGXGridParam* pParam = (CGXGridParam*)GetParamInYourDocument();
        ar.Write(NULL, CGXGridParamFTR(pParam));
    }
    else
    {
        CGXGridParam* pParam = NULL;
        ar.Read(NULL, CGXGridParamFTR(pParam));
        BindParamToYourGrid(pParam);
    }
}
```

22.4 Sample

For more information on this feature, take a look at our sample at `<stingray-installdir>\Samples\Grid\General\Gridapp`. 
Chapter 23

XML Read/Write Library

23.1 Introduction

XML is fast becoming the data interchange dialect of the Internet. When a common XML format has been agreed upon for the data to be exchanged, this data can then be interchanged between systems that run on disparate hardware and software. Some technical articles have heralded XML as the ASCII of the internet. The reasons for such a parallel are readily visible. ASCII permits the interchange of vast amounts of information between completely different systems. E-mail systems that link the digital universe are a prime example for such exchange. XML would permit such exchange between software systems of various hues. Enough about the importance of XML; let us proceed to discuss the actual purpose and use of the Stingray XML library that comes with Objective Grid.

23.2 Library Purpose

Microsoft provides an XML parser with IE 5.0 (and provides one as part of the operating system with Windows 2000). We can use this parser to parse an XML file. Thereafter, nodes in the XML document are readily available through a rich object model that mirrors the underlying data. This object model is usable from inside any ActiveX environment. We can readily write code that can read and write an XML file using this object model.

As with any hierarchical data, it is of paramount importance that data written out or read in is viewed in context. For example, consider the following XML data:

```xml
<transaction>
  <item_name>echo</item_name>
  <price>10000</price>
</transaction>
```

The price is valid only in the context of the transaction node. This is a very simple arrangement but when writing a complex schema, is prone to the introduction of several errors. When performing a write (or read), our code has to correctly remember the parent node (the nesting level) and so on. When XML read/write code is implemented in this manner, it is very difficult to maintain, as the data access code is intertwined with the XML read/write code.
One solution (actually the solution that we chose) that would help remedy some of this would be to have a parsed representation of an XML schema (or DTD) that can be processed by a system and then written out or read in with appropriate context information. The system would look somewhat like this:

Figure 133 – XML Dataflow

```
<table>
<thead>
<tr>
<th>DTD/Schema</th>
<th>Parsed Representation</th>
<th>Processor</th>
<th>XML file</th>
</tr>
</thead>
</table>
```

The above representation can be reversed to perform a read.

Some advantages of using an approach like the above are:

- The schema is effectively separated from the data. In other words, the code that performs the data access is logically separated from the code that is responsible for creating and reading an XML file. The two essential parts of this operation are thus separated. This allows for better maintenance.

- It allows for one section of the code to be written independent of the other. When you can have the parsed representation of the schema available, the data itself could be populated from a variety of sources, each with different data access methods. The resultant XML would be the same and the code that deals with the schema would also remain the same. You can easily add other data sources as required.

We used this logical design as the basis for the implementation of our XML library. There is an XML map structure that is basically a parsed version of a schema/DTD. This is coupled with what we call a resolver object (this is nothing but a feedback interface for data gathering and feedback). Now that we have covered the basic logical design, we can look at some of the implementation details of this framework.
23.3 Implementation

Please refer to the samples xml_grid and xml_samp, available from the Rogue Wave Web site, as explained in Section 3.6.1, “Location of Sample Code,” in the Stingray Studio Getting Started Guide.

The XML map which is the parsed version of the XML schema/DTD is represented using the following data structure:

class XML_ENTRY_MAP
{
    ...

    /* The name of the node.*/
    LPCTSTR pszName;   // 1

    /* Unique id assigned to this node. When an element appears in more
    than one part of a document, each occurrence should usually have
    a different relative id.*/
    ENTRYID dwID; // 2

    /* The type of the node. Please refer to DOMNode type in the MSDN
    documentation.*/
    DOMNodeType entryType; // 3

    /*Qualifying namespace if any.*/
    LPCTSTR pszNameSpace; // 4

    /*List that holds nested nodes.*/
    CXMLList listXMLMapEntries_p; // 5
};

This structure has all the essential elements that are needed to represent a node in the XML document. These are:

//1 The node name (the pszName member): This is the name of the XML node.

//2 The unique node ID (the dwID member): This is an ID that is not known to the XML file as such. It is only used internally by us when writing and reading the file. This node id has to be unique even when an element appears in more than one part of a document. This id enables us to tie together that node with a piece of data when processing.

//3 The DOMNodeType (the entryType member): This member defines the type of node. Please refer to the MSDN documentation on DOMNodeType for further information.

//4 Qualifying namespace, if any (the pszNameSpace member): This member specifies the qualifying namespace. This is optional.

//5 XML_ENTRY_MAP list for nested nodes (the listXMLMapEntries_p member): This member holds all child nodes for this node. Each of the child nodes is in turn described by an XML_ENTRY_MAP class, thus allowing us to represent an XML tree of any depth and complexity.

When the processing system reads this data structure, it can loop through the nodes and, depending on whether an XML read or write operation is taking place, call feedback interfaces. The purpose of these feedback interfaces varies between the read and write operation. Let us look at IXMLRResolver, the read feedback interface, first.
The *IXMLRResolver* interface is the primary interface that is used by the XML read code that processes *XML_ENTRY_MAP* structures when reading XML data streams.

Given below is the interface specification for this interface.

```cpp
class IXMLRResolver
{
...
//@cmember
/* Will be called when the system starts to read data from the
XML data stream.*/
virtual void OnStartRead() = 0;

//@cmember
/* Will be called when the system completes reading from the
stream*/
virtual void OnCompleteRead() = 0;

//@cmember
/* Will be called when read is started for a particular
node type. */
virtual bool OnStartReadLoop(const XML_ENTRY_MAP* pEntry) = 0;

//@cmember
/* Will be called to give the value of a node (as the nodes are
read out) */
virtual bool OnSetValue(const XML_ENTRY_MAP* pEntry,
    MSXML::IXMLDOMNode* pThisNode, XMLVALUE& xmlValue) = 0;
};
```

When `read()` starts, the system calls `OnStartRead()`, and when read ends, the system calls `OnCompleteRead()`. This is pretty straightforward. You can perform any one-time initialization (and cleanup) when these calls are made. You usually do not have to implement these and the other `IXMLRResolver` methods, as they are implemented by macros which we shall see later.

`OnStartReadLoop()` requires a little explanation. This is called the first time a node with a particular ID is read. For example, consider the following XML snippet:

```xml
<address>
  <street>Pine drive</street>
  <apt>432</apt>
</address>
<address>
  <street>Oak drive</street>
  <apt>32</apt>
</address>
```

When this is read, if there is an handler for `street` (identified to the system by the unique id, say, `idStreet`), then `OnStartReadLoop()` will be called when this node is hit for the first time. This is useful if you have to perform any initialization to be able to read this data. When the actual element is read, `OnSetValue()` is called. This has parameters (the XML_ENTRY_MAP value and the node pointer) that identify the node in question. The value itself is stored in an XML_VALUE parameter. This can be used to initialize your data structures.
The write feedback interface is very similar.

class IXMLWResolver
{
    ...

    // write feedback interface
    // @access public
    /* Will be called when write is started */
    virtual void OnStartWrite() = 0;

    /* Will be called when write is completed */
    virtual void OnCompleteWrite() = 0;

    /* Will be called when a value needs to be supplied (just before the write operation for a node takes place) */
    virtual bool OnGetValue(const XML_ENTRY_MAP* pEntry, MSXML::IXMLDOMNode* pParentNode, XMLVALUE& xmlValue) = 0;

    /* Will be called when a node is to be written out for the first time. */
    virtual bool OnStartWriteLoop(const XML_ENTRY_MAP* pEntry) = 0;

    // will be called when to check whether write needs to be continued on a node type
    virtual bool OnCanContinueWriteLoop(const XML_ENTRY_MAP* pEntry, MSXML::IXMLDOMNode* pNodeAdded) = 0;

    // will be called when writing is stopped for a particular node type
    virtual bool OnEndWriteLoop(const XML_ENTRY_MAP* pEntry) = 0;
};

There are some differences from IXMLWResolver. OnGetValue() will be called to retrieve the value for the node that is being written. OnStartWriteLoop() will be called when the node is written for the first time (same as when the node is read for the first time). However, if this function returns true, it will be repeatedly called.

In addition, OnEndWriteLoop() will be called when the write process is complete for a loop. Also OnCanContinueLoop() will be called to check whether there is additional data to be written out. You can use this to control how many node instances get written out for a given ID. For example if you have a linked list of addresses, you could return false from this function when you have reached the end of the linked list in order to write the whole list.

Populating these structures can be automated in two ways:

- Using macros
- Using tools that can generate code based on input parameters

We have followed both these approaches in order to make development with the Stingray XML library easier.
23.4 Macros and Tools Ease Development

There are two sets of macros. One set of macros populate the XML_ENTRY_MAP structures. With these macros the most important issue is that the nesting of these has to be just right. Shown below is a set of these macros that describe an XML schema to store addresses.

```c
BEGIN_XML_MAP(CAddressXMLMap)
    XML_ROOT_ENTRY()
    SEC_XML_NEST()
    SEC_XML_START()
    XML_ELEMENT_ENTRY(ADDRESSES, eAddresses)
    SEC_XML_NEST()
    SEC_XML_START()
    XML_ELEMENT_ENTRY(ADDRESS, eAddress)
    SEC_XML_NEST()
    WRAP(XML_TEXT_ENTRY(FIRST, eFirst))
    SEC_XML_START()
    XML_TEXT_ENTRY_NS(s:LAST, eLast, _T("stingray.com"))
    SEC_XML_NEST()
    WRAP(XML_ATTRIBUTE_ENTRY(SALUTATION, eSalutation))
    END_SEC_XML_NEST()
    SEC_XML_END()
    END_SEC_XML_NEST()
    WRAP(XML_TEXT_ENTRY_NS(s:OFFICECODE, eOfficeCode, _T("stingray.com"))
    SEC_XML_NEST()
    SEC_XML_START()
    XML_TEXT_ENTRY(EMAIL, eEmail)
    SEC_XML_NEST()
    WRAP(XML_COMMENT_ENTRY(EMAILCOMMENT, eEmailComment))
    END_SEC_XML_NEST()
    SEC_XML_END()
END_XML_MAP()
```

Some important points to be noted:

- All XML_MAP definitions have to start with XML_ROOT_ENTRY(). This cannot be changed in this version. If you wish to have additional data written as part of the root you can add these on completion of the write operation.

- When a node has child nodes, these have to be listed within SEC_XML_NEST() and END_SEC_XML_NEST() macros.

- When a node has no children, then it can be completely wrapped within the WRAP(...) macro.

- When a node does have children, it has to be bounded by SEC_XML_START() and SEC_XML_END() macros.
While composing the XML_MAP using this set of macros is not hard, it is quite tedious. Just the kind of work that begs for automation. We supply a tool (xml_tool.exe, located in the bin folder of your Objective Grid installation) that automates this step. This tool takes a valid schema and generates this map, as well as some other defines that are essential, notably the unique defines and an enum.

Writing the second map, the WRITE_MAP, is a lot easier. This has no hierarchy. A sample map is shown below. Each unique id is mapped to a handler function. Internally these macros map the resolver interfaces to this handler. Each of the resolver call backs maps to a different context value when the handler is called.

Please refer to the help file discussion of eTypeRWContext for a description of the various context values.

The resolver that is used along with the macros is to be derived from IXMLRWResolver and CentryValueHolder.

### 23.4.1 Sample WRITE_MAP

```c
BEGIN_WRITE_MAP(CAddressResolver)
    WRITE_MAP_ENTRY(eAddresses, AddressesHandler)
    WRITE_MAP_ENTRY(eAddress, AddressHandler)
    WRITE_MAP_ENTRY(eFirst, FirstHandler)
    WRITE_MAP_ENTRY(eLast, LastHandler)
    WRITE_MAP_ENTRY(eSalutation, SalutationHandler)
    WRITE_MAP_ENTRY(eOfficeCode, OfficeCodeHandler)
    WRITE_MAP_ENTRY(ePhone, PhoneHandler)
    WRITE_MAP_ENTRY(eExt, ExtHandler)
    WRITE_MAP_ENTRY(eEmail, MailHandler)
    WRITE_MAP_ENTRY(eEmailComment, eMailCommentHandler)
END_WRITE_MAP()
```

This map is not generated by the tool. This has to be written by hand. Each handler is defined as:

```c
void Handler(XML_ENTRY_PARAM& pParam, eTypeRWContext context, bool& bRetval, XMLVALUE& value)
```

XML_ENTRY_PARAM is just a wrapper around XM_ENTRY_MAP and other essential data. Refer to the help file for information on eTypeRWContext and XMLVALUE. Looking at the resolver interfaces you can easily map to one of the eTypeRWContext values.
23.5 Usage

Once both maps are defined, using the code to read and write files is quite simple. Shown here is a typical write operation. A read operation is very similar, but requires an additional step, a call to pResolver->InitializeForRead().

```cpp
CAddressResolver* pResolver = new CAddressResolver(&addressLst, &phoneMap, &extMap, &mailMap);
pResolver->Initialize();

// map
CAddressXMLMap map;
map.Initialize();
map.SetResolver(pResolver);

// file
CXMLFileIO address;
address.SetXMLWrite(&map);
address.SetFileName(_T("address.xml"));
address.Write();
```

One note about passing context information to the resolver. Information that is useful to the whole resolver should be added as part of the derived resolver. Information that is specific to nodes should be assigned to these nodes with `CAddressResolver` (or any derived resolver) :: SetContext. Remember that this information also applies to all nodes that will be written with that ID and not to any unique node.

If you are using Microsoft Visual Studio, the latest version of the Microsoft® Platform SDK is a requirement for building the Stingray Studio libraries with support for XML serialization functionality. In order to properly build XML support as part of the Objective Grid library or the Objective Chart library, our XML classes require you to install the latest version of the Platform SDK on your system. Moreover, the Platform SDK header files must precede any other include files and/or lib files in your development environment path. We tested our components against MSXML.dll version 6.10.1200.0 (MSXML 6.0 SP1). Later versions of this DLL will also work. You can download the Platform SDK from:

24.1 Introduction

The hierarchical grid architecture included in Objective Grid allows you to establish a one-to-many parent-child relationship between grids. This architecture enables you to expand/collapse each parent row so you can view/hide the row’s associated children in a tree-like fashion.

Figure 134 – Example Hierarchy Grid

These grids derive from the CGXRegularGrid template, as the class diagram below illustrates, and are called Regular grids, discussed in Section 24.2. Although they are not related functionally, the hierarchical grid implementation also implements two new types of grids: the Wrapped grid, dis-
discussed in Section 24.3, and the Multiple-row-type grid, discussed in Section 24.4. If you do not want these types of grids to exhibit the feature of the hierarchical grid, you can configure them so they do not.

Figure 135 – Hierarchy Grid Class Diagram

Hierarchical grid features:

- A grid parent and child can be instances of different types.
- Alternatively, if you specify a particular class type for the child, a new instance of that type is created for each child. You would typically populate such a child grid in the virtual mode, based on the corresponding parent row.
- You can explicitly bind a specific instance of a child grid to a parent row.
- Seamless current cell movement between the parent and the child.
- Automatic rearrangement of the children when the parent rows are moved.
- You can turn on a common child-grid header just below the parent header.

Note that the three grid types provide alternate APIs for certain common CGXGridCore APIs. For example, the Regular Grid type provides a SetLogRowCount/SetLogColCount. You should use this special API, rather than the standard SetRowCount/SetColCount, because the Regular grid implementation includes an expander column (with a plus-minus button) that adds to the logical columns in the underlying grid and also inserts one additional row for each logical row in the parent to host a child. The absolute rows/cols differ from logical rows/cols in each of the grid types. It is important to recall this distinction when you are calling the CGXGridCore functions directly.

Please refer to the sample located at <stringray-installdir>\Samples\Grid\HierarchicalGrid.
24.1.1 Absolute Logical Coordinate Transformation

The underlying implementation uses covered cells in the parent grid to host a child and to manage the unique column count in each row. Accordingly, the absolute cell coordinates of a logical cell are not apparent, and one should always use the APIs below for such transformations.

GetAbsCellFromLogCell() / GetLogCellFromAbsCell()

Converts the logical cell coordinates to absolute cell coordinates, and the reverse.

GetAbsCellContainingChild()

Returns the absolute cell coordinates where a child grid is hosted.

24.2 Regular Grid

A Regular Grid is a hierarchical grid with a look-and-feel similar to that of any other grid derived from \texttt{CGXGridCore}.

Figure 136 – Regular Hierarchical Grid

To enable this functionality, you need to derive your window/view class from the \texttt{CGXRegularGrid} template. See Figure 135 for more information.
For example:

```cpp
class CRegularGridView : public CGXRegularGrid<CGXGridView>
```

Besides providing all the hierarchical grid interfaces, the regular grid also provides the following interfaces:

- Row/Column Count
- `SetLogRowCount()`/`GetLogRowCount()`
- `SetLogColCount()`/`GetLogColCount()`
- Set and get the logical row/column count

In all functions defined in `CGXGridCore`, the row/column parameters denote the absolute coordinates. You need to use the transformation APIs in Hierarchy Grid to convert them to logical coordinates and vice-versa.
24.3 Wrapped Grid

A wrapped grid wraps a logical row between multiple absolute rows. You can assign each logical column a unique width. You can set the row breaks after any logical column.

This is useful when you want to restrict the horizontal width of the grid to avoid horizontal scrolling. The wrapped logical row binds the logical columns with better affinity and better readability than if it were a single long row.

Figure 137 – A Wrapped Grid

Although not functionally related, the wrapped grid is also a hierarchical grid, and you can turn off the hierarchical functionality by configuring it.

To enable this functionality, derive your wnd/view class from the `CGXWrappedGrid` template.

For example:

```cpp
class CWrappedGridView : public CGXWrappedGrid<CGXGridView>
```

See Figure 135 for more information. Section 24.1.1, “Absolute Logical Coordinate Transformation,” describes how to convert between logical and absolute cell coordinates.

Users can visually resize the column width of each column and also rearrange the column positions and row-breaks by selecting and dragging the column headers.
24.3.1 Seamless Current Cell Movement Between Wrapped Rows

The wrapped grid implementation uses covered cells, so the user can change the width of individual cells. Each logical cell width is a multiple of the underlying absolute cell’s width. If the absolute cell’s width were 5 pixels (the default), then the logical cell’s width could only be a multiple of 5. You can reduce the default absolute cell width with `SetColWidthPrecision()`.

24.3.2 Row/Column Layout Interfaces

`SetWGColCount()`/`SetWGRowCount()`

Sets the logical row/column count.

`SetWGRowBreaks()`

Sets row breaks after one or more columns within the logical row.

`SetWGColWidth()`

Sets the column width of one or more logical columns.

`ResizeWGColWidthsToFit()`/`ResizeWGRowHeightsToFit()`

Resizes the column and row to fit the values in the corresponding cells.

`InsertWGCols()`/`InsertWGRows()`

Inserts one or more cols/rows after a particular col/row.

`RemoveWGCols()`/`RemoveWGRows()`

Removes one or more cols/rows after a particular col/row.

`MoveWGRows()`/`MoveWGCols()`

Moves one or more cols/rows after a particular col/row.

`SetWGColBaseStyle()`/`SetWGRowBaseStyle()`

Sets the column/row base style of a logical column/row.

In all functions defined in CGXGridCore, the row/column parameters denote the absolute coordinates. You should use the transformation APIs in the Hierarchy Grid to convert absolute coordinates to logical coordinates and the reverse.
24.4 Multiple Row Type Grid

The multiple Row Type Grid (MRT Grid) allows you to specify a different column count for each row. Each column in each row can take a unique column width.

This is useful for displaying a sequence of logically unrelated rows.

Figure 138 – Multiple Row Type Grid

The MRT grid is also a hierarchical grid. You can turn off the hierarchical functionality by configuring it.

To create a multiple row type grid, you need to derive your wnd/view class from the CGXMRT-Grid template.

For example:

```cpp
class CMRTGridView : public CGXMRTGrid<CGXGridView>
```

See Figure 135 for more information. Section 24.1.1, “Absolute Logical Coordinate Transformation,” describes how to convert between logical and absolute cell coordinates.
24.4.1 Visually Resizing Columns

The MRT implementation uses covered cells so the user can change the width of individual cells. Each logical cell width is a multiple of the underlying absolute cell’s width. If the absolute cell’s width were 5 pixels (the default), then the logical cell’s width could only be a multiple of 5. You can reduce the default absolute cell width with `SetColWidthPrecision()`.

24.4.2 Row/Column Count API

`SetMRGCOnCountInRow()`

Set the column count in a particular row. If the specified row is greater than the current row count, it automatically increases the current row count. There is no separate API to set the row count.
The ATL and Objective Grid

25.1 Introduction

Customers often ask whether Objective Grid can be used with ATL. Our answer has been that Objective Grid, like any other MFC library, can be used within an ATL application with MFC support. However, there are minor areas that need to be addressed to enable easy and effective use of Objective Grid inside of ATL applications. None of these issues are really specific to Objective Grid. In fact, you will run into these issues if you write and reuse a simple MFC extension DLL.

25.2 Wrapping Objective Grid In an ATL ActiveX Control

ATL is very powerful when it comes to implementing COM functionality. It is very easy to implement dual interfaces. MFC, on the other hand, has somewhat convoluted COM support. It starts out being very easy to use, but more often than not, it ends up being rather difficult to maintain and extend. Also, dual interfaces are somewhat more difficult to implement with MFC. ATL is therefore usually the first choice when it comes to writing an ActiveX wrapper for any C++ code.

Combining ATL with a rich MFC UI library like Objective Grid can produce an ActiveX control that is reusable, extensible, and customizable. We can easily link statically to MFC, thus eliminating the need for redistributing the MFC DLL. (MFC ActiveX controls cannot be linked statically to the MFC.) One issue is that these controls will be somewhat larger than controls that are written in native Win32 without MFC. However, with the use of advanced optimization techniques and just plain clever linking, it is possible to have a lean control. Using Objective Grid’s optimization techniques it is possible to create a very basic grid ActiveX control, with ATL used for the COM implementation, for under 600k. Remember that this control will have no dependency on MFC run times. Packaged in a CAB file it will be even smaller. Such a control will also be much more maintainable in the long run when compared to a native Win32 control. And, of course, consider all the time that will be saved when you reuse a powerful control such as Objective Grid instead of writing a Win32 based control from scratch.
Having laid out the case for the usefulness of Objective Grid ATL ActiveX controls, let us look into implementation issues. The most common way to use MFC controls inside of ATL ActiveX controls is to create the MFC control window as a child of the ATL control window. This approach is useful if the MFC control is a small part of the control UI and functionality. In this section, we are more concerned with ActiveX controls that expose MFC based controls, with the MFC control forming the dominant part of the UI and functionality. If this is the case, then the cited approach is not efficient. That approach results in the creation of two windows (one by the ATL control and one by the embedded control). Having a parent window that is essentially unused is a significant drawback, and creates several problems. Though these problems can be worked around, it is desirable to have one control window.

We have come up with an easy approach that allows the usage of one window in cases such as these. This solution basically involves chaining together the MFC message maps (in turn the MFC window procedure) and the ATL message maps. We have implemented a base class that does much of this work. We have also included an ATL object wizard that can be used to generate these controls easily. You can find this wizard under the Stingray category. After you have inserted an Objective Grid control, you can easily add methods and properties using the ATL wizards. With this wizard, writing an ATL based Objective Grid ActiveX control is very easy. The generated control exposes a few predefined properties and methods that you can remove as required. These are provided as sample code.

In use, you will only need to refer to the CGXGridWnd (or derived class) instance that is defined in the base class, m_wndClassImpl.

**Figure 139 – The Objective Grid ActiveX control Object Wizard**
25.3 Using Objective Grid Inside ATL Applications

Using Objective Grid inside ATL applications poses no problems if MFC support is enabled for the application, although there are state-related issues that are addressed in Section 25.4. Without MFC support, Objective Grid cannot be linked in as an extension library or as a static lib. It is possible to wrap Objective Grid in a regular DLL, and then use this DLL from an ATL/Win32 application without MFC support.

25.4 Module State Issues

The Objective Grid libraries require initialization before they can be used. During this process, resources are allocated and made available to the module that uses them. It is important that any such resources be available only to the module and not to the application. If such resources were to live in the application then several conflicts could arise.

Consider a case where two ATL-based DLLs link in Objective Grid. Assume that the first performs control registration. The second is then loaded. Both work fine. Then let us assume that the first control gets terminated, while the rest of the application continues to run. Like any good control, the first control cleans up after itself, un-registering the class. When the second control tries to create a control of this class, it fails.

Objective Grid is completely aware of these issues and can be used freely inside different ATL modules. Remember to call AFX_MANAGE_STATE(AfxGetStaticModuleState()) when you export functions that will be called from other modules. Non-module state aware MFC controls will fail under these situations. For more details on these issues, please see Section 15.1, “Objective Grid DLL Architecture.”

In summary, we have provided guidelines and working code, as well as code generation helpers, for better ATL compatibility with Objective Grid. Please contact us if there are issues with this support, or if there are other features that you would like to see implemented.
26.1 Introduction

Objective Grid provides two implementations for Grid-Excel interaction:

1. Reading and writing of Microsoft Excel files to the Stingray Objective Grid class library.
   This feature is enabled with ExcelReadWrite option in Grid Build Wizard. This approach works with BIFF 8 format of Excel files and does not work with Unicode.

2. Excel Automation for integration with Grid.
   This implementation is independent from the Excel read/write option and, unlike that option, also supports Unicode.

 WARNING: To use this option with Unicode, ensure that the Excel read/write option is unchecked in the Grid Build Wizard.

In addition, it works with the newest versions of Excel, implements more cell formats, and can be easily extended to support even additional formats. Further, run-time data exchange between Grid and Excel is supported. Note that this approach requires that you have Excel installed on your system.
26.2 Structure of a BIFF 8 file

In this section we will describe the basic structure of a BIFF 8 file. For additional information on the complete BIFF file format please refer to either msdn.microsoft.com or to the MSDN library CD.

26.2.1 Basic Structure of a BIFF File

The basic structure of a BIFF file is shown below:

Start Global data
  Font table
  Color table
  Format table
  Style table
  ...
End Global data

Start Worksheet data
  Data that is specific to the sheet goes here.
  This data often refers to the global table.
  Styles, for example, are stored in the global table and are referenced here by index.
End Worksheet data

Start Next Worksheet data
  ...
End Next Worksheet data

This is the structure of the documented stream. Excel has other streams. Some are not documented and some (like summary information) are. For our purposes we will only handle the data in the main stream, the WorkBook Stream.

Shown below is a typical Excel file (written out by Objective Grid) as seen in BiffView, a utility that ships with the Excel Software Development Kit (SDK). The sequence of global and local records are detailed above. Note that the global records are grouped together and yet are discrete. This makes them conceptually similar to a table that is accessible by index.
The structure of each actual record is quite simple. The first part is always a record identifier.

26.2.2 Record Identifier

With Excel (BIFF 8) the record identifier is 2 bytes (a word). The different fields that are read and written by Excel are detailed in the Excel SDK. Each of these is denoted by a 2 byte constant.
26.2.3 Record Length

The record length varies from record to record. This value is the size of the total record less the size of the fixed header for each record. The fixed header is the sum of the record identifier size and the record length size, i.e. 4 bytes with BIFF 8.

For example, let us take the case of a sample record, the BOF (beginning of file) record, defined in the Excel SDK as follows:

The BOF record marks the beginning of the Book stream in the BIFF file. It also marks the beginning of record groups (or “substreams” of the Book stream) for sheets in the workbook. For BIFF2 through BIFF4, you can determine the BIFF version from the high-order byte of the record number field, as shown in the following table. For BIFF5, BIFF7, and BIFF8 you must use the vers field at offset 4 to determine the BIFF version.

The SDK documentation describes each of the constituent parts of the Excel SDK in detail. For our purposes we will examine the binary dump of this field from a typical Excel File. This is shown below:

01F14: [BOF] (9h 9dec) Binary dump of BOF field from an Excel file

09 08 10 00 00 06 10 00 d3 10 cc 07 c9 00 00 00
06 00 00 00

To study the general structure, we are interested in only the first four bytes. In the above case these are: 09 08 10 00

The first two bytes in this byte sequence stand for the record type (in this case typeBOF). According to the Excel SDK, we copy these declarations (for the fields that we use) into gxexhdr.h:

#define typeBOF 0x0809

This is what we have considering that the byte sequence is reversed on Little Endian machines. The first two bytes are typeBOF.

Now let us look at the next two bytes. Taking byte swapping into account we have 0x0010 or in decimals 16 (bytes). You can see that, apart from the fixed 4 byte header, 16 bytes form the body of the record. The interpretation of each of these bytes is given in the Excel SDK. Based on this we decipher these bytes and convey data to Objective Grid. When writing, the reverse occurs.

26.2.4 Structure of the Read/Write Mechanism

Let us examine the structure of the read mechanism first. When the grid reads in an Excel file it knows the basic structure of the file as detailed above (types and lengths). Once it reads the type and the length, it checks to see if it can handle this type. If it can be handled at all, each type can be handled by exactly one handler. Two different types of objects handle data. One object is called handler and the other is called table. Handlers are typically for local worksheet data and tables are for global data.
The Objective Grid Excel code looks in a map to see if there is a handler for the record type. If one exists, the stream is read and the data is retained and interpreted. If not, the grid performs a seek (based on the record length that was read from the record header) and goes on to look at the next entry. This continues until the file is read completely.

Two maps are defined in the application—one for handlers and the other for tables. These maps are composed of macros in a manner that is very similar to that of `CWnd` message maps. A typical map structure for tables and handlers is shown below:

```cpp
BEGIN_GX_EXCEL_MAP(CGXExcelTable)
  GX_EXCEL_MAP_ENTRY(typeFont, CGXExcelFontTable, 2)
  GX_EXCEL_MAP_ENTRY(typeStyle, CGXExcelStyleTable, 2)
  GX_EXCEL_MAP_ENTRY(typeXF, CGXExcelXFTable, 2)
  GX_EXCEL_MAP_ENTRY(typePalette, CGXExcelColorTable, 2)
  GX_EXCEL_MAP_ENTRY(typeFormat, CGXExcelFormatTable, 2)
  GX_EXCEL_MAP_ENTRY(typeSST, CGXExcelSSTTable, 2)
  GX_EXCEL_MAP_ENTRY(typeBoundSheet, CGXExcelBoundSheetTable, 2)
END_GX_EXCEL_MAP()

BEGIN_GX_EXCEL_MAP(CGXExcelHandler)
  GX_EXCEL_MAP_ENTRY(typeLabel, CGXExcelLabelHandler, 2)
  GX_EXCEL_MAP_ENTRY(typeNumber, CGXExcelNumberHandler, 2)
  GX_EXCEL_MAP_ENTRY(typeRk, CGXExcelRkHandler, 2)
  GX_EXCEL_MAP_ENTRY(typeMulRk, CGXExcelMulRkHandler, 2)
  GX_EXCEL_MAP_ENTRY(typeColInfo, CGXExcelColInfoHandler, 2)
  GX_EXCEL_MAP_ENTRY(typeRow, CGXExcelRowHandler, 2)
  GX_EXCEL_MAP_ENTRY(typeFormula, CGXExcelFormulaHandler, 2)
  GX_EXCEL_MAP_ENTRY(typeBOF, CGXExcelBOFHandler, 2)
  GX_EXCEL_MAP_ENTRY(typeBlank, CGXExcelBlankHandler, 2)
  GX_EXCEL_MAP_ENTRY(typeMulBlank, CGXExcelMulBlankHandler, 2)
  GX_EXCEL_MAP_ENTRY(typeSSLable, CGXExcelSSTHandler, 2)
  GX_EXCEL_MAP_ENTRY(typeDimension, CGXExcelDimensionHandler, 2)
  GX_EXCEL_MAP_ENTRY(typeWindow2, CGXExcelWindow2Handler, 2)
  GX_EXCEL_MAP_ENTRY(typeMergedCell, CGXExcelMergedCellsHandler, 2)
END_GX_EXCEL_MAP()
```

Each map macro entry has three values. The first is the type of record that this entry can handle. The second is the object type to be used when handling this entry. (Remember that the map merely maps a record type to an object that can handle it). The third (rather redundant) value specifies the size of the record type data that is to be used for comparison. With the current version (Biff 8) this value can always be taken as two bytes. However, in anticipation of changes in future versions of Excel, we designed this product with flexibility and facility in mind.

If you were to add support for another table or handler, you would simply add another entry to this map. If you do not intend to add support for any additional fields you can use the maps from one of the samples.

The lookup sequence is very simple. Let us look at some code that shows this inside the grid:

```cpp
// Read the type of the record.
// Try to locate the handler.
CGXExcelHandler* pHandler = GXExGetHandlerMap()->LocateHandler(wType);
CGXExcelTable* pTable = GXExGetTableMap()->LocateHandler(wType);
ASSERT((pHandler && pTable) == FALSE);
// It has to be either a table or a regular record handler.
// You can't have both kind of handlers simultaneously.
```
One advantage of having this structure is that it affords easy changes to the type of handlers that can be added. It makes it very easy to add support for fields that are not supported by the product out of the box. It also makes it easy to log details about fields that are not read in for debugging purposes.

Writing is slightly different since the writing code lays out the file structure and calls the correct handlers and tables to build the .xls file. Again a map controls the order in which the writing takes place. This map merely controls the sequence of the write and calls the same handlers as the write code to handle the actual write. Let us take a look at this map.

This is defined not in the application but in **CGXExcelReaderImpl**. (For more details on this class, please refer to the online help.)

```cpp
CGXWriteEntry* CGXExcelReaderImpl::GetWriteTableMap()
{
    static CGXWriteEntry _map[] =
    {
        GX_EXCEL_WRITEMAP_ENTRY(typeBOF, gxHandler)
        GX_EXCEL_WRITEMAP_ENTRY(typeWindow1, gxHandler)
        GX_EXCEL_WRITEMAP_ENTRY(typePalette, gxTable)
        GX_EXCEL_WRITEMAP_ENTRY(typeFont, gxTable)
        GX_EXCEL_WRITEMAP_ENTRY(typeXF, gxTable)
        GX_EXCEL_WRITEMAP_ENTRY(typeStyle, gxTable)
        GX_EXCEL_WRITEMAP_ENTRY1(typePalette, gxTable, gxPassTwo)
        GX_EXCEL_WRITEMAP_ENTRY(typeBoundSheet, gxTable)
        GX_EXCEL_WRITEMAP_ENTRY(typeEOF, gx Handler)
        GX_EXCEL_WRITEMAP_ENTRY(ixfNULL, gxHandler)
    };
    return _map;
}

// handlers for local fields
CGXWriteEntry* CGXExcelReaderImpl::GetWriteHandlerMap()
{
    static CGXWriteEntry _map[] =
    {
        GX_EXCEL_WRITEMAP_ENTRY(typeBOF, gxHandler)
        GX_EXCEL_WRITEMAP_ENTRY1(typeBoundSheet, gxTable, gxBoundSheetSpecialPass)
        GX_EXCEL_WRITEMAP_ENTRY(typeColInfo, gxHandler)
        GX_EXCEL_WRITEMAP_ENTRY(typeDimension, gxHandler)
        GX_EXCEL_WRITEMAP_ENTRY(typeRow, gxHandler)
        GX_EXCEL_WRITEMAP_ENTRY(typeRk, gxHandler)
        GX_EXCEL_WRITEMAP_ENTRY(typeWindow2, gxHandler)
        GX_EXCEL_WRITEMAP_ENTRY(typeMergedCell, gxHandler)
        GX_EXCEL_WRITEMAP_ENTRY(typeEOF, gxHandler)
        GX_EXCEL_WRITEMAP_ENTRY(ixfNULL, gxHandler)
    };
    return _map;
}
```

One rather confusing aspect of these maps requires mentioning. You can see from above that although there are two maps (one for tables and one for handlers), they both contain handlers and tables. There is no essential difference between the two as there is with the read maps. The read maps contain tables or handlers. The reason for this is that while the global structure of an Excel file is largely composed of table structures, some fields are handler fields (BOF and EOF, for example).
Remember that this demarcation is something that we created for convenience and is not laid out in the Excel SDK. Therefore, these maps contain both types of objects. However, you can readily see that in each one, one type of object is more predominant than the other.

Another interesting aspect is that these maps can be used to write data in multiple passes. For example, in some cases we may have to calculate offsets and then later write these out. The write code can check for pass one and do the actual write with pass two. Passes are more an implementation detail than anything else; you will not have to use these directly in most cases.

---

In the event that you do have to use passes, please contact our support team for further assistance. The implementation details of passes are not documented and might change in a future version.
26.3 Shared and Unshared Data

26.3.1 Shared Objects

One important point to remember is that when working with Excel workbooks in Objective Grid, the following objects will be shared.

```c
// Property-Object
CGXProperties* m_pProperties;
// Styles Map
CGXStylesMap* m_pStylesMap;
// Pointer to an optional PrintDevice-Object
CGXPrintDevice* m_pPD;
```

26.3.2 Unshared Objective Grid Data

The following will be unique to each sheet grid:

```c
// pointer to an optional Data-Object
CGXData* m_pData;
```

This structure mirrors Objective Grid global and local data with respect to each worksheet. The stylesmap is global while the data itself is local. Incidentally, the current implementation does not allow the first tab to be deleted since it has ownership of the shared data.

26.4 Usage Samples

In this section we will discuss various usage scenarios for the Excel read/write code. The following samples ship with the product:

1. An application that presents a workbook interface, enabling this application to save data in Excel file format. Sample: `<stringray-installdir>\Grid\Excel\ExRdFor`.

2. A grid in a dialog that supports Excel read/write. Sample: `<stringray-installdir>\Grid\Excel\ExDlg`.

3. Adding Excel read/write capability to an existing grid view or window. We will also discuss adding Excel clipboard support to an existing application. Sample: `<stringray-installdir>\Grid\Excel\ExToExist`.

Let us now look at these scenarios in greater detail.
26.4.1 Sample 1

To follow along with this sample, please refer to the `exdlg` sample that is installed under the `Samples\Grid\Excel` folder of your install.

1. Start with the application class implementation. Change the base class so that the application class additionally derives from `CExcelReadAdapter` and `CGXExMapHolder`.

   ```cpp
   class CExrdApp : public CWinApp, public CExcelReadAdapter,
   public CGXExMapHolder
   ```

2. Define `CExcelReadAdapter` in your application’s header (exrd.h in sample) as shown below.

   ```cpp
   // derive from the abstract CGXAppAdapter
   class CExcelReadAdapter : public CGXAppAdapter
   {
   public:
      CExcelReadAdapter();
      virtual ~CExcelReadAdapter();
      CGXNodeManager<CGXTabWndMgr>* CreateTabWndHolder();
      virtual CDocTemplate* GetDocTemplate(CRuntimeClass* pViewClass, CRuntimeClass* pDocClass);
   
   protected:
      CMultiDocTemplate* m_pDocTemplate;
   };
   ```

3. Implement this class as shown below.

   ```cpp
   // Implementation of CExcelReadAdapter,
   // the application adapter for this application.
   CExcelReadAdapter::CExcelReadAdapter()
   {
      m_pDocTemplate = NULL;
   }

   CExcelReadAdapter::~CExcelReadAdapter()
   {
      if(m_pDocTemplate != NULL)
      {
          m_pDocTemplate->CloseAllDocuments(TRUE);
          delete m_pDocTemplate;
      }
   }

   // you have to override this function and return a
   // CDocTemplate class that is appropriate for this
   // view and document. The frame is unimportant
   // Please note that you have to keep track of and
   // delete all instances of CDocTemplates that you create
   // here.

   CDocTemplate* CExcelReadAdapter::GetDocTemplate(CRuntimeClass* pViewClass, CRuntimeClass* pDocClass)
   {
   ```
if (m_pDocTemplate != NULL)
    return m_pDocTemplate;

m_pDocTemplate = new CMultiDocTemplate(
    IDR_EXRDTYPE,
    RUNTIME_CLASS(CExrdDoc),
    RUNTIME_CLASS(CGXExcelChildFrame), // custom MDI child frame
    RUNTIME_CLASS(CExrdView));
return m_pDocTemplate;
}

CGXNodeManager<CGXTabWndMgr>*
CExcelReadAdapter::CreateTabWndHolder()
{
    return new
        CGXNodeManager<CGXTabWndMgr>(RUNTIME_CLASS(CGXExTabWndMgr));
}

The implementation of this class does not vary much except for the part that specifies the
run-time classes of the objects (shown above in bold) and can be copied from project to proj-
ect for the most part. This can also be replaced with any other compatible implementation
that suits your needs.

4. Add the following to the top of the application’s .cpp file (exrd.cpp in sample):

    BEGIN_GX_EXCEL_MAP(CGXExcelTable)
    GX_EXCEL_MAP_ENTRY(typeFont, CGXExcelFontTable, 2)
    GX_EXCEL_MAP_ENTRY(typeStyle, CGXExcelStyleTable, 2)
    GX_EXCEL_MAP_ENTRY(typeXF, CGXExcelXFTable, 2)
    GX_EXCEL_MAP_ENTRY(typePalette, CGXExcelColorTable, 2)
    GX_EXCEL_MAP_ENTRY(typeFormat, CGXExcelFormatTable, 2)
    GX_EXCEL_MAP_ENTRY(typeSST, CGXExcelSSTTable, 2)
    GX_EXCEL_MAP_ENTRY(typeBoundSheet, CGXExcelBoundSheetTable, 2)
    END_GX_EXCEL_MAP()

    BEGIN_GX_EXCEL_MAP(CGXExcelHandler)
    GX_EXCEL_MAP_ENTRY(typeLabel, CGXExcelLabelHandler, 2)
    GX_EXCEL_MAP_ENTRY(typeNumber, CGXExcelNumberHandler, 2)
    GX_EXCEL_MAP_ENTRY(typeRk, CGXExcelRkHandler, 2)
    GX_EXCEL_MAP_ENTRY(typeMulRk, CGXExcelMulRkHandler, 2)
    GX_EXCEL_MAP_ENTRY(typeColInfo, CGXExcelColInfoHandler, 2)
    GX_EXCEL_MAP_ENTRY(typeRow, CGXExcelRowHandler, 2)
    GX_EXCEL_MAP_ENTRY(typeFormula, CGXExcelFormulaHandler, 2)
    GX_EXCEL_MAP_ENTRY(typeBOF, CGXExcelBOFHandler, 2)
    GX_EXCEL_MAP_ENTRY(typeBlank, CGXExcelBlankHandler, 2)
    GX_EXCEL_MAP_ENTRY(typeMulBlank, CGXExcelMulBlankHandler, 2)
    GX_EXCEL_MAP_ENTRY(typeSSLLabel, CGXExcelLabelSSHandler, 2)
    GX_EXCEL_MAP_ENTRY(typeDimension, CGXExcelDimensionHandler, 2)
    GX_EXCEL_MAP_ENTRY(typeWindow2, CGXExcelWindow2Handler, 2)
    GX_EXCEL_MAP_ENTRY(typeMergedCell, CGXExcelMergedCellsHandler, 2)
    END_GX_EXCEL_MAP()

These are the handler and table maps that were explained earlier. These maps have all the
objects that are available for dealing with BIFF records. You can exclude records that you do
not wish to use or add ones that you write to this map. (For further information on adding
your own derived handlers and tables, please refer to the advanced section at the end of
this chapter.)
5. Once the maps have been defined, we have to initialize the application object to hook these maps into the read chain. This is done by adding the following lines to the application object’s constructor.

```csharp
SetHandlerMap(new CGXHandlerMapImpl);
SetTableMap(new CGXTableMapImpl);
```

We do not have to add code to clean up these objects. They will be cleaned up by the application object when it goes out of scope.

6. Add a call to `AfxOleInit()` to the application `InitInstance()` call. This call is very important; its absence will result in inexplicable failures.

7. Add a grid to the dialog as explained in Chapter 9, “DlgGrid Tutorial.”

8. Don’t forget to set the /GR compiler option.

9. Remember to open `stdafx.h` and add the following line at the end of the file:

```csharp
#include "grid\gxexrd.h"
```

10. Add a member of type `CGXExcelReadWnd` to the dialog instance. (This can also be a class derived from this class.)

    ```csharp
    // Excel reader grid control
    CGXExcelReadWnd m_gridWnd;
    ```

11. Add the following to the `OnInitDialog()` code:

    ```csharp
    m_gridWnd.SubclassDlgItem(IDC_GRID, this);
    m_gridWnd.Initialize();
    CString strPath;
    CFileDialog dlgFile(
        TRUE,
        _T("*.xls"),
        NULL,
        OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
        _T("Excel 97 Files (*.xls)|*.xls|All Files (*.*)|*.*||")
    );
    if (dlgFile.DoModal() == IDCANCEL)
        return FALSE;
    strPath = dlgFile.GetFileName();
    m_gridWnd.ReadExcelFile(strPath);
    ```

This will display a dialog at startup and, once you select an Excel file, will populate the grid with the 0th tab of the Excel grid.
12. Add one button for opening Excel files and another button for saving to Excel files. Use the following code for opening and saving files:

**Opening Code (very similar to above):**

```cpp
CString strPath;

CFindDialog dlgFile(
    TRUE,
    _T(".*xls"),
    NULL,
    OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
    _T("Excel 97 Files (*.xls)|*.xls|All Files (*.*)|*.*||")
);
if (dlgFile.DoModal() == IDCANCEL)
    return;

strPath = dlgFile.GetFileName();

BOOL b = m_gridWnd.LockUpdate(TRUE);

m_gridWnd.GetParam()->EnableUndo(FALSE);
m_gridWnd.SetRowCount(0);
m_gridWnd.SetColCount(0);

// If you want to read a particular index of the worksheet
// you can specify the index as the second argument.

m_gridWnd.ReadExcelFile(strPath, 1);
m_gridWnd.LockUpdate(b);
m_gridWnd.GetParam()->EnableUndo(TRUE);
m_gridWnd.Redraw(GX_UPDATENOW);
```

In the call to ReadExcelFile() you can change the index (the second argument) to change the tab being read in. (This goes from zero to the number of tabs minus 1.)

**Saving Code:**

```cpp
CString strPath;
CFindDialog dlgFile(
    FALSE,
    _T(".*xls"),
    NULL,
    OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
    _T("Excel 97 Files (*.xls)|*.xls|All Files (*.*)|*.*||")
);

if (dlgFile.DoModal() == IDCANCEL)
    return;

strPath = dlgFile.GetFileName();
m_gridWnd.SaveExcelFile(strPath);
```

13. Compile the application and try opening a file. You should see the tab that you want displayed.
26.4.2 Sample 2

In this case we will be adding support to an existing application to read and write Excel files.

1. Step 1 is the same as in Sample 1. Changes listed there need to be made to the application object for this sample.

2. Change the view class so that it derives from CGXExcelGrid.

   ```
   class CExtoexistView : public CGXGridView, public CGXExcelGrid
   ```

3. Add the following overrides to the view class as shown below:

   ```
   //Excel specific overrides that delegate to CGXExcelGrid
   virtual BOOL OnPasteFromClipboard(const CGXRange& rg);
   virtual BOOL OnCheckClipboardFormat();
   virtual BOOL CopyRange(const CGXRangeList& selList);
   virtual void OnPreExcelReadFile();
   virtual void OnPostExcelReadFile();
   ```

   ```
   // operations
   void ReadExcelFile(const CString& strFileName, UINT index = 0);
   void SaveExcelFile(const CString& strFileName);
   ```

4. The implementation for these is standard and for the most part can be copied over from one project to the other. This is shown below.

   ```
   BOOL CExtoexistView::OnPasteFromClipboard(const CGXRange& rg)
   {
     return OnPasteFromClipboardImpl(rg);
   }
   ```

   ```
   BOOL CExtoexistView::OnCheckClipboardFormat()
   {
     return OnCheckClipboardFormatImpl();
   }
   ```

   ```
   BOOL CExtoexistView::CopyRange(const CGXRangeList& selList)
   {
     return CopyRangeImpl(selList);
   }
   ```

   ```
   void CExtoexistView::OnExcelopen()
   {
     CString strPath;
     CFileDialog dlgFile(
       TRUE,
       _T("*.xls"),
       NULL,
       OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
       _T("Excel 97 Files (*.xls)|*.xls|All Files (*.*)|*.*|*||*"));
     if (dlgFile.DoModal() == IDCANCEL)
       return;

     strPath = dlgFile.GetFileName();
   }
   ```
BOOL b = LockUpdate(TRUE);

GetParam()->EnableUndo(FALSE);
SetRowCount(0);
SetColCount(0);

// If you want to read a particular index of the worksheet
// you can specify the index as the second argument.
ReadExcelFile(strPath, 0);

LockUpdate(b);
GetParam()->EnableUndo(TRUE);

Redraw(GX_UPDATENOW);
}

void CExtoexistView::OnExcelsave()
{
    CString strPath;
    CFileDialog dlgFile(FALSE, _T(".*xls"), NULL, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT, _T("Excel 97 Files (*.xls)|*.xls|All Files (*.*)|.*|.*|*") );
    if (dlgFile.DoModal() == IDCANCEL)
        return;

    strPath = dlgFile.GetFileName();

    SaveExcelFile(strPath);
}

void CExtoexistView::ReadExcelFile(const CString& strFileName, UINT index)
{
    // add code here to initialize from an Excel file

    CGXExcelWndDocAdapter docAdapt;
    docAdapt.SetGrid(this);
    OnPreExcelReadFile();

    try
    {
        docAdapt.ReadExcelFile(strFileName, index);
    }
    catch(egxExcelRWFlags flags)
    {
        GX_EX_FMT_TRACE("OG Error: CGXExcelReaderImpl::ExcelWriteImpl.")
        CGXExRWErrorHandler::HandleError(flags);
    }
catch(...)  
{
    GX_EX_FMT_MSG()
    return;
}
OnPostExcelReadFile();

void CExtoexistView::OnPreExcelReadFile()
{
    // add pre read code here
    CGXGridParam* pParam = GetParam();
    ASSERT(pParam != NULL);
    pParam->EnableUndo(FALSE);
}

void CExtoexistView::OnPostExcelReadFile()
{
    // add post read code here...typically common Excel
    // emulation code is added
    CGXGridParam* pParam = GetParam();
    pParam->EnableUndo(TRUE);
    SetFloatCellsMode(gxnFloatDelayEval);
    ChangeStandardStyle(CGXStyle().SetWrapText(FALSE));
}

void CExtoexistView::SaveExcelFile(const CString& strFileName)
{
    CGXExcelWndDocAdapter docAdapt;
    docAdapt.SetGrid(this);
    docAdapt.SaveExcelFile(strFileName);
}

5. As in the sample, hook the code that opens the file and saves the file to command messages.
6. Compile the project. Your existing view should be able to read and write Excel files as well as support rich data exchange with Excel through the clipboard.
26.5 Advanced Section

26.5.1 Extending the Reader

In this section we demonstrate how support for additional handlers can be added. Much of this information can be applied to adding support for tables, too.

The reader is fully extensible. If there is a field that you would like to read you just have to decide whether it is to be a table (global data) or handler (local data). Tables usually have globally accessible data referenced by index. You will normally not have to add support for additional tables. Adding support for handlers will probably be more common.

Stripped of trivial data members and such, `CGXExcelHandler` looks like this:

```cpp
class CGXExcelHandler : public CObject
{
   // data declarations
   enum gxexType
   {
      nullPtr,
      corePtr,
      stylePtr,
      paramPtr,
      dataPtr
   };

   // nested classes
   struct gxexpointer
   {
      // default is core pointer
      gxexpointer(void* pv);
      gxexpointer();
      gxexpointer(CGXGridCore* p);
      virtual ~gxexpointer();

      // data
      gxexType vt;
      union
      {
         CGXGridCore* pCore;
         CGXGridParam* pParam;
         CGXStyle* pStyle;
         CGXData* pData;
      };
   };

   // operations

   // Read/Write from/to stream

   virtual BOOL ReadFromStream(IStream* pStream,
      gxexpointer p, DWORD dwLength);
};
```
virtual BOOL WriteToStream(IStream* pStream,
    gxexpointer p, DWORD& dwLength, UINT nTabIndex = 0);

// logging support
// members omitted
// public for the base class
virtual void ResetInternalState() = 0;

// static functions
protected:
    virtual BOOL DelegateHelper(gxexpointer p, BOOL bStore = FALSE);

    // actual init functions
    virtual BOOL InitializeStyle(CGXStyle* pStyle,
            BOOL bStore = FALSE);
    virtual BOOL InitializeCore(CGXGridCore* pCore,
            BOOL bStore = FALSE);
    virtual BOOL InitializeParam(CGXGridParam* pParam,
            BOOL bStore = FALSE);
    virtual BOOL InitializeData(CGXData* pData, BOOL bStore = FALSE);
};

Let us take a brief look at the nested data structures gxexType and gxexpointer. gxexType denotes the type of pointer that is contained inside the gxexpointer structure. This structure gets passed in from the read and write code when the object, which is derived from CGXExcelHandler, is asked to decipher the field. The actual type of the object can change with the field being interpreted. This is normally not the case, however, and the most probable use if this will contain a CGXGridCore pointer. However, if you need an object that does not need access to the core pointer but operates on any other grid data structure, the support for handling this is already built in.

The initialize functions correspond to the data structures that are supported by default. As we mentioned earlier, the structure that is normally used is the CGXGridCore pointer and the corresponding initialization function is InitializeCore().

To better understand this, look at the sequence of what happens during a typical read operation. The reader code reads the type of field and attempts to locate a handler for it. If a handler is found, it stuffs the CGXGridCore pointer that it wants initialized with data that is interpreted in the form of a gxexpointer and calls the object’s ReadFromStream() method (assuming that this is a handler and not a table).

The object would (inside its ReadFromStream() code) read the data structure that pertains to this field and then interpret and apply this to the CGXGridCore object that was passed in. To make this a little more systematic and easier to extend, the actual read and application to the object take place as two steps. First, the actual read takes place. Next, the object calls its base class. Finally, the base class resolves the type of initialization function that is to be called to initialize the object (based on the type of the object as specified by the gxexType (vt) member). The DelegateHelper() function is responsible for this. The following code should clarify this:
BOOL CGXExcelHandler::DelegateHelper(CGXExcelHandler::gxexpointer p,
    BOOL bStore /*=FALSE*/)
{
    BOOL b(FALSE);
    switch(p.vt)
    {
    case CGXExcelHandler::corePtr:
    {
        b = InitializeCore(p.pCore, bStore);
        break;
    }
    case CGXExcelHandler::stylePtr:
    {
        b = InitializeStyle(p.pStyle, bStore);
        break;
    }
    case CGXExcelHandler::paramPtr:
    {
        b = InitializeParam(p.pParam, bStore);
        break;
    }
    case CGXExcelHandler::dataPtr:
    {
        b = InitializeData(p.pData, bStore);
        break;
    }
    default:
    {
        TRACE0("Error in CGXExcelHandler::DelegateHelper. Unrecognized format\n");
    }
    return b;
}

Now let us step back and look at how the data read is performed. To make things simpler we have
a read mechanism that you can also use in your derived handlers. (You are not limited to this and
can read the stream in any manner that you find convenient.)

Typical read code from ReadFromStream() is shown below:

static GXREADSEEKDATA readData[6];

readData[0].Init(&m_lStruct.row); //row
readData[1].Init(&m_lStruct.col); //column
readData[2].Init(&m_lStruct.ixfe); //index to XF record
readData[3].Init(&m_lStruct.wLength); // Length of the string
readData[4].Init(&m_lStruct.grbit); //grbit flags

GXEX_TRY(ReadSeekHelper(pStream, &readData[0], 5))

GXREADSEEKDATA can contain any (common) kind of data that needs to be read in. This can be ini-
tialized with the structures that are to hold the data to be read in. These are normally members of a
structure that is a member of this class. You can declare variables on the stack if you have cases
where the data does not need to persist until the initialization functions are called to initialize the
data. (One typical example would be the length of a string. Once the string is read in, this data does
not need to be retained.)
ReadSeekHelper() can then be called to read this data from the stream and fill the data members that have been passed in. ReadSeekHelper() takes care of checking for errors, much like Data Exchange in dialogs.

Writing is very similar. WriteToStream() is called. The object that has the data that is to be used to compose the record is passed in as a CGXExcelHandler::gxexpointer. Here you can choose types that you want to handle and reject others.

BOOL CGXExcelWindow2Handler::WriteToStream(IStream* pStream, CGXExcelHandler::gxexpointer p, DWORD& dwLength, UINT nTabIndex /*= 0*/)

A helper implementation that can ease the process of writing out data is shown below:

BYTE pszBuffer[GX_EX_SIZEOF_WINDOW2];
__gxdata(wr)
__gxw(typeWindow2)
__gxw(GX_EX_SIZEOF_WINDOW2 - 4)
__gxw(ws.grbit)
__gxw(ws.rwTop)
__gxw(ws.colLeft)
__gxdw(64)
__gxw(ws.wScaleSLV)
__gxw(ws.wScaleNormal)
__gxdw(0) // reserved
__gxend

ULONG cb(0);
__gxExWrite(wr, pszBuffer);
GX_EX_WRITEEX(pStream->Write(pszBuffer, GX_EX_SIZEOF_WINDOW2, &cb));
GXEXDUMP(pszBuffer, GX_EX_SIZEOF_WINDOW2)
dwLength+=cb;

The above code is self explanatory. You can compose a map containing the structures and values that you would like to be written out. These can then be written to a buffer with a call to __gxExWrite(wr, pszBuffer). This buffer can be written to the stream.

One key aspect to remember is to increment the DWORD parameter dwLength that is passed in by reference. This count is used to compute various offsets that get written into the file. There is no surer way to get Microsoft Excel to go down than to forget to increment this count.

From what was detailed above, we can list the overrides that are essential for adding support for a new record type.

virtual BOOL ReadFromStream(IStream* pStream, gxexpointer p, DWORD dwLength);
virtual BOOL WriteToStream(IStream* pStream, gxexpointer p, DWORDs dwLength, UINT nTabIndex = 0);
virtual BOOL InitializeCore(CGXGridCore* pCore, BOOL bStore = FALSE);

One other method needs to be overridden, the pure virtual function ResetInternalState():

virtual void ResetInternalState() = 0;
In your override of this function, clean up any data that you allocate to hold data that is read in or written out. For example, if you had allocated a string buffer to read in a string, delete it in this override. This will be called for clean up.

### 26.5.2 Extending Tables

The essential logic is much the same for tables. The overrides are:

```cpp
virtual BOOL ReadFromStream(IStream* pStream, DWORD dwLength);
virtual BOOL WriteToStream(IStream* pStream, DWORD& dwLength,
    CGXGridParam** pParam, UINT nTabs = 1);
virtual BOOL ApplyToObject(DWORD index, CGXStyle*& pStyleFont);
virtual void ResetInternalState() = 0;
```

The major differences in the implementation logic are explained below:

While handlers are typically used to manipulate CGXGridCore objects, tables are usually associated with a type of object at compile time. They manipulate this object alone. For example, the XF field corresponds to CGXStyle in the grid and correspondingly the XF table manipulates CGXStyle objects. This is specified at compile time by deriving from CGXExcelBaseTable as shown below.

```cpp
class CGXExcelXFTable : public CGXExcelBaseTable<XFStruct,
    CGXStyle>
CGXExcelBaseTable
    provides certain memory management features for derived classes. It allocates and tracks objects of both types passed as template arguments on demand.

A reason for this difference in implementation is the manner in which tables are applied to data when compared to handlers. Tables are almost never applied directly. In fact, you will notice that the ReadFromStream() override for tables does not take any objects pointers. Instead, tables are read in and retained in memory. When a handler needs the services of a table it looks it up and then calls the special function ApplyToObject(). ApplyToObject() takes two parameters, one an index into the table and the other a pointer to a object that can be modified by this table. (The meaning of the index depends on the table, but it is usually interpreted as a zero based index into the table.)

One way to call ApplyToObject() is to allocate an object of this type and then ask the table to apply the data that it has at the index specified to the passed in object. Another way to call it is to simply pass a NULL pointer. The base class will perform the requisite memory management and return a fully composed instance of this object with the data at the specified index.

The purpose of ResetInternalState() is much the same as it is with handlers. One difference is that it is called at the end of the cumulative read or write operation and not after each record is accessed. This is because once a table is read in it must remain accessible to all records that are read in after that. Once the read or write is complete, the table can be freed and ResetInternalState() will be called to do just that.
26.5.3 Supported Excel Fields

The following Excel fields are supported by the Objective Grid library. For a description of these fields, please refer to the Excel SDK documentation.

FONT
STYLE
XP
PALETTE
FORMAT
SST
BOUNDSHEET
LABEL
RK
MULRK
COLINFO
ROW
FORMULA
BOF
BLANK
MULBLANK
SST
DIMENSIONS
WINDOW2
MERGEDCELL

The above records form the core of the BIFF file structure. Almost all of the other records can be added with ease, based on the current architecture.

While we write a good subset of Objective Grid features out in Excel file format, we do not include all of the Excel fields. This means that you might not be able to pull a rich native Excel file in its entirety into Objective Grid. The emphasis of the current implementation is on being able to write out Objective Grid data as .xls rather than on being able to read all the Excel fields. We do leave this extensible to allow users to add additional fields as required.

We would appreciate your feedback on the basic structure. In addition please advise us of other essential fields that you would like added as defaults in future releases of this product.

26.5.4 Background Color

26.5.4.1 Unwanted Black Background

If you change the background color of cells in your grid, try to stick to Excel’s default color palette. When you set the background color of a cell in the grid to certain colors—like RGB(255, 255, 159) for example—and then you copy and paste those cells from a grid into Excel, their background shows up as black in Excel.

If you search the Excel help about colors, you will see information about setting the Excel color palette. If you go to the Tools | Options menu in Excel and click the Color tab, you will see the palette of colors Excel is using. Excel has a palette of 40 standard colors that you can use for setting the background color of a cell (these are the 40 boxes above the line in the Color tab). You may modify
any of these 40 colors and redefine them to whatever you wish. However, the palette of 40 colors that you see is the default Excel palette, and it is only these colors that are supported as cell background colors. Although RGB(255, 255, 159) may be a true color in the Windows sense, it still isn't in the Excel default palette. If the exact color you have chosen for the grid cell is in the Excel default palette, it will show correctly in Excel. If there is no exact match, Excel maps it to black. Excel does not seem to use a 'nearest-color' match; it's either an exact match or it's black.

The Excel documentation goes on to say that you can redefine any of the colors of the palette and save it with the workbook. This would seem at first to solve the problem, but it does not. The problem is that the XLS file is completely generated from the grid information and there is no good deterministic way for the grid to alter the default palette to suit your custom color needs. If you happen to have more than 40 different background colors for your grid cells, you will never be able to see all of them in Excel, no matter what you do to the palette in the XLS file; it just isn't possible. Therefore, the black color is simply a limitation of Excel.

The bottom line is: if you want Excel to reflect your color settings, stick to colors in the default Excel palette.

### 26.5.4.2 Removing Background Color

If you want to remove only the changed background color for a cell style put in place with a `SetInterior()` call, you can simply call `SetIncludeInterior(FALSE)`.
26.6 Important Notes

In this section we list some limitations as well as general notes about the Excel read/write implementation with Objective Grid source code.

- If you have any data that is not recognized by this application (such as Visual Basic macros, for example) this code will be lost when the file is written out. If you are converting an existing Excel file to Objective Grid native format or editing it in Objective Grid, be sure to back up the Excel file first as a precaution. You may want to retain some fields that are not handled by the Objective Grid implementation. These fields will normally be lost when the file is saved by Objective Grid.

- We support only a small subset of the available Excel fields. We offer an extensible architecture that allows easy addition of support for any other fields that you desire.

- It is possible (and easy) to add support for Excel read/write to existing grid applications. This is demonstrated in a sample located at `<stringray-installdir>\Samples\Grid\Excel\ExToExist`.

- Full clipboard support for BIFF 8 is provided. The current implementation has a couple of restrictions, however:
  - When formulas are copied from Excel to Objective Grid only the values that are the result of the Excel formulas are copied over to Grid; the formulas themselves are not copied over.
  - Copying formulas from Grid to Excel is supported; formulas are copied, but because of syntax differences, Excel might not support all of the Objective Grid formulas.

- If you set the Text Color without setting the font, it will not appear in the saved Excel file because no FONT record will be created in the absence of other FONT data.

- Only U.S. English Excel files are supported.

- Unicode is not supported in this version of the Excel read/write classes. When building Grid libraries with Unicode, an alternative to the read/write classes is Excel Automation, discussed in Section 26.7, “Excel Automation with Objective Grid.”

To use Excel Automation, ensure that the Excel read/write option is unchecked in the Grid Build Wizard.
26.7 Excel Automation with Objective Grid

Developed by Microsoft, the Automation approach allows reading from and writing to Excel files and creating applications with run-time data exchange. Automation provides more flexibility and allows developers to use all of the power of MS Excel in their custom applications.

To ensure the ability to build with UNICODE, the Excel read/write option should be unchecked in Grid Build Wizard.

Review the following descriptions and examples of Excel Automation before developing applications with Excel-Grid interoperability:

- Microsoft KB Article 178781 HOWTO: "Automate Excel Using MFC and Worksheet Functions":

- Microsoft KB Article 178749 HOWTO: "Create Automation Project Using MFC and a Type Library":
  http://support.microsoft.com/default.aspx?scid=kb;EN-US;178749#appliesto

- Microsoft KB Article 184663 HOWTO: "Embed and Automate a Microsoft Excel Worksheet with MFC":

- Microsoft KB Article 192348 HOWTO: "Automate Excel and Then Know the User Closed It":

- Microsoft KB Article 311546 HOW TO: "Embed and Automate an Excel Worksheet by Using C++ .NET and MFC":
  http://support.microsoft.com/?kbid=311546

- Microsoft KB Article 179706 HOWTO: "Use MFC to Automate Excel and Create/Format a New Workbook":
  http://support.microsoft.com/default.aspx?scid=http://support.microsoft.com:80/support/kb/articles/q179/7/06.asp&NoWebContent=1

- Microsoft Office Development with Visual Studio:

The following sample demonstrates only a limited implementation of the Excel-Grid integration using Microsoft's Automation. Support for the implementation of additional features based on this technology is beyond regular Technical Support Services; however, Consulting Services may provide additional assistance. Contact your sales representative for more information.
26.7.1 Enabling Excel Integration in a Grid Project

The ExcelGrid sample discussed in this section demonstrates how to enable integration with Excel in a Grid project.

This sample is provided at `<stingray-install-dir>\Samples\Grid\Excel\ExcelGridAutomation`.

26.7.1.1 Creating an ExcelGrid Sample

Create an ExcelGrid sample by adding Excel-related functionality to the regular Tab Grid MDI project (see `CGXTabWnd` in Objective Grid Class Reference) with Formula Engine-enabled and Excel-like features set, as follows:

1. Generate Excel classes from Excel type library as described in KB articles (files Excel9, `.h` and `.cpp), include `*excel9.h` in stdafx.h.
2. Override the `CGXCurrencyEdit` class to implement more Excel-looking formats. (A new control is registered in `CTabGridView::OnInitialUpdate`.)

26.7.1.2 Using the Excel-Related Classes

To use the Excel-related classes:

1. Add `CExcelExchange * m_pExcelExchange;` into class `CTabGridView`, as follows:
   - `m_pExcelExchange` is created in `CTabGridView::OnInitialUpdate`:
     ```cpp
     m_pExcelExchange = new CExcelExchange(this,((CChildFrame *)GetParentFrame())->app);
     ```
   - `m_pExcelExchange` is deleted in the destructor for `CTabGridView`:
     ```cpp
     CTabGridView::~CTabGridView()
     {
         delete m_pExcelExchange;
     }
     ```
2. Note that members for Excel objects are added into class `CChildFrame`:
   - `Application app;
   - LPDISPATCH lpDisp;
   - _Workbook book;
   - _Worksheet sheet;
   - Workbooks books;
   - Worksheets sheets;
3. Use the function `CChildFrame::OpenCloseExcel` for processing an open/close Excel object and exceptions handling.
4. The following function is just a wrapper for Excel function `Workbooks::Open(...)`:  
   ```cpp
   OpenExcel(CString strExcelFile = _T(**)) ;
   ```
In our sample, we call this function with a number of parameters that are required for Excel 9 files. If Excel files are generated from later versions of Excel, the number of parameters should be changed (see Microsoft’s Knowledge Base articles).

5. If the project is built on a machine with Excel installed, then the fully qualified path to the Excel file should be passed into function `Open` to run the executable on another machine. In our sample, we decided to keep the initial Excel file with the format examples in the same folder as the executable. We use the following code to determine the path:

```c++
TCHAR szBuffer[1024];
GetModuleFileName(NULL, szBuffer, 1024);
CString str(szBuffer);
str.Replace(_T("TabGrid.exe"), _T("ExcelFormats.xls"));
```

6. To verify that the Excel file exists:

```c++
WIN32_FIND_DATA wfd;
HANDLE hFind = FindFirstFile(str, &wfd);
if (hFind == INVALID_HANDLE_VALUE)
    str = "";
FindClose(hFind);
```

If there is no Excel file where it is expected, an empty string is passed into function `Open(…)` and a new workbook is opened.

In the destructor, we use the exception processing module to call `app.Quit();` as shown here:

```c++
CChildFrame::~CChildFrame()
{
    OpenCloseExcel(FALSE);
}
```

An `ExcelExchange` object is created for each view in the tab/worksheet, and the following code is used to iterate through views/worksheets in functions `CChildFrame::OnReadBook` and `CChildFrame::OnWriteBook`:

```c++
CView * pView;
POSITION pos = pDoc->GetFirstViewPosition();
short nSheet = 0;
while (pos != NULL)
{
    nSheet++;
    pView = pDoc->GetNextView(pos);
    if (pView->IsKindOf(RUNTIME_CLASS(CTabGridView)))
    {
        ((CTabGridView *)pView)->GetExcelExchange()->ReadWrite(
            WRITE, TRUE, TRUE, nSheet);
    }
}
```

The technique used in this sample fixes an Objective Grid bug which affects scrollbar behavior. For more details on this bug fix, please refer to these `CTabGridView` functions: `GetParentTabWnd`, `GetScrollBarCtrl`, `UpdateScrollbars`, and `GetScrollBarCtrl`. 

---

488
26.7.1.3 Data And Style Exchange Classes

This section describes the classes and functions provided in the Objective Grid integration with Excel.

Class CExcelExchange

The function CExcelExchange::ReadWrite provides general processing of read/write procedures and exception handling.

Performance of exchange between Grid and an automated Excel object depends significantly on how this exchange is implemented. For example, you can set all style formatting both in Grid and Excel only in OnInitialUpdate and then change values in Excel as soon as they are changed in Grid. This scenario will demonstrate very good performance, although it will depend on specific logic of the application. In our sample, we demonstrate the worst-case scenario to show the approach is usable even in this case. You’ll see that the lookup of all styles is provided as styles and values are translated and sent between Grid and Excel. In this scenario, it would be advantageous to review which styles are actually used in a particular application and exclude unused styles from translation. When all styles are excluded and only values lookup and exchange are provided, performance is adequate.

The CExcelExchange::Read and CExcelExchange::Write functions determine if styles or only values lookup should be provided. The function CExcelExchange::IndexToString is used to transform cell coordinates used in Grid (nRow, nCol) into Excel specific coordinates (such as AD55).

Class CExcelWrite

Class CExcelWrite includes WriteStyles and FormatRangeWrite functions, just for improving the structure of the code. The function WriteValueFormat translates the format of the value (such as string, datetime, currency, etc.) from Grid to Excel and includes the following functions:

- WriteCoveredCells
- WriteFont
- WriteInterior
- WriteBorders
- WriteFormula
- WriteVerticalAlignment
- WriteHorizontalAlignment
- WriteWrapText

There are four style types in Grid: Table, Column, Row, and Cell. As indicated by their names, each of the following functions translates one of these styles:

- WriteTableStyles
- WriteColStyles
- WriteRowStyles
- WriteCellStyles

The following functions are used to write an array of values from Grid to Excel (as described in Microsoft’s Knowledge Base articles):

- WriteValues
- WriteValuesArray
- FillSafeArray
- ValuesToArray
Class CExcelRead

The following functions simply improve the structure of the code:

ReadValues
ReadStyles
FormatRangeRead

Function ReadValueFormat translates the format of a value (for example: string, datetime, currency, et cetera) from Excel to Grid and includes the following functions:

ReadCoveredCells
ReadFont
ReadInterior
ReadBorders
ReadVerticalAlignment
ReadHorizontalAlignment
ReadFormula

The following functions translate Excel styles into four Grid style types -- Table, Column, Row, and Cell -- as indicated by the function name:

ReadTableStyles
ReadColStyles
ReadRowStyles
ReadCellStyles

The function ReadValuesArray is used to read an array of values from Excel to Grid (as described in Microsoft’s Knowledge Base articles).

26.7.1.4 Notes for ExcelGrid Sample

- There is no one-to-one correspondence between formats and styles in Grid and Excel, so there is room for setting this correspondence more or less voluntarily. For example, in our sample datetime controls in Grid are set to correspond with datetime formatting in Excel, while thick non-solid borders in Excel are set to correspond with thin borders in Grid.

- It is difficult to find descriptions of many C++ functions that are implemented in generated Excel classes. Nevertheless it is usually not difficult to find and use the descriptions of similar VB functions instead.

- To build the sample on a machine with some version other than Excel 2000, Excel files should be generated as described in Microsoft’s Knowledge Base articles, files Excel9 in our sample should be replaced, and the call to function Open(...) should be changed.

- To enable copying a formula that is different between Grid and Excel, the formula should be translated (only the name or, if necessary, parameters also) in functions ReadFormula and WriteFormula.
26.7.2 Testing the ExcelGrid Sample

This section describes how to test the ExcelGrid sample that we created in Section 26.7.1, “Enabling Excel Integration in a Grid Project,” and also includes a description of the menu items that get implemented in this sample.

To test the ExcelGrid sample located at <stingray-installdir>\Samples\Grid\Excel\ExcelGridAutomation:

1. Run the sample; the ExcelFormats file will open simultaneously with Grid.
2. On Sheet 1, all implemented formats are displayed. Any different format should be implemented in code.
3. On Sheet 3, there are samples of formula usage. For other functions, if name or parameters are not identical in Grid and Excel, translation should be implemented.
4. For testing, we recommend opening a new workbook, reading from ExcelFormat into Grid, and then writing from Grid into a new workbook.

The following menu items are implemented in the sample:

- Exchange between current sheet in Excel and current View in Grid: ReadSheet and WriteSheet.
- Exchange between current workbook in Excel and current TabGrid: ReadBook and WriteBook.
- Exchange values only between current sheet in Excel and current View in Grid: ReadSheetData and WriteSheetData.

26.7.3 Benefits of Excel-Grid Integration

The sample outlined above demonstrates the ease of integrating Excel into your Objective Grid project. This Automation-based integration also offers the following new features:

- UNICODE support
- More formats implemented and easy customization of formatting
- Easy customization of formula translation
- Ability to implement run-time exchange and access to all features of Excel (use formulas, web-services, etc.)
27.1 Overview

The Objective GridEx Extension Library extends Objective Grid to provide Feature Pack functionality to the Objective Grid binaries. This extension library is built as a separate binary that works in conjunction with your Objective Grid application, so there is no need to rebuild the Objective Grid libraries.

This design allows you to:

- Easily add or extend new Feature Pack functionality to existing Objective Grid applications.
- Simply compile and link the Objective GridEx library to existing Objective Grid applications.
- Quickly gain Feature Pack functionality with little to no modifications to existing applications.
- Easily extend your own functionality.
- Additionally link the extended Stingray Foundation Library (SFLEEx) which may be used to extend your Grid application’s frames, windows controls, etc. to take advantage of the Feature Pack.

This chapter illustrates how to extend your Grid application to add GridEx features.

27.1.1 Architecture

The library is based on MFC classes introduced with the MFC Feature Pack in Microsoft Visual Studio 2008 SP1. Objective GridEx is built as an extension of, and is tightly coupled with, the Objective Grid MFC extended libraries. The approach used is described in Section 15.3, “Sharing Extended Functionality Across Classes,” which uses template classes derived from CGXGridCore. All code in Objective GridEx is unmanaged. Further, all code uses the namespace ogex to avoid collisions.
Objective GridEx contains the following functionality in the following header files:

Table 32 – GridEx Header Files

<table>
<thead>
<tr>
<th>Headers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GridEx\ExcelSelection\GXGridCoreEx.h</td>
<td>Template class for extended Excel functionality</td>
</tr>
<tr>
<td>GridEx\ExcelHeaders\GXHeaderExcel.h</td>
<td>CGXExcelHeader class derived from CGXHeader</td>
</tr>
<tr>
<td>GridEx\ColorButton\GXColorButton.h</td>
<td>CGXColorButton derived from CMFCColorButton &amp; CGXPushbutton</td>
</tr>
<tr>
<td>GridEx\PopupMenuEdit\PopupMenuEdit.h</td>
<td>CPopupMenuEdit class derived from CGXEditControl</td>
</tr>
<tr>
<td>GridEx\ToolTips\gxttnex.h</td>
<td>CGXToolTipCtrl class derived from CMFCToolTipCtrl</td>
</tr>
<tr>
<td>GridEx\ToolTips\GXGridToolTipsEx.h</td>
<td>Template class to implement, enable and set Tool Tips</td>
</tr>
</tbody>
</table>

27.1.2 GridEx Features

Objective GridEx provides support for:

1. Feature pack style tool tips supporting multiline text and icons
2. Feature pack color buttons embedded in a cell
3. Alpha-blending Excel-looking selections
4. Context menus in active and inactive grid cells
5. Excel-like headers supporting styling (Blue, Silver and Black Office styles)

In addition, Objective Grid itself includes another feature related to GridEx:

- Use of Objective Grid as a pseudo-3D color chart

27.1.3 System Requirements

The GridEx features work with all currently supported platforms. Please review the Supported Platforms document for the current list of supported compilers per release.

27.1.4 Code Source and Sample Locations

27.1.4.1 Source Locations

The GridEx Feature Pack extension product file locations are:

Include files:

<Install_dir>\Include\GridEx  // header files
Resources:
<Install_dir>\Include\Res      // resource files

Libraries:
<Install_dir>\Lib\vc9\x86      // Win32 binary files
<Install_dir>\Lib\vc9\x64      // 64-bit binary files

Source files:
<Install_dir>\Src              // *.sln/*.vcproj and \obj subdir
<Install_dir>\Src\GridEx       // implementation files

Note: Shipped examples that demonstrate the Color Button and Context Menu features use GridEx resources. To
dynamically link the GridEx library, add the macro _OGEXDLL to the Preprocessor Definitions. If a dynamic link is
not set, you must add the includes OGResEx.h and OGResEx.rc.

27.1.4.2 Sample Locations

Two sets of samples ship that illustrate the new features:

- GridEx sample, <Install_dir>\Samples\GridEx
- Objective Grid 13.0.1 sample, illustrating Print/Print Preview,
  <Install_dir>\Samples\Grid\Print Preview

Note: Build the Objective Grid libraries before building the GridEx library and the new samples. For the ExcelGrid sample, you must also build the FoundationEx library.
27.2 Tool Tips

Objective GridEx extends the tool tip functionality by deriving a new class `CGXToolTipCtrlProxyEx` from the `CMFCToolTipCtrl` class, extending the tool tip functionality to display an icon, a label, and a description. You can also customize a tool tip’s visual appearance using a gradient fill, custom text and border colors, bold text, rounded corners, or a balloon style.

27.2.1 Implementation

- New class `CGXToolTipCtrl` derives from `CMFCToolTipCtrl` (files `gxttnex.h`, `gxttnex.cpp`).
- New class `CGXToolTipCtrlProxyEx` derives from `CGXNoToolTipCtrlProxy` and uses `CGXToolTipCtrl` instead of `CToolTipCtrl`. The derived class overrides some functions including `m_wndTtnToolTip` (files `gxttnex.h`, `gxttnex.cpp`).
- New template class `CGXGridToolTipsEx<T>` derives from `CGXGridCore` and overrides the following functions in `GXGridToolTipsEx.h`:
  ```cpp
  void ImplementToolTips();
  BOOL EnableGridToolTips(BOOL b);
  void SetToolTipParams(CMFCToolTipInfo* pParams);
  ```
- A public accessor `GetTooltipProxy()` was added to return the protected member `m_pTooltipProxy` value in the `CGXGridCore` class (existing file `gxcore.h`).
- The following instantiations are provided for convenience:
  ```cpp
  typedef CGXGridToolTipsEx<CGXGridView> CGXGridViewToolTipsEx;
  typedef CGXGridToolTipsEx<CGXGridWnd> CGXGridWndToolTipsEx;
  ```

27.2.2 Setting Tool Tips in a Grid

On Grid initialization, call function `EnableGridToolTips(TRUE)`. The extended tool tips are now displayed with the current Windows theme or style, depending on the application style programmatically set. Using the `CMFCToolTipInfo` class to store information about the visual appearance of tool tips, you can customize a tool tip’s appearance or style by setting values on its various properties. For example:

```cpp
CMFCToolTipInfo Params;
Params.m_bBoldLabel = FALSE;
Params.m_clrFill = RGB(255, 255, 255);
Params.m_clrFillGradient = RGB(228, 100, 24);
Params.m_clrText = RGB(0, 83, 255);
Params.m_clrBorder = RGB(255, 255, 0);
Params.m_bVislManagerTheme = TRUE;
Params.m_bDrawSeparator = FALSE;
Params.m_b RoundedRectangles = TRUE;
Params.m_nGradientAngle = 30;
WndGrid.SetToolTipParams(&Params);
```
Please refer to the MSDN description at http://msdn.microsoft.com/en-us/library/bb984634.aspx for detailed information regarding the implementation and use of the `CMFCToolTipInfo` class.

Tool tips allow multiline text, formatted as "line1\nline2\nline3". The description is separated from the title with delimiter "@@". Then, add your multiline text to the following Grid function call, like so:

```cpp
.SetUserAttribute(GX_IDS_UA_TOOLTIPTEXT, "Color \nButton @@ CMFC ToolTip")
```

To display bitmaps in a tool tip, the bitmap file should be included in your application’s resources using a resource ID, for example, `IDB_BITMAP1`. Then, add the bitmap resource to the following Grid function call:

```cpp
.SetUserAttribute(GX_IDS_UA_TOOLTIPTEXT + 1, (LONG)IDB_BITMAP1)
```

Refer to Section 5.2.7, “Registration and Usage of User Attributes,” for more details.

### 27.2.3 Related Samples

Tool Tips are demonstrated in the following samples:

- ToolTipColorPicker
- VisualDataSet
- ExcelGrid
27.3 Color Button

Objective GridEx combines a new Feature Pack control CMFCColorButton with CGXPushButton to create a CGXColorButton that provides an enhanced color picking.

By default, the CGXColorButton class behaves as a push button that opens a color picker dialog box. The color picker dialog box contains an array of small color buttons and an “other” button that displays a custom color picker. (The standard system “other” button is labeled “More Colors…”)

When you select a new color, the CGXColorButton object reflects the change and displays the selected color.

27.3.1 Implementation

A new class CGXColorButton derives from CMFCColorButton and CGXPushButton. The CGXColorButton control is designed to be embedded in one or more grid cells. Implementation details are in the files GXColorButton.h and GXColorButton.cpp.

To implement a GridEx color button control, add a CGXColorButton variable to your application, and then call the constructor and Create() methods of the CGXColorButton object.

The color picker dialog box is displayed by the CGXColorButton::OnShowColorPopup() method when the framework calls the OnLButtonDown event handler. To support custom color selection, you can override the CGXColorButton::OnShowColorPopup() method.

The CGXColorButton object notifies its parent that a color is changing by sending it a WM_COMMAND | BN_CLICKED notification. The parent uses the CGXColorButton::GetColor() method to retrieve the current color.

Please refer to the ColorDlg.cpp file in the GridEx sample ToolTipColorPicker for details on constructing, registering, embedding, and using this new control.

27.3.2 Setting CGXColorButton in a Grid

To set or embed a color button in a grid:

1. Add #include "CGXColorButton.h" to the .cpp file for your CGXGridView-derived class, or to your CDialog-derived class if Grid is used on a dialog.

2. Register the control, for example:

   ```cpp
   CGXColorButton* pColorButton = new CGXColorButton(&m_wndGrid);
pColorButton->EnableOtherButton(_T("Other"));
pColorButton->EnableAutomaticButton(_T("Automatic"),
                                       RGB(255, 0, 255));
pColorButton->SetColumnsNumber(5);
pColorButton->SizeToContent(TRUE);
m_wndGrid.RegisterControl(IDS_CTRL_COLORBUTTON_OGEX,
                           (CGXControl*)pColorButton, FALSE, FALSE);
   ```
3. Set the control in the \textit{CGXStyle} object in a cell, like so:

\begin{verbatim}
.SetControl(IDS_CTRL_COLORBUTTON_OGEX)
.SetValue(RGB(255,0,0))
\end{verbatim}

\section*{27.3.3 Intercepting Calls in the \textit{CGXColorButton} Control}

You can intercept calls to functions in the \textit{CGXColorButton} control in two ways:

\begin{itemize}
\item Via message handling
\item By overriding the function \textit{OnModifyCell()} in your \textit{CGXGridCore}-derived class
\end{itemize}

\subsection*{Message Handling}

In the header file for the class in which the message should be trapped, add:

\begin{verbatim}
afx_msg void OnColorPicker();
\end{verbatim}

Add to the message map:

\begin{verbatim}
ON_BN_CLICKED(IDS_CTRL_COLORBUTTON_OGEX, OnColorPicker)
\end{verbatim}

For example, your handle code might look like this:

\begin{verbatim}
void CColorPickerDlg::OnColorPicker()
{
    CGXControl* pControl = m_wndGrid.GetCurrentCellControl();
    if(CString(pControl->GetControlClass()->m_lpszClassName) == 
       _T("CGXColorButton"))
    {
        COLORREF color = ((CGXColorButton*)pControl)->GetColor();
        CString s;
        s.Format(_T("%d,%d,%d"),
                 GetRValue(color),
                 GetGValue(color),
                 GetBValue(color));
        ROWCOL nRow, nCol;
        m_wndGrid.GetCurrentCell(nRow, nCol);
        m_wndGrid.SetStyleRange(CGXRange(nRow+2,nCol),
                                  CGXStyle()
                                  .SetValue(s).SetHorizontalAlignment(DT_CENTER));
    }
    m_wndGrid.Redraw();
}
\end{verbatim}
Overriding `OnModifyCell()`

Example code:

```cpp
void CToolTipGridWnd::OnModifyCell(ROWCOL nRow, ROWCOL nCol)
{
    if(CString(
        GetCurrentCellControl()->GetControlClass()->m_lpszClassName) ==
        _T("CGXColorButton"))
    {
        CGXStyle style;
        GetStyleRowCol(nRow, nCol, style);
        COLORREF color = (COLORREF)(style.GetDWordValue());
        SetStyleRange(CGXRange( nRow+2, nCol),
                      CGXStyle().SetInterior(color));
    }
    CGXGridCore::OnModifyCell(nRow, nCol);
}
```

27.3.4 User Interaction

Cells that change values as a result of direct or indirect interaction are redrawn. Users cause direct interactions, while indirect interactions are the result of cell reference changes.

Users can interact with the `CGXColorButton` control in a Grid application using a mouse or keyboard.

**Keyboard actions:**

- *Arrow keys*: Moves current cell to make control active
- *Arrow Left, Right, Up, Ctrl+Down, and Tab keys*: Moves current cell from cell with control
- *Down Arrow or Space key when control is active*: Shows pop-up
- *Space key when pop-up is displayed*: Hides pop-up
- *Arrow Up and Down*: Navigates in pop-up
- *Enter*: Chooses selection on pop-up and hides pop-up
- *Arrow left and right when pop-up is displayed*: Hides pop-up and moves current cell

**Note:** The key navigation on the Colors dialog was not updated, so it is similar to `CMFCCColorButton`.

27.3.5 Related Samples

`CGXColorButton` is demonstrated in following samples:

- ToolTipColorPicker
- VisualDataSet
27.4 Excel-Like Selection

Excel-Like selection functionality is improved in Objective GridEx with the use of the CDrawingManager class from MFC's Feature Pack. One of these improvements takes advantage of the feature that draws transparent or semitransparent pixels, called alpha-blending.

In addition to alpha-blend drawing, other Excel-like implementations have been updated, including drawing for selection frame, drag and drop, Excel look and feel, and copy/cut and paste.


27.4.1 Implementation

All new features are implemented in an override of the CGXGridCore class in the file GXGridCoreEx.h. To make these new features applicable both for CGXGridWnd and CGXGridView-derived grids, the implementation uses a template class:

template<class T>
class CGXGridCoreEx : public T

The following instantiations are provided:

typedef CGXGridCoreEx<CGXGridView>  CGXGridViewEx;
typedef CGXGridCoreEx<CGXGridWnd>   CGXGridWndEx;

GridEx uses the CDrawingManager class to take advantage of its complex drawing algorithms available in all versions of the MFC Feature Pack. The example below shows how to implement alpha-blending in your GridEx application:

CDrawingManager dm(*pDC);
dm.HighlightRect(rectItem,
                  m_AlphaBlend.nPercentage,
                  m_AlphaBlend.clrTransparent,
                  m_AlphaBlend.nTolerance,
                  m_AlphaBlend.clrBlend);

Please refer to the CDrawingManager class description in Microsoft's MSDN Library for further details.

27.4.2 Setting an Excel-Like Selection in a Grid (for CGXGridView-Derived grid)

1. Add #include "GXGridCoreEx.h" in a View class derived from CGXGridViewEx, for example:

   class CExcelGridView : public CGXGridViewEx

Or, to use tool tips, derive from the following template class:

   class CExcelGridView : public CGXGridToolTipsEx<CGXGridViewEx>
2. In the function `OnInitialUpdate()`, call `InitializeExcel()`.

   Optionally, for more similarity with Excel, call `EnableFormulaEngine()` before calling `CGXGridView::OnInitialUpdate()`.

3. To customize alpha-blending parameters:
   
   - Set all values using the function `AlphaBlendParams()`:
     ```cpp
     virtual void AlphaBlendParams(int nPercentage = 50,
                                    COLORREF clrTransparent = (COLORREF)-1,
                                    int nTolerance = 0,
                                    COLORREF clrBlend = RGB(236,234,245) );
     
     For example, `AlphaBlendParams(50,(COLORREF)-1,0,RGB(0,255,0));`.
     ```
   
   - Set individual values using the specific functions:
     ```cpp
     void SetSelectionColor(COLORREF clrBlend)
     {m_AlphaBlend.clrBlend = clrBlend;}
     
     void SetSelectionColor Transparent(COLORREF clrTransparent)
     {m_AlphaBlend.clrTransparent = clrTransparent;}
     
     void SetSelectionPercentage(int nPercentage)
     {m_AlphaBlend.nPercentage = nPercentage;}
     
     void SetSelectionTolerance(int nTolerance)
     {m_AlphaBlend.nTolerance = nTolerance;}
     
     You can also change the frame color for a copy-cut selection, for instance, `SetCopyCutSelectionColor(RGB(0,255,0));`.
     ```

27.4.3 Excel-Like Selection Drawing Behaviors

You can perform the following keyboard or mouse exercises with the selections in a grid application to view each drawing behavior.

- **Using the mouse**: Selects the alpha-blend drawing over the selected range except for the current cell, and frames the selection with a black solid frame featuring a marker on the right bottom corner. The selection is expanded automatically to include the entire covered area if any cell from this area is in the selection.

- **Clicking a column header**: Selects the alpha-blend drawing over the selected column, except for the header and the current cell that is set on row 1, and frames the selection with a black solid frame featuring a marker on the top right corner.

- **Clicking a row header**: Selects the alpha-blend drawing over the selected row except for the header and the current cell that is set on column 1, and frames the selection with a black solid frame featuring a marker on the bottom left corner.

- **Using the Ctrl key**: Selects a multi-range alpha-blend drawing over the selected range except for the current cell, and frames only the current cell with a thin solid rectangle.
◆ **Holding down the Shift key**: Expands the alpha-blend drawing selection over the selected ranges, and frames the expanded range if there is no multi-range selection. The position of the current cell is not changed.

◆ **Selecting or unselecting the entire grid**: Clicking on cell (0,0); for the selected table sets the current cell to (1,1) and frames all cells except headers.

◆ **Dragging and dropping**: Drag and drop one selected range to either copy the selection or to move it if pressing Ctrl. This method moves all covered cells, similar to the functionality in Excel. Note that you cannot drop onto a covered area, i.e. a range of cells that span across multiple cells.

◆ **Autofilling**: Expand the selection by dragging the selection’s marker to copy data. This works for both individual cells as well as for column or row selections.

◆ **Selecting a range to copy, cut, or paste**: When using the Copy or Cut functions via either keyboard shortcuts (Ctrl+C or Ctrl+X) or the Edit or Context menus, if implemented, the selected range is designated by a dashed rectangle frame. If the operation is Cut, the frame is removed. If the operation is Paste (using either Ctrl+V or the menu option) or Copy — and multiple paste is available — the frame is removed with the use of the Esc key. For both the Copy and Cut operations, the frame is removed and the data is pasted with the use of the Enter key; the frame is also removed when typing a character in any cell. Pasting onto a covered area is not allowed if more than one cell is selected for Copy or Cut.

◆ **Copying or cutting a multi-range selection**: This feature goes beyond Excel functionality. Use the Ctrl key as described above to capture a multi-range selection in which each range is framed with a dashed rectangle. When the Paste operation is executed, the cells with data are shifted to exclude the empty cells between them, and the union range is selected.

### 27.4.4 Related Samples

Alpha-blending selection is demonstrated in the sample ExcelGrid.
27.5 Context Menu

GridEx supports a right-click context menu for both an inactive cell and for an active cell with CGXEditControl. The implementation is based on use of the Feature Pack class CContextMenuManager.

27.5.1 Implementation

In the handlers for a right-click event, GridEx uses CContextMenuManager::ShowPopupMenu():

CGXGridCoreEx<T>::OnRButtonClickedRowCol (file GXGridCoreEx.h):
ClientToScreen(&pt);
((SFLWinAppEx*)AfxGetApp())->GetContextMenuManager()->ShowPopupMenu(IDR_POPUP_EDIT, pt.x, pt.y, this, TRUE);

Note: SFLWinAppEx is a class implemented in the FoundationEx library.

For an active cell, new class CPopUpMenuEdit derives from CGXEditControl with the code implemented in handler OnContextMenu in files PopUpMenuEdit.h and PopUpMenuEdit.cpp:

GetContextMenuManager() ->
ShowPopupMenu( IDR_POPUP_EDIT, point.x, point.y, Grid() ->
GridWnd(), TRUE );

27.5.2 Setting a Context Menu in Grid

GridEx provides a default menu IDR_POPUP_EDIT_OGEX, but you can replace it with a custom menu in your application if you wish. The default menu provides Copy, Cut and Paste commands.

1. In CExcelGridApp::InitInstance(), call InitContextMenuManager().
2. In CExcelGridApp::PreLoadState(), call

getContextMenuManager() -> AddMenu(strName, IDR_POPUP_EDIT_OGEX);

You can enable or disable a context menu using these CGXGridCoreEx functions:

- In an active cell with an Edit control: EnableContextMenuInEditCell()
- In the current inactive cell or selected range: EnableContextMenuInCurrentCell()
- In both above cases: EnableContextMenu()

27.5.3 User Interaction

- Right-clicking on an inactive cell: The cell becomes the current cell and a context menu displays.
Right-clicking on a selected range: The upper left cell becomes the current cell in the selected range, and a context menu displays.

### 27.5.4 Related Samples

The Context Menu is demonstrated in the sample ExcelGrid.

### 27.6 Excel-Like Headers

GridEx modifies the default Grid row and column headers to closely mimic Excel, and includes the following features:

- Color schemes (Excel Blue, Black, Silver styles and custom)
- Gradient drawing of column headers
- Specific drawing of the top-left cell
- Color change when mouse cursor is over the header

### 27.6.1 Implementation

Class `CGXHeaderExcel` derives from existing class `CGXHeader` and contains new and overridden functions in the files `GXHeaderExcel.h, GXHeaderExcel.cpp`.

The function `CGXHeaderExcel::GetHeaderState()` determines the state of the header which is drawn, particularly if it is a row, column or the top-left header. This function reports whether it is pressed (i.e. the row, column or entire grid selected), it has the mouse cursor over it, or if any selected cells are in a contiguous row or column.

The overridden `Draw()` function draws headers depending on header state and application style chosen while the MFC class `CDrawingManager` is used for gradient drawing. For example:

```cpp
CDrawingManager dm(*pDC);
dm.FillGradient2(rc,
  ExcelHeaderColors[HeaderState],
  ExcelHeaderColors[HeaderState+1],90);
```

The implementation includes the following enumeration in file `GXHeaderExcel.h` used for header component colors that change with application style or with custom settings:

```cpp
enum enHeaderColor {Row, RowMouse, RowSelected, RowSelectedMouse, RowPressed, RowPressedMouse, Column, ColumnTop, ColumnMouse, ColumnMouseTop, ColumnSelected, ColumnSelectedTop, ColumnSelectedMouse, ColumnSelectedMouseTop, ColumnPressed, ColumnPressedMouse, ColumnPressedTop, ColumnPressedMouseTop, ZeroCell, ZeroCellMouse, ZeroCellPressed, ZeroCellPressedMouse, TopLine, Separator, SeparatorSelected, SeparatorPressed,ZeroCellTriangle, ZeroCellHighlightBorder, ZeroCellBorderTop, ZeroCellBorderLeft, ZeroCellBorderRight);```
For gradient drawing, two colors for one component are used, such as Column and ColumnTop.

### 27.6.2 Setting Excel-Style Headers in Grid

Set headers with the following code in function `CGXGridCoreEx<T>::InitializeExcel()`:

```cpp
CGXHeader* pHeader = new CGXHeaderExcel(this);
((CGXStatic*)pHeader)->ImplementCellTips();
RegisterControl(GX_IDS_CTRL_HEADER, pHeader);
ChangeColHeaderStyle(CGXStyle().SetControl(GX_IDS_CTRL_HEADER));
```

**Note:** Currently, the Excel-look features do not work separately, so do not exclude anything from function `InitializeExcel()` to avoid unpredictable behavior.

Color schemes are selected automatically when function `CGXGridCoreEx<T>::OnStyleChanged()` is called. If the application style is changed from the menu, it is convenient to trap the message `WM_APPSTYLE` (that is sent from class `SFLMDIFrameWndEx` implemented in FoundationEx library) and call `OnStyleChanged()` in its handler.

You can set a custom color for any member of the enumeration `enHeaderColor`, calling the function `SetCustomHeaderColor(enHeaderColor type, COLORREF color)`, for example:

```cpp
SetCustomHeaderColor(Row, RGB(255, 0, 0));
```

### 27.6.3 User Interaction

Header colors change in following cases:

- When a row or column is selected by clicking on header
- When any cell is selected in a contiguous row or column
- When the mouse cursor hovers over the header or leaves it

### 27.6.4 Related Samples

Excel-like headers are demonstrated in the sample ExcelGrid.
27.7 **Use of Objective Grid as a Pseudo-3D Color Chart**

When a dataset displayed in a grid is large enough, it is difficult to analyze it using numbers in cells. As part of the GridEx functionality, Objective Grid can be set to represent a large dataset as a color picture.

Each cell in the grid displays a specific color instead of a number. The colors that appear in the grid are the result of value encoding: the whole range of values is divided into subranges, with a different color corresponding to each subrange.

### 27.7.1 Implementation

An override of the function `GetStyleRowCol()` implements color coding, with the following code:

```cpp
val=_ttoi(m_pDlg->m_Data.GetValueRowCol(nRow-1,nCol));
if (val<m_pDlg->m_nRange[0])
    ind=0;
else if (val>=m_pDlg->m_nRange[0] && val<m_pDlg->m_nRange[1])
    ind=1;
else if (val>=m_pDlg->m_nRange[1] && val<m_pDlg->m_nRange[2])
    ind=2;
else if (val>=m_pDlg->m_nRange[2] && val<m_pDlg->m_nRange[3])
    ind=3;
else if (val>=m_pDlg->m_nRange[3] && val<m_pDlg->m_nRange[4])
    ind=4;
else if (val>=m_pDlg->m_nRange[4] && val<m_pDlg->m_nRange[5])
    ind=5;
else if (val>=m_pDlg->m_nRange[5])
    ind=6;
style.SetInterior(m_pDlg->m_col[ind]);
```

### 27.7.2 Sample

Use of Objective Grid as a Pseudo-3D Color Chart is demonstrated in a sample VisualDataSet in which two grids are placed on a dialog.

In this sample, the upper grid displays a dataset in 150 x 365 cells (for example, sales of 150 products during a year), and each cell represents a color value. For instance, you can see a number for each cell in a ToolTip using class `CGXToolTipCtrl`.

For navigation, you can call vertical and horizontal cursors. The cursors' position is set by double-clicking on an active part of a grid or by using the DateTime and SpinEdit controls in the right part of the upper grid. To remove the cursors, just click on the grid. In the sample, data is simulated and stored in a `CGXData` object, and its contents are displayed with a grid. In real applications, the data source (a database, for instance) can be bound to a `CGXData` object.

A second grid features `CGXColorButton` and `CGXSpinEdit` controls used for encoding settings. You can specify which color will represent values in a particular subrange and can determine the value limits for each subrange. Clicking the button Apply redraws the upper grid with the new settings.
Symbols
_GXDLL 78

Numerics
1stGrid tutorial 183–209, ??–238

A
absolute cell coordinates 110, 111
    translating to logical 453
absolute cell reference 322, 331
AcceptCommasInNumbers() 136
Active Template Library. See ATL 460
ActiveX 5
AddNew() 32
AddRow() 433
AddToken() 130
AddUserProperty() 128
ADO browser 34
ADORecordset 436
AFX_MANAGE_STATE 371
AFX_MANAGE_STATE() 461
AFX_MODULE_STATE 368
AfxGetStaticModuleState() 371
AfxOleInit() 142, 473
application size
    reducing 155–181, 404
applications
    multithreaded 372
ApplyToObject() 482
AppWizard 150–??
arithmetic operators 328
ATL
    support in Objective Grid 459–461
automatic linking 297
Automation 486
autoscrolling 141
available documents 6
cell attribute inheritance 264
cell coordinates 110, 111
    converting to window coordinates 112
cell reference
    absolute 314, 322, 331
    computed 330
    indirect 332
    relative 331
cell tips 143
cell types 264
    registering 157
cell values 91
    clearing 93
    getting 93
    retrieving 92
cells
    copying 141
    definition 6
    definition of covered 6
    definition of current 7
    disabling 94
    floating 96
    merge 97–98
    moving 141
    read-only 94, 95
CentryValueHolder 449
CExcelExchange 489
CExcelRead 490
CExcelReadAdapter 471
CExcelWrite 489
CGridDSplitDoc 420
CGXAbstractControlFactory 63, 157
CGXAbstractUserAttribute 61
CGXADOEngine 436
CGXAdoGrid 34, 68
CGXAdoParam 69
CGXAdoRecordView 34, 69
CGXAdoRecordWnd 34, 69
CGXBitmapButton 51
CGXBitmapButtonChild 59
CGXBrowserParam 67
CGXBrowserView 67
CGXBrowserWnd 67
CGXDbParam 67
CGXDynamicRecordset 67
CGXODBCGrid 67
CGXRecordInfoBeam 68
CGXRecordInfoSplitterWnd 68
CGXRecordInfoWnd 68
CGXRecordView 68
CGXRecordViewWnd 68
CGXRecordWnd 68
CGXRecSetInfo 68
hierarchy diagram 66
BrwsGrid tutorial 413–417
build configurations 82
Build Wizard 161–162
built-in functions, enabling 320
C
CanAppend() 32
CancelRecord() 32
CanSelectCurrentCell() 27
CanUpdate() 32
CChildFrame 487

B
background color 483
    removing 484
base styles 99, 101
    definition 6
BIFF 8, structure of file 464
BiffView 464
browser
    ADO 34
    architecture 31–34
    generic 31
    ODBC 34
browser classes 66
    CGXAdoGrid 68
    CGXAdoParam 69
    CGXAdoRecordView 69
    CGXAdoRecordWnd 69
    CGXBrowseParam 67
    CGXBrowseGrid 67
    CGXBrowseView 67
    CGXBrowseWnd 67
    CGXDbParam 67
    CGXDynamicRecordset 67
    CGXODBCGrid 67
    CGXRecordInfoBeam 68
    CGXRecordInfoSplitterWnd 68
    CGXRecordInfoWnd 68
    CGXRecordView 68
    CGXRecordViewWnd 68
    CGXRecordWnd 68
    CGXRecSetInfo 68
    hierarchy diagram 66
BrwsGrid tutorial 413–417
build configurations 82
Build Wizard 161–162
built-in functions, enabling 320

C
CanAppend() 32
CancelRecord() 32
CanSelectCurrentCell() 27
CanUpdate() 32
CChildFrame 487

INDEX
CGXGridView 21, 48, 421, 424
drawing and updating with 20
CGXGridWnd 48
using Print and Print Preview
in conjunction with 388
CGXHeader 54
CGXHeaderFooterDialog 64, 130
CGXHierADOEngine 436
CGXHierDBGrid 435
CGXHierGridLayoutManager 43
5
CGXHotSpotEdit 54
CGX IntelliMousePlugin 74, 385
CGX ListBox 51, 55
CGXLongOperation 70, 125
 SetTicksContinued() 125
 SetTicksFirstTime() 125
CGXMaskControl 55
CGX MouseWheelPlugin 74
CGXMRTGrid 457
CGXNoOleDataObjectProxy 64
CGXNoOleDataSourceProxy 64
CGXODBCGrid 33, 34, 67
CGXPageBreakPlugin 427
CGXPasswordControl 55
CGXPen 61
CGX PluginComponent 74, 380
CGXPrintDevice 65
CGXPrintPropertiesDialog 64
CGXProgressCtrl 55
CGX Properties 63
 AddColorProperty() 128
 AddToken() 130
 AddUserProperty() 128
 and tokens 130
 GetColor() 128
 GetDataFooter() 130
 GetDataHeader() 130
 GetDistance() 130
 GetFirstPage() 130
 GetTokenArgs() 130
 GetUserProperty() 128
 SetColor() 128
 SetDistance() 130
 SetFirstPage() 130
 SetUserProperty() 128
 SubstTokenText() 130
CGXPushbutton 55
CGXRadioButton 56
cgxr buttonEx 56, 57
CGXRange 13, 14, 64
 SetCells() 14
 SetCols() 14
 SetRows() 14
 SetTable() 14
CGXRangeList 65
CGXRecordInfoSplitterWnd 68
CGXRecordInfoWnd 68
CGXRecordView 34, 68
CGXRecordWnd 34, 68
CGXRecSetInfo 68
CGXRegularGrid 435, 451, 453
CGXRichEditCtrl 57
CGX ScrollTip 71
CGXScrollTipPlugin 75
CGXSpinEdit 57
CGXSplitterWnd 71
CGXStatic 57
CGX Style 12, 13, 15, 19, 60, 482
 accessors 19
 attributes 15
 ChangeStyle() 19
 GetStyle() 93
 GetValue() 133
 interfaces 18
 SetEnable() 94
 SetFormat() 36, 135
 SetPlaces() 36, 304
 SetReadOnly() 94
 SetValue() 133
CGX StylesDialog 61
CGX StylesSheet 61, 102
CGX StylesMap 61, 102
CGX TabBeam 71
CGX TabbedComboBox 57, 58
CGX TabbedComboBoxWnd 58
CGX TabInfo 71
CGX TabWnd 71
CGX UserAttribute 61
CGX UserAttributePage 101
CGX View 48
CGX VScrollEdit 58
CGX WndWrapper 58
CGX WorksheetFunctions 72
CGX WrappedGrid 455
ChangeStyle() 19
gxCopy 19
gxExclude 19
gxOverride 19
character sets 140
charts
creating a pseudo-3D color chart 507
Chinese 395
circular reference 324
class hierarchy
browser classes 66
control classes 49, 264
data structures 62
drawing classes 47
formula engine 72
plug-in components 74
style classes 60
utility classes 70
class hierarchy
data structures 62
class summary 46
clipboard
formula support 307–308
Grid to Excel support 485
CMainFrame 420
color button
user interaction 500
color buttons
extended by GridEx 498
color palette, for Excel 483
columns
changing count 105
hidden 107
resizing 106
width 106, 112
command stage 21
common terms 7–8
components
reference 166–181
concatenation 329
concrete implementations
registering 158
conditional expression 325
conditional operators 330
constraint checking 325
constraint expression 325, 333
constraints 333
context menus
setting in a grid 504
control
architecture 26–30, 264
behavior, unpredictable 371
definition 6
factories 157
IDs, table 29
control child
definition 6
control child classes
CGXBitmapButtonChild 59
CGXButton 59
CGXChild 59
cell
absolute 111
class
transforming from absolute to logical 453
window 111
CopyCells() 22, 141
copying
cells 141
covered cells 96
definition 6
uncovering 96
CreateStyle() 94
creating
starter application 77–78
style objects on the heap 94
CSliderCtrl tutorial 405–411
CTabGridView 487
current cell
definition 7
value 93
CView
OnDraw() 20
cyclic dependency 324
D
data
fetching from databases 433
data objects 470
Excel 470
data source
definition 7
data structures 63
CGXAbstractControlFactory 63
CGXCommand 63
CGXControlFactory 63
CGXData 63
CGXDisplayPropertiesDialog 64
CGXGridDropTarget 64
CGXGridParam 63
CGXHeaderFooterDialog 64
CGXNoOleDataObjectProxy 64
CGXNoOleDataSourceProxy 64
CGXPrintDevice 65
CGXPrintPropertiesDialog 6
CGXProperties 63
CGXRange 64
CGXRangeList 65
hierarchy diagram 62
database engine 431–??
initializing 436
Databases
fetching data 433
Date and Time Control 167
spinner speed 167
dates 304
DBCS 395
DBEngine 432
fetching APIs 432
initializing APIs 432
updating APIs 433
DbQuery tutorial 211–??
DeleteRows() 32, 433
deriving a grid 117, 294
design objectives 261
design overview 11, 261
dialog boxes
prebuilt 126–131
using Objective Grid 79–81
digital logic function 335
direct interactions 302
disabled cells 94
distributing files 5
DlgGrid tutorial 239–250
DLLs 363
architecture 363–370
extension 367
documentation
available documents 6
formats 5, 6
DoLButtonDown() 27
DOMNodeType 445
DoRButtonDown() 28
drag and drop 141
drawing 20–25
gird lines 114
drawing classes 47
CGXGridCore 48
CGXGridHandleView 48
CGXGridHandleWnd 48
CGXGridHint 48
CGXGridView 48
CGXGridWnd 48
dwLength, incrementing 481
dynamic creation of style objects 94
dynamic splitter windows 419–425
E
Edit() 32, 433
embedded tools 338
EnableFormulaEngine() 302, 305
EnableLocaleNumbers() 137
EnableOleDbDataSource() 142
EnableOleDropTarget() 142
EnableScrollTips() 143
EnableTrackColWidth() 107
EnableTrackRowHeight() 107
END_SEC_XML_NEST() 448
escape codes 317
eTypeRWContext 449
events 115–116
Excel
Automation 486
benefits of Grid integration 491
classes, using 487
color palette 483
data objects 470
enabling Grid integration 487
fields, supported 483
handler 466
read/write mechanism 466
shared data objects 470
table 466
unshared data objects 470
Excel exchange classes 489
CChildFrame 487
CExcelExchange 489
CExcelRead 490
CExcelWrite 489
CTabGridView 487
explicit dependency 333
extending functionality 294
extension DLLs 367
F
Fetching incrementally 433
FetchRecords() 433
FindReplace 158
floating cells 96
formula engine 35–38, 72, 121, 133, 301
CGXDefaultWorksheetFunctions 72
CGXFormula 73
CGXFormulaSheet 72
CGXWorksheetFunctions 72
classes 72
formula expression, display of in cells 308
formula grids, initializing 398
formulas 327
supporting 121
forward reference 325
frozen columns 108
frozen rows 108
G
general settings 144
GetClientRow() 111
GetCol() 111
GetColCount() 23, 113, 403
GetColor() 128
GetColWidth() 23, 112
GetCurrentCell() 26
GetDataFooter() 130
GetDataHeader() 130
GetDefaultColWidth() 23
GetDefaultRowHeight() 23
GetDistances() 130
GetExpressionRowCol() 306
GetFieldCount() 433
GetFirstPage() 130
GetFrozenCols() 24, 111
GetFrozenRows() 24, 111
GetHeaderCols() 24
GetHeaderRows() 24
GetIncludeTextColor() 18
GetLeftCol() 111
GetRecordCount() 433
GetRow() 111
GetRowCount() 24, 113, 403
GetRowHeight() 24, 112
GetSheetContext() 305
GetStyle() 93
GetStyleRowCol() 24
GetTextColor() 18
GetTokenArgs() 130
GetTopRow() 111
GetUserProperty() 128
GetValue() 433
GetValueRowCol() 93
GetZoom() 24
glossary 6
goal-seeking 324
grid classes inheriting from 117
grid derivation 117
grid lines drawing 114
grid type, how to choose which 42–43
grid view updates 21
command stage 21
storing stage 21
update stage 21
GridDSplitView.h 420
GridEx
color button extension 498
context menus 504
creating a pseudo-3D color chart 507
Excel-like headers 505
Excel-like selection 501
MFC feature pack extension 493
source code and samples 494
tool tip extensions 496
grids changing colors 500
creating a pseudo-3D color chart 507
Excel-like headers 505
Excel-like selection 501
multiple row type 457
regular 453
setting context menus 504
transforming coordinates from absolute to logical 453
virtual 118–120
wrapped 455
GX_CELLHIT 113
GX_HEADERHIT 113
GX_HORIZLINE 113
GX_IDS_CTRL_BROWSEARROWHEADER 30
GX_IDS_CTRL_CBS_DROPPDOWN N 30, 53
GX_IDS_CTRL_CBS_DROPPDOWNNLIST 30, 53
GX_IDS_CTRL_CBS_TABBED_DROPPDOWN 30, 58
GX_IDS_CTRL_CBS_TABBED_DROPPDOWNNLIST 30, 58
GX_IDS_CTRL_CHECKBOX 30, 51
GX_IDS_CTRL_CHECKBOX_EX 51
GX_IDS_CTRL_CHECKBOX3D 30, 51
GX_IDS_CTRL_CHECKBOX3D_EX 51
GX_IDS_CTRL_CHECKLIST_COMBOBOX 30, 52
GX_IDS_CTRL_COMBOBOX 30, 52
GX_IDS_CTRL_CURRENCY 30, 53
GX_IDS_CTRL_DATETIME 30, 54
GX_IDS_CTRL_DATETIME_NOCAL 30, 54
GX_IDS_CTRL_EDIT 30, 54
GX_IDS_CTRL_HEADER 30, 54
GX_IDS_CTRL_HOTSPOT 30, 54
GX_IDS_CTRL_LISTBOX 30, 55
GX_IDS_CTRL_MASKEDIT 30, 55
GX_IDS_CTRL_ONEBASED 30, 52
GX_IDS_CTRL_ONEBASED_EX 30, 52
GX_IDS_CTRL_PASSWORD 30, 55
GX_IDS_CTRL_PROGRESS 30, 55
GX_IDS_CTRL_PUSHBTN 30, 55
GX_IDS_CTRL_RADIOBTN 30, 56
GX_IDS_CTRL_RADIOBTN_EX 56
GX_IDS_CTRL_RADIOBTN3D 30, 56
GX_IDS_CTRL_RADIOBTN3D_EX 56
GX_IDS_CTRL_RADIOBTN3DX 56
GX_IDS_CTRL_RADIOBTN3D_DX 56
GX_IDS_CTRL_RADIOBTN3DX 56
GX_IDS_CTRL_RADIOBTN3DEX 56
GX_IDS_CTRL_RICHEDIT 30, 57
GX_IDS_CTRL_SCROLLEDIT 30, 58
GX_IDS_CTRL_SPINEDIT 30, 57
GX_IDS_CTRL_STATIC 30, 57
GX_IDS_CTRL_TABBED_COMBOobox 30, 58
OBOX 30, 58
GX_IDS_CTRL_TEXTFIT 30, 52
GX_IDS_CTRLZEROBASED 30, 52
GX_IDS_CTRLZEROBASED_EX 30, 52
GX_IDS_UA_RADIOBUTTON_ALIGNMENT 56
GX_IDS_UA_RADIOBUTTONNTAXALIGN 56
gxall.h 78, 257
gxApplyNew 19
gxCopy 19
GXEnableWorksheetFunctions() 36, 121
gxExclude 19
GXForceTerminate() 370
GXInit() 78
GXInitDBEGridFromEngine 434
gxnMergeDelayEval 97
gxnMergeDisable 97
gxnMergeEvalOnDisplay 97
gxOverride 19
GXREADSEEKDATA 480
gxres.rc 78
gxresrc.h 78
GXSetNewGridLineMode() 114
GXTerminate() 370
GXWND 79, 81
handler 472, 478
headers in grid cells 505
Height_DPtoLP() 112
Height_LPtoDP() 112
hidden columns and rows 107
HideCols() 22, 108
HideRows() 22, 108
hit values 113
HitTest() 113, 403
hit-testing 113
HTML persistence 3
include bit 19
indirect interactions 302
indirect reference 332
inheritance 117
multiple 295
initialization
run-time 39–43
InsertCols() 22
InsertRows() 22
IntelliMouse 385
supporting 139
internationalization 395
IsCurrentCell() 26
IsEOPSeen() 433
IsReadOnly() 24
iterative calculation 324
IXMLRResolver 445
IXMLWResolver 447
Japanese 395
keyboard shortcuts 116

linking
automatic 297
linking Objective Grid libraries 297
locale settings 136
logic function 335
logical coordinates, translating to absolute 453
logical operators 328
long operations 125

m_bDirectCopyPaste 141
m_bDirectCutPaste 141
m_bDirectDragDrop 141
merge cells 97
message-map architecture 297
MFC feature pack extension 493
mfc42.dll 371
Microsoft Platform SDK 441
minimal recalculation 404
modes
  gxnMergeHorzOnly 97
gxnMergeVertOnly 97
modify types
  gxApplyNew 19
gxCopy 19
gxExclude 19
gxOverride 19
module state 368
MoveCells() 22, 141
MoveCols() 22
MoveCurrentCell() 26
MoveRows() 22
MoveTo() 32
moving
  cells 141
MRT grids 457
multiple inheritance 295
multiple-row-type grids 452, 457
multithreaded applications 372

N
named cell 321
named range 321
Named Range utility 321
naming a cell or range 321
naming conventions 4
natural order recalculation 323
numbers
  formatting 133–137

O
Objective Grid
  architecture overview 261–292
  ATL support 459–461
database features 431–433
design goals 261
features 1
XML support 443–444
Objective Grid Designer 143–150
  Cell Attributes 144
  General Settings 144
  Grid View 144
  tutorial 148
ODBC browser 33, 34
OLE drag and drop 141
OnAddedNewRecord() 32
OnCanceledEditing() 27
OnClickedButtonRowCol() 27
OnDeleteCell() 27
OnDrawItem() 25
OnDrawTopLeftBottomRight() 2
5
OnEndEditing() 27
OnFlushCellValue() 33
OnFlushRecord() 33
OnGetRecordCount() 33
OnGridDraw() 25
OnInitCurrentCell() 27
OnLButtonDownClickRowCol() 27
OnLButtonDownDblClickRowCol() 27
OnLButtonDownHitRowCol() 27
OnLeftCell() 27
OnLoadCellStyle() 33
OnMButtonDownClickRowCol() 27
OnMButtonDownDblClickRowCol() 27
OnMButtonDownHitRowCol() 27
OnModifyCell() 28
OnMovedCurrentCell() 28
OnPrepareDC() 111
OnPrintHeaderAndFooter() 130
OnRButtonDownClickRowCol() 28
OnRButtonDownDblClickRowCol() 28
OnRButtonDownHitRowCol() 28
OnStartEditing() 28
OnValidateCell() 28
opening Excel file, code 474
operators 328
  arithmetic 328
  conditional 330
  logical 328
text 328
optimization 397–404

P
page breaks
  adding to user interface 428
  supporting 427–430
parameter object 14
performance, improving 397–404
persistence
  using HTML 3
Platform SDK 441
plug-in components 74, 376
  CGXGridIntelliMousePlugin 75
  CGXGridMouseWheelPlugin 75
  CGXIntelliMousePlugin 74
  CGXMouseWheelPlugin 74
  CGXPluginComponent 74
  CGXScrollTipPlugin 75
predefined controls 30
Print and Print Preview
  functionality 388
Print.dll 388
printing, scaled 428
profiling 404
properties
  definition 7
PutDecInValue() 136
R
range 320
  definition 7
  specification 314
range objects 13–14
read only cells 94
read only grid 94
read/write mechanism 466
read/write mechanism, Excel 466
Reader, extending 478
ReadFromStream() 479, 480
ReadSeekHelper() 481
recalculation
  automatic 324
  frugal 324
  list 306
  minimal 324
  modes 323
  natural order 323
record identifier 465
record length 466
RecycleStyle() 94
Redo 121–125, 308
reducing
  application size 155–181
references
  returning 296
  regional settings 136
registering
  base styles 99
  cell types 157
controls 104–105
user attributes 101
window classes 159
regular DLLs 364
dynamically linked to MFC 364
regular grids 451, 453
relative cell coordinates 110, 111
relative reference 331
RemoveCols() 22
RemoveRows() 22
Requery() 33
ResetInternalState() 482
resizing
rows and columns 106
resource files 395
resources
gxres.rc 78
gxresrc.h 78
returning
references 296
reusing style objects 94
rotated text 140
rows
changing count 105
height 112
hidden 107
resizing 106
run-time initialization 39–43
S
samples
GridEx 494
on Rogue Wave Web site 4
saving to Excel file, code 474
scaled printing 428
scroll tips 143
SDK 441
SEC_XML_END() 448
SEC_XML_NEST() 448
SEC_XML_START() 448
selection
Excel-like with GridEx extensions 501
SelectRange() 22
SetAccelArrowKey() 141
SetAccelScrollbars() 141
SetAutoScroll() 141
SetCells() 14
SetColCount() 22, 106
SetColor() 128
SetCols() 14
SetColWidth() 22, 107
SetCurrentCell() 26
SetDaFTools() 139
SetDefaultColWidth() 23, 107
SetDefaultRowHeight() 23, 107
SetDistances() 130
SetEnable() 94
SetExpressionRowCol() 36, 304–305
SetFaceName() 140
SetFirstPage() 130
SetFormat() 36
SetFrozenCols() 23
SetFrozenRows() 23
SetGridLineStyle() 114
SetIncludeInterior() 12, 484
SetIncludeTextColor() 18
SetInterior() 12, 484
SetLockReadOnly() 94
SetMergeReadOnly() 94
SetMergeCellsMode() 97
gxnMergeDelayEval 97
gxnMergeDisable 97
gxnMergeEvalOnDisplay 97
gxnMergeHorzOnly 97
gxnMergeVertOnly 97
SetOrientation() 140
SetPlaces() 36
SetRangeName() 321
SetReadOnly() 23, 94
SetRowCount() 23
SetRowHeight() 23, 107
SetRows() 14
SetSortColsOnDbIClick() 125
SetSortRowsOnDbIClick() 125
SetStyleRange() 13, 14, 19, 23
SetTable() 14
SetTextColor() 19
settings
general 144
SetUserProperty() 128
SetValue() 433
SetValueType() 133
SetZoom() 23
shared DLL
Objective Grid as 155
significant digits 134
sorting 125
alphabetic 125
source code
GridEx 494
speed 397–404
splitting windows 419–425
Stingray AppWizard 150
StoreColWidth() 24
StoreCopyCells() 25
StoreDefaultColWidth() 24
StoreDefaultRowHeight() 24
StoreHideCol() 24
StoreHideRow() 24
StoreInsertCols() 24
StoreInsertRows() 25
StoreMoveCells() 25
StoreMoveCols() 25
StoreMoveRows() 25
StoreReadOnly() 25
StoreRemoveCols() 25
StoreRemoveRows() 25
StoreRowHeight() 25
StoreStyleRowCol() 25, 121
StoreZoom() 25
storing stage 21
string function 335
StripSeparatorFromValue() 136
struct member alignment 396
style
architecture 13, 15
creation on heap 94
definition 7
dynamic creation 94
include bit 19
object 13, 94
objects 94
objects, reusing 94
style classes 60
CGXAbstractUserAttribute 6
CGXBrush 61
CGXFont 61
CGXPen 61
CGXStyle 60
CGXStylesDialog 61
CGXStyleSheet 61
CGXStylesMap 61

516 Index
Index

CGXUserAttribute 61
hierarchy diagram 60
style objects 11–13
styles architecture 15–20
SubstTokenText() 130
supporting
page breaks 427–430

T
table 472, 478
technical overview 45
terms, common 6
text color, and setting font 485
text operators 328
time, represented as fraction of day 304
tokens
manipulating with CGXProperties methods 130
tool tips
extended by GridEx 496
transactions 122
and CGXGridCore 121
TransferCurrentCell() 26, 93
tutorials
1stGrid 183–209, ??–238
BrwsGrid 413–417
CSliderCtrl 405–411
DbQuery 211–??
DlgGrid 239–250
VirtGrid 251–255

U
Undo 121–125, 308
turning off 402
Undo/Redo 284
UndoRecord() 32
UNICODE 395
support in Excel import/export 485
update stage 21
Update() 32, 433
updating 20–25, 433
user attributes 101
user interface
adding page breaks 428
utility classes 70
CGXDrawingAndFormatting 70
CGXLongOperation 70

V
value types 133
GX_VT_ERROR 133
GX_VT_EXPRESSION 133
GX_VT_NUMERIC 133
GX_VT_STRING 133
VC++ 83
VirtGrid tutorial 251–255
virtual grids 118–120
virtual mode 402
visible-flag 107
Visual Studio version
compatibility 1

W
Width_DPtoLP() 112
Width_LPtoDP() 112
window classes
registering 159
window coordinates 110, 111
converting to cell coordinates 112
windows
dynamic splitter 419–425
Windows XP 84
WM_SETTINCHANGE 136
WM_WININICHANGE 136
workbook
definition 7
WorkBook Stream 464
worksheet
definition 7
WRAP(...) 448
wrapped grids 451, 455
WRITE_MAP 449
WriteToStream() 481

X
XML support 443–??
XML_ENTRY_MAP 448
XML_ENTRY_PARAM 449
XML_MAP() 448
xml_tool.exe 449