Programmer's Manual

List & Label® 26

Full-Powered Reporting Functionality

The information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. The availability of functions described in this manual depends on the version, the release level, the installed service packs and other features of your system (e.g. operating system, word processing software, email software etc.) as well as the general configuration. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of combit GmbH.

The license agreement can be found at www.combit.net and is displayed in the setup application.

JPEG coding and encoding is done with help of the JPEG Library of the IJG (Independent JPEG Group).

Avery and all Avery brands, product names and codes are trademarks of Avery Dennison Corporation.

PDF creation utilizes wPDF3 (c) wpCubed GmbH - www.pdfcontrol.com.

DataMatrix, MicroPDF417 and QRCode generation is done using components (c) J4L Components.

Aztec Barcode creation utilizes free code from Hand Held Inc.

Not all features are available in all editions. Please note the hints on LL_ERR_LICENSEVIOLATION.

Copyright © combit GmbH; Rev. 26.000
www.combit.com
All rights reserved.

Contents

1. Introduction 9
1.1 Before Installation 9
1.1.1 System Requirements 9
1.1.2 Licensing 9
1.2 After Installation 9
1.2.1 Start Menu 9
1.2.2 Designer Quick Start via Sample Application 10
1.2.3 Programming Samples 11
1.2.4 Documentation 11
1.3 Important Concepts 11
1.3.1 Basic Principles 11
1.3.2 Project Types 12
1.3.3 Variables and Fields 13
1.3.4 Available User Interface Languages 13
1.4 Getting Started With Programming 13
1.4.1 Overview 13
1.4.2 Integration With .NET 14
1.4.3 Integration With Visual Basic 14
1.4.4 Integration With Delphi 14
1.4.5 Integration With C++ Builder 14
1.4.6 Integration With C/C++ 14
1.4.7 Integration With Java 15
1.4.8 Integration With Other Programming Languages 15
1.4.9 Hints on Table, Variable and Field Names 15
1.4.10 Debugging Support 16

2. Programming With .NET 17
2.1 Introduction 17
2.1.1 Integration in Visual Studio 17
2.1.2 Components 18
2.2 First Steps 19
2.2.1 Integrate List & Label 19
2.2.2 License Component 20
2.2.3 Binding to a Data Source 21
2.2.4 Design 21
2.2.5 Print 22
2.2.6 Export 23
2.2.7 Important Properties of the Component 25
2.3 Other Important Concepts 25
2.3.1 Data Providers 25
2.3.2 Variables, Fields and Data Types 31
2.3.3 Events 34
2.3.4 Project Types 35

2.3.5 Varying Printers and Printing Copies ... 36
2.3.6 Edit and Extend the Designer ... 36
2.3.7 Objects in the Designer ... 38
2.3.8 Report Container ... 40
2.3.9 Object Model (DOM) ... 40
2.3.10 List & Label in WPF Applications ... 41
2.3.11 Error Handling With Exceptions ... 42
2.3.12 Debugging ... 42
2.3.13 Repository Mode for Distributed (Web) Applications ... 45
2.4 Usage in Web Applications ... 46
2.4.1 Web Reporting ... 47
2.4.2 Web Designer ... 48
2.4.3 HTML5 Viewer ... 52
2.5 Examples ... 53
2.5.1 Simple Label ... 54
2.5.2 Simple List ... 55
2.5.3 Invoice Merge ... 55
2.5.4 Print Card With Simple Placeholders ... 56
2.5.5 Sub Reports ... 57
2.5.6 Charts ... 58
2.5.7 Cross Tables ... 58
2.5.8 Database Independent Contents ... 58
2.5.9 Export ... 61
2.5.10 Extend Designer by Custom Function ... 62
2.5.11 Join and Convert Preview Files ... 64
2.5.12 Sending E-Mail ... 65
2.5.13 Store Project Files in a Database ... 66
2.5.14 Network Printing ... 66

3. Programming With the OCX Component ... 68
3.1 Integration of the Component ... 68
3.2 Simple Print and Design Methods ... 68
3.2.1 Working Principle ... 68
3.2.2 Using the UserData Parameter ... 69
3.3 Transferring Unbound Variables and Fields ... 69
3.3.1 Pictures ... 70
3.3.2 Barcodes ... 70
3.4 Language Selection ... 70
3.5 Working With Events ... 70
3.6 Displaying a Preview File ... 71
3.7 Working With Preview Files ... 71
3.7.1 Opening a Preview File ... 71
3.7.2 Merging Multiple Preview Files ... 71
3.7.3 Debugging ... 72
3.8 Extending the Designer ... 72
3.8.1 Using the Formula Wizard to Add Your Own Functions ... 72

3.8.2 Adding Your Own Objects to the Designer 74
3.9 The Viewer OCX Control 74
3.9.1 Overview 74
3.9.2 Registration 75
3.9.3 Properties 75
3.9.4 Methods 76
3.9.5 Events 77
3.9.6 Visual C++ Hint 78
3.9.7 CAB Files Packaging 79
3.9.8 Inserting the OCX Into Your Internet Page 79

4. Programming With the VCL Component 80
4.1 Integration of the Component 80
4.2 Data Binding 80
4.2.1 Binding List & Label to a Data Source 80
4.2.2 Working With Master Detail Records 81
4.2.3 Additional Options for Data Binding 82
4.3 Simple Print and Design Methods 83
4.3.1 Working Principle 83
4.3.2 Using the UserData Parameter 83
4.4 Transferring Unbound Variables and Fields 84
4.4.1 Pictures 84
4.4.2 Barcodes 84
4.5 Language Selection 85
4.6 Working With Events 85
4.7 Displaying a Preview File 85
4.8 Working With Preview Files 85
4.8.1 Opening a Preview File 85
4.8.2 Merging Multiple Preview Files 86
4.8.3 Debugging 86
4.9 Extending the Designer 86
4.9.1 Using the Formula Wizard to Add Your Own Functions 87
4.9.2 Adding Your Own Objects to the Designer 88

5. Programming Using the API 91
5.1 Programming Interface 91
5.1.1 Dynamic Link Libraries 91
5.1.2 General Notes About the Return Value 93
5.2 Programming Basics 93
5.2.1 Database Independent Concept 93
5.2.2 The List & Label Job 94
5.2.3 Variables, Fields and Data Types 94
5.3 Invoking the Designer 99
5.3.1 Basic Scheme 99
5.3.2 Annotations 100
5.4 The Print Process 102

5.4.1 Supplying Data ... 102
5.4.2 Real Data Preview or Print? ... 102
5.4.3 Basic Procedure ... 103
5.4.4 Annotations ... 106
5.5 Printing Relational Data ... 108
5.5.1 Using a Custom Print Loop ... 109
5.5.2 Using the ILLDataProvider Interface ... 116
5.5.3 Handling 1:1 Relations ... 126
5.6 Callbacks and Notifications ... 128
5.6.1 Overview ... 128
5.6.2 User Objects ... 129
5.6.3 Definition of a Callback Routine ... 130
5.6.4 Passing Data to the Callback Routine ... 130
5.6.5 Passing Data by Messages ... 131
5.6.6 Further Hints ... 132
5.7 Advanced Programming ... 132
5.7.1 Direct Print and Export From the Designer ... 133
5.7.2 Drilldown Reports in Preview ... 136
5.7.3 Supporting the Report Parameter Pane in Preview ... 140
5.7.4 Supporting Expandable Regions in Preview ... 141
5.7.5 Supporting Interactive Sorting in Preview ... 142
5.7.6 Handling Chart and Crosstab Objects ... 143
5.8 Using the DOM-API (Professional/Enterprise Edition Only) ... 145
5.8.1 Basic Principles ... 145
5.8.2 Examples ... 149

6. API Reference ... 153
6.1 Function Reference ... 153
6.2 Callback Reference ... 296
6.3 Managing Preview Files ... 322
6.3.1 Overview ... 322
6.3.2 The Preview API ... 322

7. The Export Modules ... 350
7.1 Overview ... 350
7.2 Programming Interface ... 350
7.2.1 Global (De)activation of the Export Modules ... 350
7.2.2 Switching Specific Export Modules On/Off ... 350
7.2.3 Selecting/Querying the Output Format ... 351
7.2.4 Setting Export-specific Options ... 352
7.2.5 Export Without User Interaction ... 353
7.2.6 Querying the Export Results ... 353
7.3 Programming Reference ... 354
7.3.1 PDF Export ... 354
7.3.2 Excel Export ... 358
7.3.3 Word Export ... 365

7.3.4 PowerPoint Export ... 370
7.3.5 RTF Export ... 373
7.3.6 XPS Export ... 378
7.3.7 XHTML/CSS Export ... 379
7.3.8 MHTML Export ... 386
7.3.9 JSON Export ... 386
7.3.10 Text (CSV) Export ... 388
7.3.11 Text (Layout) Export ... 390
7.3.12 XML Export ... 392
7.3.13 Picture Export ... 396
7.3.14 SVG Export ... 399
7.3.15 TTY Export ... 401
7.3.16 Windows Fax Export ... 402
7.3.17 Unsupported Export Formats ... 403
7.4 Digitally Sign Export Results ... 413
7.4.1 Start Signature ... 413
7.4.2 Programming Interface ... 414
7.5 Send Export Results via E-Mail ... 416
7.5.1 Overview ... 416
7.5.2 Setting Mail Parameters by Code ... 416
7.5.3 Sending Mail via 64 Bit Application ... 420
7.5.4 Hints for Selecting the MAPI Server ... 420
7.6 Export Files as ZIP Compressed Archive ... 421

8. Miscellaneous Programming Topics ... 423
8.1 Passing NULL Values ... 423
8.2 Rounding ... 423
8.3 Optimizing Speed ... 423
8.4 Project Parameters ... 424
8.4.1 Parameter Types ... 424
8.4.2 Querying Parameter Values While Printing ... 425
8.4.3 Predefined Project Parameters ... 425
8.4.4 Automatic Storage of Form Data ... 426
8.5 Web Reporting ... 428
8.6 Hints for Usage in Multiple Threads (Multithreading) ... 429
8.7 Scripting Support ... 429
8.7.1 Introduction ... 429
8.7.2 Preprocessor and Options ... 431
8.7.3 Quick Reference and Examples ... 434

9. Error Codes and Warnings ... 438
9.1 General Error Codes ... 438
9.2 General Warnings ... 441
9.3 Additional Error Codes of the Storage API ... 442
9.4 Additional Warnings of the Storage API ... 443

10. Debug Tool Debwin 444

11. Redistribution: Shipping the Application 445
11.1 System Requirements 445
11.2 The Standalone Viewer Application 445
11.2.1 Overview 445
11.2.2 Command Line Parameters 445
11.2.3 Registration 445
11.2.4 Necessary Files 446
11.3 List & Label Files 446
11.4 Web Designer Setup 448
11.4.1 Command Line Options for Windows Installer Setup 448
11.5 Other Settings 448

12. Update Information for Version 26 450
12.1 Overview of new Features 450
12.2 Overview of Changes 450
12.2.1 .NET 450
12.2.2 General 451
12.2.3 API 451
12.3 Updating to List & Label 26 451
12.3.1 General 451
12.3.2 Updating .NET Projects 452
12.3.3 Updating Projects Using the OCX (e.g. Visual Basic) 452
12.3.4 Updating Projects Using the VCL (e.g. Delphi) 452
12.3.5 Updating Projects Using the API (e.g. C/C++) 452

13. Help and Support 454

14. Index 455

1. Introduction

List & Label is a high-performance tool for printing reports, lists, labels and barcodes.

With this tool, you will have no problems enabling your programs to print professionally and with an attractive design.

1.1 Before Installation

1.1.1 System Requirements

List & Label is running under Windows 8.1 to Windows 10 (version 1803 - 20H2), as well as under Windows Server 2012 to 2019.

Note: Older versions which are no longer supported by the respective manufacturer ("end-of-life") may still be used, however combit does not assure that.

You also need a Windows standard printer driver with a resolution of at least 300 dpi even if you are not printing.

Some of the described functions (or the way these are accessed) depend on version, release, patch level etc. of your system/operating system and its configuration. Some functions may not be able to be used in all operating systems. You will find such limitations documented in the corresponding chapters.

If you want to use the Word (DOCX) export, the .NET Framework 4.7 is required on the developer as well as the end user machine.

1.1.2 Licensing

List & Label is being offered in various editions, which vary regarding features and licensing conditions. You will find a verbose description and comparison of the different editions and licensing conditions at https://www.combit.com/reporting-tool/faqs/.

1.2 After Installation

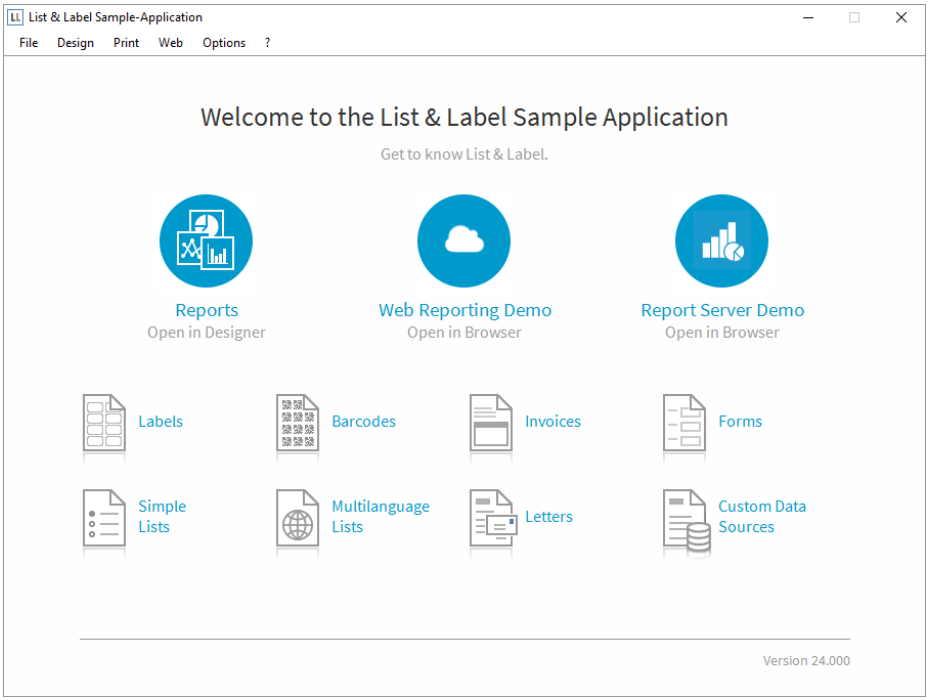
1.2.1 Start Menu

After installing List & Label, you will find the program group combit > combit List & Label 26 in the Windows start menu. This program group enables you to access all important information regarding integration, documentation and examples, as well as further useful tips and tricks. This group will be the starting point for the following chapters.

1.2.2 Designer Quick Start via Sample Application

A quick way to become acquainted with the Designer and its possibilities is to use the List & Label sample application. This standalone sample application is just for demonstration purposes and shows the various possibilities that the Designer offers. The data is taken from a fixed sample database.

You will find the application in the start menu group. It enables you to start the List & Label Designer immediately, and gain an overview of its functionality and flexibility through the wide variety of layout examples provided. The Designer is started by clicking *Design* from the menu and selecting an entry - e.g. invoice. Before the actual start, you can select an existing project file in the file selection dialog – or enter a new file name. The full functionality of the List & Label Designer – from the perspective of this sample application – is now available to you.



In addition, the List & Label sample application allows you to print existing or newly created projects using sample data records, or to use one of the export formats for output. Select one of the items in the *Print* menu. In the subsequent print options dialog, you can choose the output destination or export format.

1.2.3 Programming Samples

In order to ensure quick familiarization with the List & Label concept, a wide variety of programming examples are supplied with the installation. You will find these in the start menu group under "Samples".

You will find many different programming examples in the directories, depending on the installed development environment.

Further information on the individual examples as well as explanations on the methods and components used can be found in the List & Label Start Center, which is started directly after installation or is available in the start menu group.

1.2.4 Documentation

You will find all available documentation under the "Documentation" start menu group.

This includes the Programmers' Manual and the Designer Manual as PDF documents. You will also find various online help systems here, e.g. for the Designer or the List & Label components (.NET, VCL, OCX), as well as further information on redistribution, web reporting, debug etc.

1.3 Important Concepts

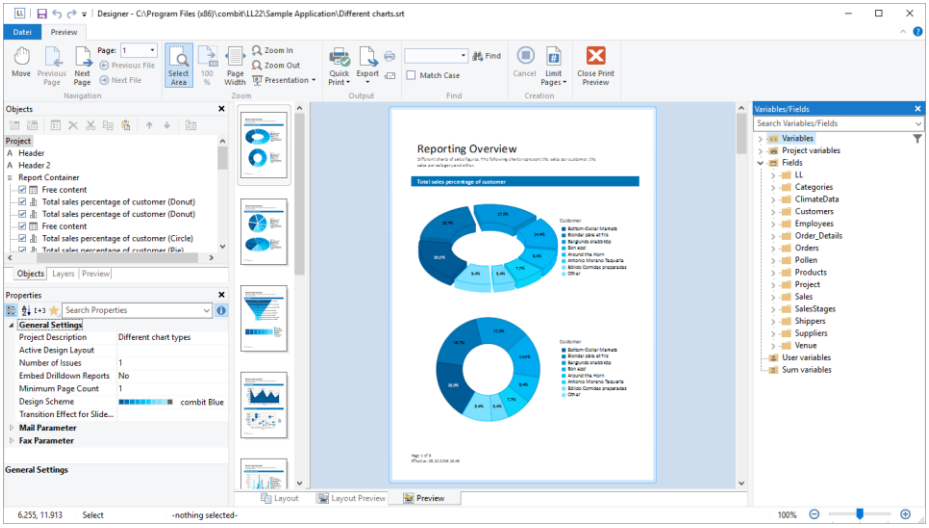
1.3.1 Basic Principles

List & Label is not a standalone application but a development component that is integrated into your application. With just a few lines of code you can enhance your application with reporting and printing capabilities of various kinds: Reports, subreports, lists, crosstabs, charts, diagrams, gauges, forms, labels, printing, print preview, export and web reporting.

Creating Report Templates in the Designer

The Designer functionality for interactive visual creation of reports, print templates etc. is an integrated part of the List & Label component and therefore will become part of your application. The Designer is not a standalone application, but it will be programmatically launched from your application. This is typically implemented within an event handler triggered by a menu item. The Designer will show up as a modal pop-up window overlapping you application window.

You can pass on this designing capability to your end users so that they can define individual templates or adapt the templates offered by you to personal requirements.



Print or Export: Generating Reports

To generate reports that have been designed by you or your end-users and send them to the printer or display them in the print-preview all data/records to be processed is being passed on to List & Label. Depending on the programming language this will either happen under the surface automatically when List & Label directly accesses your application's data by using specific data providers or it is being done explicitly by your source code, e.g. if your data is not stored in a database at all. A mixture between database data and application specific data is possible, too.

Besides sending the report to the printer or preview, List & Label offers various other output formats such as PDF, XHTML/CSS, XML, RTF, XLS, DOCX, TIFF, JPEG, PNG, plain text, bitmap and others. This is achieved by special export modules. From the developer's point of view there is no difference between printing and exporting the report.

Displaying Reports

The print preview can be used to display List & Label outputs automatically, save them to file, convert them to other formats, send them as email, and more. Additionally, it can be embedded via a separate component into your own dialogs resp. forms or HTML internet/intranet pages.

1.3.2 Project Types

List & Label can handle different project types: label and file card projects on the one hand, and list projects on the other.

Labels and File Cards

These projects consist of an arrangement of objects, which are printed once (file cards) or multiple times (in columns or rows, labels) per page.

Lists

Lists, on the other hand, consist of objects which are printed once per page, and one or more objects which are filled repeatedly with varying contents depending on the data records. The table, crosstab and the report container objects are responsible for these "repetitive areas" and therefore are only available in this mode.

1.3.3 Variables and Fields

List & Label distinguishes between two kinds of data fields: on the one hand there are data fields that are filled with content once per printed page (once per label or file card), these are called "variables" in the List & Label terminology. On the other hand, in a report, there are data fields that are filled repeatedly with different contents for a page, e.g. the data fields of an item list of an invoice. These data fields are called "fields" in the List & Label terminology.

For this reason, in file card or label projects only variables can be used, while in list projects both variables and fields can occur. For printing an invoice, an application would typically declare the invoice header data such as customer name and address as variables, while the item data such as article number, unit price, description etc. would be passed as fields.

1.3.4 Available User Interface Languages

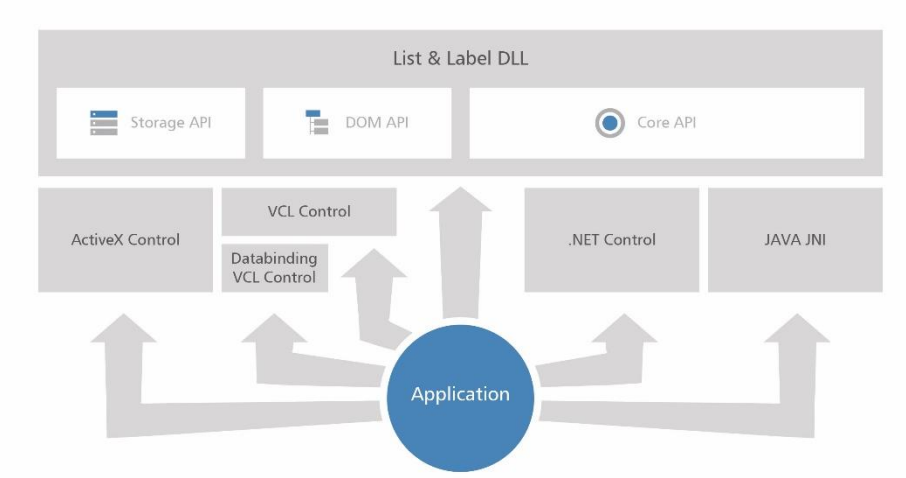
List & Label is available in several languages. Supplying additional languages is possible due to the modular architecture of List & Label. Besides the Designer, the printer, preview and export dialogs are localized by the language kits as long as they are not made up of common dialogs which are localized by the OS.

In order to integrate a language kit after purchase (the Enterprise Edition already includes all available language kits), use the corresponding language constant for *LlJobOpen()* or set the "Language" property to the desired value when using a component. Also supply your customers with the language files (cmll26??.lng, cmls26??.lng). List & Label expects the files to be in the same path as the main DLL cmll26.dll.

1.4 Getting Started With Programming

1.4.1 Overview

The following picture shows the basic structure of List & Label from a programmer's point of view:



1.4.2 Integration With .NET

Please refer to the section "Programming With .NET" for further information. Afterwards, we recommend further reading starting from section "The Export Modules", if applicable.

1.4.3 Integration With Visual Basic

For integration of List & Label with Visual Basic we recommend using the OCX/ActiveX component. Please refer to the OCX online help.

If you wish to use direct DLL access via API and no OCX component, add the file cmll26.bas, where you will find the necessary declarations. Please refer to the section "Programming Using the API" for further information in this case.

1.4.4 Integration With Delphi

Please refer to the online help for the VCL component's use.

1.4.5 Integration With C++ Builder

Please refer to the online help for the VCL component's use.

1.4.6 Integration With C/C++

Integration of List & Label with C/C++ is typically done by directly using the API. . Please refer to the section "Programming Using the API" for further information.

1.4.7 Integration With Java

Integrating List & Label into a Java application is done by adding the "combit"-package which is located in each of the provided programming samples for Java. Programming is done by calling the API directly. See chapter "Programming Using the API" for further information. Please also note that the provided Java Native Interface (JNI) Wrapper DLL has to be located in the List & Label search path. Further information can be found in chapter "Redistribution: Shipping the Application".

1.4.8 Integration With Other Programming Languages

For various programming languages, the installation of List & Label contains declaration files as well as examples. You will find these files in the corresponding sub-directories of your List & Label installation. Follow the documentation of your programming language to include DLL's via API or OCX/ActiveX components.

If your programming language is not included, you can create a declaration file on your own by following the appropriate steps for your programming language. Your programming language just needs to support invoking an API via calling DLL functions.

In case of doubt, please contact our support.

1.4.9 Hints on Table, Variable and Field Names

For table, variable and field names, the following restrictions apply:

- Variable and field names must be unique; you can't use the same descriptor for a variable and a field.
- Following characters are allowed for the descriptor:
    - First character: letter (of type UNICHAR_LETTER) or '_'
    - Following characters: letters (of type UNICHAR_LETTER), numbers (of type UNICHAR_NUMBER) as well as '.', ':', '$' and '_'
- For hierarchies and relations '.' and ':' can be used.

  The dot is the separator for the variable hierarchy. This way you can use e.g. "Person.Address.Street" and "Person.Address.City" as variable or field name. In the Designer you get a hierarchical structure, i.e. below "Person" you will find a folder "Address" with the fields "City" and "Street".

For relations see chapter "Handling 1:1 Relations".

There are two possibilities to optimize the performance of List & Label, as the internal work for processing the variable definition APIs and the formula parsing is depending on these:

- If the option LL_OPTION_XLATVARNAMES is set, the invalid characters will be replaced with '_' (please implicitly note that the descriptor of two variables

might become disambiguous). Without this option you have to take care of the correct variable names yourself, but the work for List & Label is decreased.

- List & Label is case sensitive if the option LL_OPTION_VARSCASESENSITIVE is set to TRUE, that also increases the performance.

1.4.10 Debugging Support

A significant part of the development process of an application is the detection and removal of bugs.

List & Label offers the possibility of logging all function calls in order to facilitate the removal of faults in your programs. All function calls including their parameters and the return value are shown on the debugging output. These can be displayed with the combit Debwin-Tool which is included in the package. See chapter "Debug Tool Debwin" for further details.

2. Programming With .NET

2.1 Introduction

For using List & Label with .NET several components are available, making the creation of reports on the .NET platform as easy as it can be. This tutorial shows the most important steps to work fast and productively with List & Label.

The complete programming interface is documented in the component help for .NET in detail. You will find it in the Documentation folder of your installation (combit.ListLabel26.chm).

2.1.1 Integration in Visual Studio

The List & Label .NET component is automatically integrated in Microsoft Visual Studio. For other programming environments or in case of a fresh installation of the development environment this can also be done manually. The components in form of an assembly are located in the directories "Samples\Microsoft .NET\" as well as "Redistributable Files\" of the List & Label installation. The integration is done as follows:

- Menu bar Tools > Choose Toolbox Items…
- Select tab .NET Framework Components
- Click button Browse...
- Select combit.ListLabel26.dll

Now the List & Label components can be dragged onto a form via Drag & Drop as usual. In the properties window the specific properties can be edited and event handlers can be added.

To integrate the List & Label .NET Help into the Visual Studio 2010 Help Viewer please follow these steps:

- Open Visual Studio 2010
- Select 'Help > Manage Help Settings' to start the Help Library Manager
- You might have to select a location for the local content at first. Confirm this dialog with 'OK'.
- Select the item 'Install content from disk'
- Click 'Browse' and navigate to the 'Documentation\Files' subdirectory of your List & Label installation
- Then select 'helpcontentsetup.msha' and click 'Open'
- Back in the Help Library Manager click 'Next'

- In the following dialog you will see the available help files including 'combit List & Label 26 - .NET Help'; select 'Add' here
- Now click 'Update' to integrate the help into the Help Viewer
- Click 'Yes' in the Security Alert dialog to confirm the digitally signed help file
- After updating the local library click 'Finish' to complete the integration of the help. Now you can use the List & Label .NET help by pressing F1 in Visual Studio at any time.

To remove the List & Label .NET Help from Visual Studio 2010 Help Viewer please follow the above steps and select 'Remove content' instead.

2.1.2 Components

In the tab "combit LL26" in the toolbox the following components can be found after the installation:

Components	Description
ListLabel	The most important component. All essential functions, such as print, design and export, are combined in it.
DataSource	A component that can be directly bound to a ListLabel instance as a data source. A description can be found in section " Data Provider".
DesignerControl	A component for displaying the Designer in custom forms.
ListLabelRTFControl	A RTF editor component for use in custom forms.
ListLabelPreviewControl	A preview control that can also be used in custom forms and supports the direct export to PDF for example. To perform a print into such a preview control, set the property AutoDestination to LlPrintMode.PreviewControl in the ListLabel component and select the desired preview control for the "PreviewControl" property.
ListLabelDocument	A descendant of PrintDocument. The built-in .NET preview classes can be used for displaying List & Label preview files with it.

2.2 First Steps

This paragraph guides you through the first steps that are required to integrate List & Label in your existing application.

2.2.1 Integrate List & Label

The .NET assemblies are available for .NET Framework 4.7, .NET Core 3.1 and .NET 5.

First a reference to the List & Label assembly has to be added to the project. If possible, references to the .NET assemblies should be added via the NuGet package manager. This ensures that all required dependencies are added as well. You can find our NuGet packages at https://www.nuget.org/profiles/combit.

Alternatively, the NuGet packages for offline use can be found under "Samples\Microsoft .NET\NuGet" of the installation.

Note: Please note that the List & Label NuGet packages and their dependencies may use Semantic Versioning 2.0.0, and therefore the following prerequisites apply:

- NuGet 4.3.0+

- Visual Studio 2017 version 15.3+

- Visual Studio 2015 with NuGet VSIX v3.6.0

- dotnet: dotnetcore.exe (.NET SDK 2.0.0+)

The assemblies themselves are located in the respective "Assemblies" subdirectory under "Samples\Microsoft .NET\" of the installation.

In the second step an instance of the component can be created. This can be done either by the development environment directly by dragging the ListLabel component onto a form. Alternatively, the component can also be created dynamically:

C#:

```
combit.Reporting.ListLabel LL = new combit.Reporting.ListLabel();
```

VB.NET:

```
Dim LL As New combit.Reporting.ListLabel()
```

Generally the namespaces combit.Reporting and combit.Reporting.DataProviders are prereferenced for the whole file, depending on the programming language by "using" (C#) or "Imports" (VB.NET). This saves a lot of typing later.

C#:

```
using combit.Reporting;
```

```
using combit.Reporting.DataProviders;
```

VB.NET:

```
Imports combit.Reporting
Imports combit.Reporting.DataProviders
```

When using dynamic creation, the component should be released by the Dispose method after its use, so that the unmanaged resources can be released as soon as possible.

C#:

```
LL.Dispose();
```

VB.NET:

```
LL.Dispose()
```

Due to performance reasons it is recommended that you always keep an instance of the ListLabel object in memory globally for the dynamic creation as well. It can be created in the Load event of the application's main window and released again in the FormClosed event for example. The essential advantage is that the List & Label modules won't be loaded and released for every new instance, which can lead to undesirable delays of frequent calls or e.g. multiple prints.

2.2.2 License Component

During the installation of the full version (not the Trial version) a file with the personal license and support information is created in the start menu. To successfully use List & Label on end user machines it is mandatory to pass the license key included in this file to all instances of the ListLabel object. The object provides the property LicensingInfo for that. A sample call could be as following:

C#:

```
LL.LicensingInfo = "A83jHd";
```

VB.NET:

```
LL.LicensingInfo = "A83jHd"
```

where the part in quotes must be replaced by the license code in the text file.

Please note: for the trial version, an empty string should be used as license code.

Note: In a web application, use the WebDesignerConfig.LicensingInfo property.

2.2.3 Binding to a Data Source

For design and print List & Label has to have knowledge of a data source. An overview about the available data sources can be found in the section "2.3.1 Data Providers". Of course additional unbound, custom data can be passed, too. An example for that can be found in "2.5.8 Database Independent Contents".

To bind List & Label to the data source the component provides the property DataSource. The binding can be done either interactively in the development environment by using the property windows or the SmartTags of the component or alternatively on code level:

C#:

```
LL.DataSource = CreateDataSet();
```

VB.NET:

```
LL.DataSource = CreateDataSet()
```

The CreateDataSet() routine in this sample is a placeholder for a method of your application which prepares the DataSet required for the report.

2.2.4 Design

The Designer is called by the Design method and will be displayed as a modal pop-up window that overlaps your application window. A data source always has to be assigned beforehand. This is the basis for the data available in the Designer. Therefore there is no stand-alone design application; the data is always provided directly by the application, List & Label itself never directly accesses the data.

The full call – with a DataSet as data source in this example – would be:

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();
LL.Design();
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()
LL.Design()
LL.Dispose()
```

By default a file selection dialog is displayed to the user, where he can either provide a new name for the report file and therefore create a new report or select an existing file for editing. Of course this can also be suppressed – the section "Important Properties of the Component" describes how to do that.

Using the Designer itself is explained in detail in the corresponding online help and in the Designer manual. The result of the design process is generally four files that are created by the Designer. The file extensions can be assigned freely by using the FileExtensions property of the ListLabel component. The following table describes the files for the default case.

File	Content
<Reportname>.lst	The actual project file. It contains information about the formatting of the data to print, but not the data itself.
< Reportname >.lsv	A JPEG file with a sketch/thumbnail of the project for display in the file selection dialog.
< Reportname >.lsp	File with user-specific printer and export settings. This file should not be redistributed if the design computer is not identical with the print computer as the printer stored in the file usually does not exist.
< Reportname >.~lst	Is created as soon as the project is saved for the second time within the Designer and contains a backup of the project file.

The most important file is of course the project file. The other files will automatically be created by List & Label at application runtime.

At print time the actual report is created by the combination of project file and data source. In practice it is often also desired to keep the project files in a central database. How this is done is described in section "Store Project Files in a Database".

2.2.5 Print

The print is called by the method Print(). A project file for the data structure of the selected data source must first be created in the Designer. It is easiest to bind the component to the same data source at print and design time. So the preview in the Designer displays the correct data and the user can easily visualize the result at runtime. A full call of the print would be:

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();
LL.Print();
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()
LL.Print()
LL.Dispose()
```

By default a file selection dialog is displayed at first, followed by a print options dialog. The section "Important Properties of the Component" describes how these can be avoided or be prefilled if desired.

2.2.6 Export

Export means the output to one of the supported output formats like PDF, HTML, RTF, XLS, etc. The code for starting an export is identical with a print, in the print options dialog the user can choose any export format besides the "normal" output formats Printer, File and Preview. If a format is to be preselected by default, the print start could be as follows:

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();
LL.ExportOptions.Add(LlExportOption.ExportTarget, "PDF");
LL.Print();
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()
LL.ExportOptions.Add(LlExportOption.ExportTarget, "PDF")
LL.Print()
LL.Dispose()
```

The available export targets are listed in the following table:

Export target	Value for ExportTarget
Printer	PRN
Preview	PRV
Adobe PDF Format	PDF
XHTML/CSS Format	XHTML
Multi-Mime HTML Format	MHTML
Microsoft Excel Format	XLS
Microsoft Word Format	DOCX
Microsoft XPS Format	XPS
Rich Text Format (RTF)	RTF
Multi-TIFF Picture	PICTURE_MULTITIFF
TIFF Picture	PICTURE_TIFF
PNG Picture	PICTURE_PNG
JPEG Picture	PICTURE_JPEG
Bitmap Picture	PICTURE_BMP
Metafile Picture (EMF)	PICTURE_EMF
File	FILE
Pinwriter (TTY)	TTY
Microsoft PowerPoint Format	PPTX
Text (CSV) Format	TXT
Text (Layout) Format	TXT_LAYOUT
XML Format	XML

The following export targets are not supported anymore and are only available for compatibility reasons. If you still want to use the format you have to enable it explicitly via LlSetOption(hJob, LL_OPTIONSTR_LEGACY_EXPORTERS_ALLOWED,...) or via LL.Core.LlSetOptionString(...).

Export target	Value for ExportTarget
HTML Format	HTML
HTML jQuery Mobile Format	JQM

The other options (e.g. font embedding, encryption, etc.) can also be preset with default values directly from code. This is done, as in the example above, by using the ExportOptions class; the LlExportOption enumeration contains all supported options as values.

Most frequently these are required to execute a "silent" export. It is more convenient to use the Export() method of the component. Please see the Export sample for C# and VB.NET.

2.2.7 Important Properties of the Component

The behavior of print, design and export can be controlled by some of the component's properties of the component. The most important are listed in the following table:

Property	Function
AutoProjectFile	Name of the project file to use. This is the default name for the project if a file selection dialog is provided. Otherwise it is the name of the project to use (Default: empty).
AutoDestination	Output format. If desired a format can be forced for the user by this property, e.g. Print is only allowed to printer or preview (Default: LIPrintMode.Export). If a selection of export formats is to be allowed, it can be done by setting LIOptionString.ExportsAllowed. An example can be found in section "Restriction of Export Formats".
AutoFileAlsoNew	Sets if the user is allowed to use a new file name for design in order to create a new project (Default: true).
AutoProjectType	Sets the project type. The different project types are described in section "Project Types" (Default: LIProject.List).
AutoShowPrintOptions	Sets if the print options dialog is displayed or suppressed (Default: true, display).
AutoShowSelectFile	Sets if the file selection dialog is displayed or suppressed (Default: true, display).
AutoMasterMode	Used together with the DataMember property to pass the master/parent table of 1:n linked data structures as variables. An example can be found in section "Variables, Fields and Data".

2.3 Other Important Concepts

2.3.1 Data Providers

Providing data in List & Label is done with data providers. These are classes that implement the interface IDataProvider from the combit.Reporting.DataProviders namespace. Within this namespace a lot of classes are already contained which can

act as a data provider. A detailed class reference can be found in the .NET component help.

For data formats that are apparently not directly supported, a suitable provider is found in most cases anyway. Business data from applications can generally be passed through the object data provider. If the data is present in comma-separated form, the data provider from the "Dataprovider" sample can be used. Many other data sources support the serialization to XML, so that the XmlDataProvider can be used. If only a small amount of additional information is to be passed, it is possible to do it directly. A sample is shown in paragraph "Database Independent Contents".

Once List & Label is bound to a DataProvider, it supports the following features automatically if applicable to the data source:

- real data preview in the Designer
- report container and relational data structure
- sortings
- drilldown

The following overview lists the most important classes and their supported data sources.

AdoDataProvider

Offers access to data of the following ADO.NET elements:

- DataView
- DataTable
- DataViewManager
- DataSet

The provider can be assigned implicitly by setting the DataSource property to an instance of any of the supported classes. Of course the provider can also be explicitly assigned.

This data provider automatically supports a single level sorting by any field, ascending or descending.

Example:

C#:

```
ListLabel LL = new ListLabel();
AdoDataProvider provider = new AdoDataProvider(CreateDataSet());
LL.DataSource = provider;
LL.Print();
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
```

```
Dim provider As New AdoDataProvider(CreateDataSet())
LL.DataSource = provider
LL.Print()
LL.Dispose()
```

DataProviderCollection

This data provider can be used to combine multiple other data providers into one data source. Use it if you have e.g. multiple DataSet classes from where to pull data or if you would like to have a mix of XML and custom object data. Example:

This provider supports the same sortings that the providers in the collection do support.

C#:

```
DataSet ds1 = CreateDataSet();
DataSet ds2 = CreateOtherDataSet();

// combine the data from ds1 and ds2 into one datasource
DataProviderCollection providerCollection = new DataProviderCollection();
providerCollection.Add(new AdoDataProvider(ds1));
providerCollection.Add(new AdoDataProvider(ds2));
ListLabel LL = new ListLabel();
LL.DataSource = providerCollection;
LL.Design();
LL.Dispose();
```

VB.NET:

```
Dim ds1 As DataSet = CreateDataSet()
Dim ds2 As DataSet = CreateOtherDataSet()

' combine the data from ds1 and ds2 into one datasource
Dim providerCollection As New DataProviderCollection()
providerCollection.Add(New AdoDataProvider(ds1))
providerCollection.Add(New AdoDataProvider(ds2))
Dim LL As New ListLabel()
LL.DataSource = providerCollection
LL.Design()
LL.Dispose()
```

DataSource

This data provider is in an exceptional position because it can be inserted as a component directly from the toolbox. The component provides a few properties, which are also available through the SmartTags. The most important property is “ConnectionProperties”. By using the corresponding property editor, a connection

string can be directly created in the development environment that provides access to following data sources:

- Microsoft Access
- ODBC data sources (e.g. Excel data)
- Microsoft SQL-Server (also file based)
- Oracle databases

Once configured the data source is available in the selection window for the DataSource of the ListLabel component and can therefore be directly assigned. By clicking the link "Open report designer…" in the SmartTags of the ListLabel component the Designer can also be directly opened from within the development environment, requiring not a single line of code to access the data of a DataSource.

This data provider automatically supports a single level sorting by any field, ascending or descending.

DbCommandSetDataProvider

Allows combination of multiple IDbCommand implementations into one data source. It can be used e.g. to access multiple SQL tables and define relations between them. Another possibility is to combine data from e.g. SQL and Oracle databases into one data source.

This data provider automatically supports a single level sorting by any field, ascending or descending.

ObjectDataProvider

This data provider can be used to access object structures. It can work with the following types/interfaces:

- IEnumerable (requires at least one record though)
- IEnumerable<T>
- IListSource

In order to influence the property names and types, you may either implement the ITypedList interface on your class or use the DisplayNameAttribute. To suppress members, use the Browsable(false) attribute on the members.

The provider can also parse empty enumerations as long as they are strongly typed. Otherwise, at least one element is required in the enumeration and this first element determines the type that is used for further parsing.

The provider automatically supports sorting as soon as the data source implements the IBindingList interface.

You may also use this data provider to access LINQ query results, as they are IEnumerable<T>.

When using EntityCollection<T> objects as data source the ObjectDataProvider first checks the state of the sub relation by the IsLoaded property and dynamically calls Load() if necessary. The data is provided when needed with it. Example:

C#:

```
class Car
{
    public string Brand { get; set; }
    public string Model { get; set; }
}

List<Car> cars = new List<Car>();
cars.Add(new Car { Brand = "VW", Model = "Passat"});
cars.Add(new Car { Brand = "Porsche", Model = "Cayenne"});
ListLabel LL = new ListLabel();
LL.DataSource = new ObjectDataProvider(cars);
LL.Design();
LL.Dispose();
```

VB.NET:

```
Public Class Car
    Dim _brand As String
    Dim _model As String

    Public Property Brand() As String
    Get
        Return _brand
    End Get
    Set(ByVal value As String)
        _brand = value
    End Set
    End Property
    Public Property Model() As String
    Get
        Return _model
    End Get
    Set(ByVal value As String)
        _model = value
    End Set
    End Property
End Class

Dim LL As New ListLabel
Dim Cars As New List(Of Car)()
Dim Car As New Car
Car.Model = "Passat"
Car.Brand = "VW"
Cars.Add(Car)
Car = New Car
Car.Model = "Cayenne"
Car.Brand = "Porsche"
Cars.Add(Car)
LL.DataSource = New ObjectDataProvider(Cars)
```

```
LL.AutoProjectType = LlProject.List
LL.Design()
LL.Dispose()
```

OleDbConnectionDataProvider

Allows binding to an OleDbConnection (e.g. Access database file). This data provider automatically supports a single level sorting by any field, ascending or descending.

Example:

C#:

```
OleDbConnection conn = new
OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" +
DatabasePath);
OleDbConnectionDataProvider provider = new
OleDbConnectionDataProvider(conn);
ListLabel LL = new ListLabel();
LL.DataSource = provider;
LL.Design();
LL.Dispose();
```

VB.NET:

```
Dim conn As New OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=" + DatabasePath)
Dim provider As New OleDbConnectionDataProvider(conn)
Dim LL As New ListLabel()
LL.DataSource = provider
LL.Design()
LL.Dispose()
```

OracleConnectionDataProvider

Allows binding to an OracleConnection. This data provider automatically supports a single level sorting by any field, ascending or descending.

SqlConnectionDataProvider

Allows binding to a SqlConnection. This data provider automatically supports a single level sorting by any field, ascending or descending.

Example:

C#:

```
SqlConnection conn = new
SqlConnection(Properties.Settings.Default.ConnectionString);
SqlConnectionDataProvider provider = new SqlConnectionDataProvider(conn);
ListLabel LL = new ListLabel();
```

```
LL.DataSource = provider;
LL.Design();
LL.Dispose();
```

VB.NET:

```
Dim conn As New SqlConnection(Properties.Settings.Default.ConnectionString)
Dim provider As New SqlConnectionDataProvider(conn)
Dim LL As New ListLabel()
LL.DataSource = provider
LL.Design()
LL.Dispose()
```

XmlDataProvider

Allows accessing XML data files easily. No schema information in XML/XSD files will be used and no constraints will be handled. The main purpose of this class is to provide a fast and easy access to nested XML data. This data provider does not support any sorting. Example:

C#:

```
XmlDataProvider provider = new XmlDataProvider(@"c:\users\public\data.xml");
ListLabel LL = new ListLabel();
LL.DataSource = provider;
LL.Design();
LL.Dispose();
```

VB.NET:

```
Dim provider As New XmlDataProvider("c:\users\public\data.xml")
Dim LL As New ListLabel()
LL.DataSource = provider
LL.Design()
LL.Dispose()
```

2.3.2 Variables, Fields and Data Types

Variables and fields are the dynamic text blocks for reports and contain the dynamic part of the data. Variables usually change once per page or report – an example is the header data of an invoice with invoice number and addressee. Fields on the other hand usually change for every record; a typical example would be the item data of an invoice.

Within the Designer variables are always offered outside of the report container (the "table area"), fields only inside of it, and can only be used there. The separation serves mainly to help the end user. If he were to place a field in the "outside area", the result

would -- depending on the print order -- either be the content of the first or the last record.

Both identifier types (fields and variables) can be ordered hierarchically and are displayed in a folder structure in the Designer. The database table names are automatically added by the databinding so that all data from the "OrderData" table are displayed in a folder "OrderData".

Custom data can also be ordered hierarchically by using a dot as a hierarchy separator (e.g. "AdditionalData.UserName"). How to add custom data can be found in the section "Database Independent Contents".

Variables and Fields With Databinding

Consider the case of a 1:n linked data structure such as "InvoiceHeader" or "InvoiceItems". The header data should usually be declared as variables whereas the actual invoice items should be declared as fields. The properties DataMember and AutoMasterMode can be used to achieve this:

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();

// Order data as variables
LL.DataMember = "InvoiceHeader";
LL.AutoMasterMode = LlAutoMasterMode.AsVariables;

LL.Design();
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()

' Order data as variables
LL.DataMember = "InvoiceHead"
LL.AutoMasterMode = LlAutoMasterMode.AsVariables

LL.Design()
LL.Dispose()
```

At print time a merge print is automatically generated, if e.g. an invoice form has been designed, a single invoice with its own page numbering, aggregation, etc. is created for each record from the invoice head table.

Effect of the option AutoMasterMode. Left: "AsVariables", right: "AsFields".

Data Types

Variables and fields are passed in typed form, i.e. depending on the content of the database as text, number, etc. The databinding usually cares for that automatically, an explicit passing/assignment of the type is only necessary when custom data is passed additionally. The correct data type is usually also preselected (e.g. when passing DateTime objects).

The following table shows the most important data types.

Data type	Usage
LIFieldType.Text	Text.
LIFieldType.RTF	RTF formatted text. This field type can be used in a RTF field or RTF object in the Designer.
LIFieldType.Numeric LIFieldType.Numeric_Integer	Integer. The databinding automatically differs between floating point numbers and integer values.
LIFieldType.Boolean	Logical values.
LIFieldType.Date	Date and time values (DateTime).
LIFieldType.Drawing	Drawing. Generally, the file name is passed. Directly passing a memory handle is possible for Bitmaps and EMF files. Databinding automatically checks the content of Byte fields and declares them as drawing if a suitable format is found.
LIFieldType,Barcode	Barcode. Barcodes are most easily passed as instances of the LIBarcode class directly in the

Data type	Usage
	Add methods of the Variables and fields property.
LlFieldType.HTML	HTML. The content of the variable is a valid HTML stream, a file name or an URL.

2.3.3 Events

The following table shows some important events of the ListLabel component. A full reference can be found in the component help for .NET.

Event	Usage
AutoDefineField/ AutoDefineVariable	These events are called for each field or variable before passing it to List & Label. With the event arguments you can manipulate the name and content or completely prevent the declaration of the element. Examples can be found in section "Database Independent Contents".
AutoDefineNewPage	This event is triggered for every new page when using databinding. Here, you can register additional required page-specific variables for the application which are not part of the data source by using LL.Variables.Add(). Examples can be found in section "Database Independent Contents".
AutoDefineNewLine	The event is triggered for every new line. If the application requires additional line-specific data which is not part of the data source itself, it can be added in this event by using LL.Fields.Add(). Examples can be found in section "Database Independent Contents".
DrawObject DrawPage DrawTableLine DrawTableField	These events are each called once before and after printing the corresponding elements, e.g. for each table cell (DrawTableField). The event arguments contain a Graphics object and the output rectangle so that the application can output custom information additionally. That could be a special shading, a "Trial" character or a complete specific output.
VariableHelpText	You can support your users by displaying help texts for each variable and field in the Designer.

2.3.4 Project Types

Three different modes of the Designer are available as report type. Which mode is used depends on the value of the property AutoProjectType.

Lists

This is the default and matches the value LlProject.List for the AutoProjectType property.

Typical fields of use are invoices, address lists, reports with charts and cross tables, multi-column lists, briefly all types of reports where a tabular element is required. The report container is only available in this mode (see section "2.3.8 Report Container").

Labels

This project type matches the value LlProject.Label for the AutoProjectType property.

It is used for the output of labels. As there are no tabular areas and also no report container, only variables and no fields are available (see section "Variables, Fields and Data").

If the used data source contains multiple tables, e.g. products, customers, etc. the source table for printing labels can be selected by the DataMember property of the component:

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();

// Products as data source
LL.DataMember = "Products";

// Select label as project type
LL.AutoProjectType = LlProject.Label;

LL.Design();
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()

' Products as data source
LL.DataMember = "Products"

' Select label as project type
LL.AutoProjectType = LlProject.Label

LL.Design()
```

```
LL.Dispose()
```

Cards

This project type matches the value LlProject.Card of the AutoProjectType property.

Card projects are a special case of label projects with exactly one page-filling label. Typical fields of use are printing file cards (e.g. all customer information at a glance) or mail merges. Activation is the same as "Labels", the same hints and restrictions apply therefore.

2.3.5 Varying Printers and Printing Copies

List & Label offers a comfortable support of splitting a report to different printers or the output of copies with a "Copy" watermark. The best is that these are all pure Designer features that are automatically supported by List & Label.

Regions

The regions' purpose is to split the project into multiple page regions with different properties. Typical fields of use are e.g. different printers for first page, following pages and last page. Further applications are mixing Portrait and landscape format within the same report.

You can see a demonstration e.g. in the List & Label Sample Application (in the start menu's root level) under Design > Extended Samples > Mixed portrait and landscape.

Issues and Copies

Both issues' and copies' purpose is to output multiple copies of the reports. Copies are "real" hardware copies, meaning that the printer is assigned to create multiple copies of the output. Of course all copies are identical and will be created with the same printer settings.

If the output is to have different properties (e.g. Original from tray 1, copy from tray 2) or a "Copy" watermark is to be output, issues are the way to go. The property "Number of Issues" in the Designer has to be set to a value greater than one. Then the function "IssueIndex" is available for all regions, so that a region with the condition "IssueIndex()==1" (Original) and another with the condition "IssueIndex()==2" (Copy) can be created.

The objects in the Designer get a new property "Display Condition for Issue Print" with which the printing of a watermark can be realized in a similar way.

You can see a demonstration e.g. in the List & Label Sample Application (in the start menu's root level) under Design > Invoice > Invoice with issue print.

2.3.6 Edit and Extend the Designer

The Designer is not a "Black Box" for the application, but can be manipulated in many ways. Besides disabling functions and menu items, user-specific elements can be added that move calculations, actions or outputs to the function logic.

Menu Items, Objects and Functions

Starting point for Designer restrictions is the DesignerWorkspace property of the ListLabel object. In the following table the properties listed can be used to restrict the Designer.

Property	Function
ProhibitedActions	This property's purpose is to remove single menu items from the Designer.
ProhibitedFunctions	This property's purpose is to remove single functions from the Designer.
ReadOnlyObjects	This property's purpose is to prevent objects' editability in the Designer. The objects are still visible; however they can't be edited or deleted within the Designer.

The following example shows how the Designer can be adjusted so no new project can be created. In addition the function "ProjectPath$" will be removed and the object "Demo" is prevented from being edited.

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();

// Restrict Designer
LL.DesignerWorkspace.ProhibitedActions.Add(LlDesignerAction.FileNew);
LL.DesignerWorkspace.ProhibitedFunctions.Add("ProjectPath$");
LL.DesignerWorkspace.ReadOnlyObjects.Add("Demo");

LL.Design();
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()

' Restrict Designer
LL.DesignerWorkspace.ProhibitedActions.Add(LlDesignerAction.FileNew)
LL.DesignerWorkspace.ProhibitedFunctions.Add("ProjectPath$")
LL.DesignerWorkspace.ReadOnlyObjects.Add("Demo")

LL.Design()
LL.Dispose()
```

Extend Designer

The Designer can be extended by user-specific functions, objects and actions.

User-specific functions can be used to move more complex calculations to the application or add functions which are not covered by the Designer by default.

An example for adding a new function can be found in the section "Extend Designer by Custom Function".

Examples for user-specific objects or actions as well as another user-specific function can be found in the "Designer Extension" Sample" for C# and VB.NET.

2.3.7 Objects in the Designer

Some objects in the Designer only serve for graphical design (e.g. Line, Rectangle, Ellipse). However, most of the other objects interact with the provided data. Specific data types are available for that or there are conversion functions which allow converting contents for use in the corresponding object. The following paragraphs give an overview of the most frequently used objects, their corresponding data types and Designer functions for converting contents.

The hints for the single objects are also valid in the same or similar way for (Picture, Barcode, etc.) columns in table elements.

Text

A text objects consists of multiple paragraphs. Each of these paragraphs does have specific content. This can be either a variable or a formula which combines multiple data contents. To display single variables, usually no special conversion is required. If multiple variables of a different type (see section "Data Types") are to be combined within a formula, the single parts have to be converted to the same data types (e.g. string). An example for the combination of numbers and strings would be:

```
"Total: "+Str$(Sum(Article.Price),0,2)
```

The following table lists some of the conversion functions, which are frequently needed in this context.

From / To	Date	Number	Picture	Barcode	Text
Date	-	DateToJulian()	-	-	Date$()
Number	JulianToDate()	-	-	Barcode(Str$())	FStr$() Str$()
Picture	-	-	-	-	Drawing$()
Barcode	-	Val(Barcode$())		-	Barcode$()
Text	Date()	Val()	Drawing()	Barcode()	-

Picture

The content of a picture object is set via the property window. The property **Data Source** offers three values **File Name**, **Formula** and **Variable**.

- The setting **File Name** is used for a fixed file, such as a company logo. If the file is not supposed to be redistributed it can be embedded in the report itself. The file selection dialog offers the appropriate option.
- With **Formula,** the content can be set by a string containing a path. The required function is "Drawing".
- With **Variable,** contents already passed as a picture can be displayed (see section "Data Types").

Barcode

The content of a barcode object is set in a dialog. This dialog offers the three options **Text**, **Formula** and **Variable** for the data source.

- The setting **Text** is used for fixed text/content in the barcode. In addition to the content, the type – e.g. with 2D-Barcodes – and other properties for error correction or encoding can be set.
- With **Formula** the content can be set by a string which contains the barcode content. The required function is "Barcode".
- With **Variable** contents already passed as a barcode can be displayed (see section "Data Types").

RTF Text

The content of a RTF-Text object is set in a dialog. This dialog offers the options **(Free Text)** or a selection of possible passed RTF variables (see below) under **Source**

- The setting **(Free Text)** is used for fixed text/content in the RTF object. Within the object, data content can be used at any position (e.g. for personalized multiple letters) by clicking the formula icon in the toolbar.
- By selecting a variable, contents already passed as RTF can be displayed (see section "Data Types").

HTML

The content of an HTML object is set in a dialog. This dialog offers the three options **File**, **URL** and **Formula** as a data source.

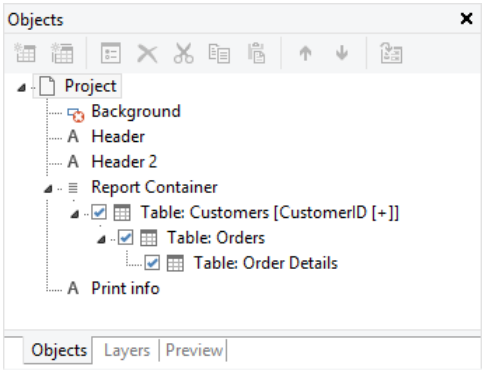
- The setting **File** is used for a fixed HTML file.
- With **URL,** a URL can be passed from where the HTML content should be downloaded.
- With **Formula,** contents already passed as an HTML stream can be displayed (see section "Data Types").

Also see the hint for "HTML Formatted Text" in section "Variables, Fields and Data Types".

2.3.8 Report Container

The report container is the central element of list projects. It allows displaying tabular data (also multi-columnar or nested), statistics and charts as well as cross tables. Data can also be output in different form – e.g. at first for a graphical analysis of the sales by years and then in a detailed tabular list.

The contents of the container are visible underneath the Report Container object in the "Objects" tool window. Using this window, new content can be added or existing content can be edited. The window is a sort of "screenplay" for the report since the exact order of the single report elements is shown in it.



The Objects tool window in the Designer

To make the report container available, a data provider (see section "Data Providers") has to be used as data source. Generally, it is also possible to perform the complete printing on your own by using the low-level API functions of the LICore object, however this is not the recommended practice since many features (Designer preview, Drilldown, report container, …) would have to be specially supported. If in doubt, it makes more sense to write your own data provider. See section "Database Independent Contents".

All provided list samples make use of the report container and therefore provide demonstration material for the different operation purposes.

A detailed description for using this element can be found in the Designer Manual under section "Inserting Report Container".

2.3.9 Object Model (DOM)

Whereas the Designer is providing a very comfortable and powerful interface for editing project files, it can often be desired to set object or report properties per code. For example, the application can display a dialog prior to the Designer with a data preselection and then start the Designer with a project already prepared with this selection. An example for that can be found in the "Simple DOM" sample for C# and VB.NET.

Access to the object model is only available from the Professional Edition and higher.

The following table lists the most important classes and properties of the namespace combit.Reporting.Dom.

Class	Function
ProjectList ProjectLabel ProjectCard	The actual project classes. These represent the root element of the project. Key methods are Open, Save and Close.
<Project>.Objects	A list of all objects within the project. The objects are descendants of ObjectBase and each object contains its own properties and enumerations (e.g. Text paragraphs).
<Project>.Regions	An enumeration of the layout regions of the project. A page dependent printer control can be realized this way for example. Further information can be found in section "Regions".
ObjectText	Represents a text object. Key property is Paragraphs, the actual content of the text.
ObjectReportContainer	Represents a report container. Key property is SubItems, the actual content of the report container.
SubItemTable	Represents a table within the report container. It consists of different line regions (Lines property), which have different columns (Columns property of a line).

Within the single classes, the properties can easily be browsed via IntelliSense. A complete reference of all classes can be found in the component help for .NET.

2.3.10 List & Label in WPF Applications

As List & Label itself is a non-visual component, it can be used in WPF applications as well as in WinForms applications. The Designer itself is no WPF window, however this does not affect its functionality. The WPF Viewer can be used for displaying preview files and is a replacement for the WinForms PreviewControl.

Please note that the WPF viewer relies on the existence of the XPS Document Writer printer driver on the target system. If the driver is not available, documents cannot be displayed.

2.3.11 Error Handling With Exceptions

List & Label defines a number of internal exceptions, which are all derived from the common base class ListLabelException and therefore can be caught at a central location. If the application is supposed to carry out its own Exception handling, calls to List & Label can be enclosed by an Exception handler:

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();

try
{
    LL.Design();
}
catch (ListLabelException ex)
{
    MessageBox.Show(ex.Message);
}
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()

Try
       LL.Design()
Catch ex As ListLabelException
       MessageBox.Show(ex.Message)
End Try
LL.Dispose()
```

The Message property of the Exception class contains an error text, which – if a corresponding language kit is present – is also generally localized and can be displayed directly to the user.

A complete reference of all Exception classes can be found in the component help for .NET.

2.3.12 Debugging

Problems occurring on the developer PC can be easily found in most cases. The usual features of the development environment can be used to spot a problem relatively quickly. The first step is to catch any occurring exceptions and to find their cause (see section "Error Handling With Exceptions").

As a development component List & Label is naturally run under a variety of different constellations on the end user side. To find problems there as easily as possible a

dedicated debug tool is available which provides a logging function for problems occurring rarely or only on certain systems so problems can also be examined under systems without a debugger.

Of course the logging function can also be used on the developer PC and provides the possibility to check all calls and return values at a glance as well.

Create Log File

If a problem only occurs on a customer system, the first thing to do is to create a log file. The tool Debwin can be used for this purpose. It can be found in the "Tools" directory of the List & Label installation.

Debwin has to be started before the application. If the application is started afterwards, all calls of the component with their return values as well as additional information about module versions, operation system, etc. will be logged.

Every exception thrown under .NET represents a negative value of a function in the log. There is usually more helpful information in the log.

If the application is supposed to create debug logs without the help of Debwin, this can be done in the configuration file of the application. Logging can be forced as follows:

```
<configuration>
 <appSettings>
  <add key="ListLabel DebugLogFilePath" value="C:\Users\Public\debug.log" />
  <add key="ListLabel EnableDebug" value="1" />
 </appSettings>
</configuration>
```

Custom Logging Mechanisms & Logs in Web Applications

Capturing log messages from List & Label using Debwin and the integrated log file writer are simple and convenient solutions for regular desktop applications.

However, for web applications, Windows services and multi-user systems this approach is of a very limited usefulness: Debwin and the integrated log file capture the log messages from all List & Label instances of a process and there is no separation between the outputs of concurrently running print jobs.

For those scenarios implementing a custom logging mechanism or using one of the popular logging frameworks like NLog or log4net becomes advisable. For that purpose, create a new class deriving from *LoggerBase* or implementing the *ILlLogger* interface directly. Passing such a logger object to the constructor of the ListLabel object will cause all log ouputs of this specific List & Label job to be sent to the specified logger. You may then filter the messages by different levels (Debug, Information, Warning and Error) and categories (e.g. data provider, .NET component, printer information, …).

The included sample project "C# Custom Logger Sample" contains examples of an own logger implementation as well as adapters which connect the prevalent logging frameworks NLog and log4net with the logging interface of List & Label.

Example: Redirecting log output to NLog:

```
ILogger   nlogLogger = NLog.LogManager.GetLogger("MyApp.Reporting");
ILlLogger llLogger = new ListLabel2NLogAdapter(nlogLogger);
ListLabel LL = new ListLabel(llLogger);
```

Please note:

- The properties "Debug" and "DebugLogFilePath" of the ListLabel class are ignored as soon as a custom logger is passed to the constructor.
- To avoid a strong decrease in performance, reduce the log outputs in your *ILlLogger* implementation to the minimum using the WantOutput() method.
- Most of the included data providers (optionally) also support an external logger object. Those data providers implement the *ISupportsLogger* interface and provide a SetLogger() method. If a data provider is not assigned a custom logger object, it inherits the logger of the ListLabel object.

Tip for NLog: Often log outputs appear as sudden bursts. Use the AsyncWrapper target of NLog for an async processing of the log messages so List & Label does not have to wait on it.

2.3.13 Repository Mode for Distributed (Web) Applications

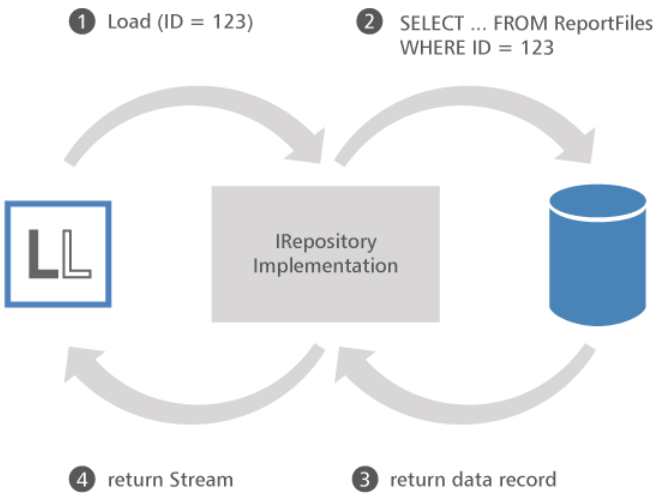
If reports are to be used in distributed applications such as web applications, all required files need to be shared between the systems involved or between the client and the server and kept synchronized at all times. Hence, it is a good idea to save the project files in a central database. However, this solution can become rather complex, especially when a project references pictures, drilldown projects, and other external files via *local* data paths, which then also need to be valid on another system.

With repository mode, the use of local files in a project can be done away with entirely, and List & Label projects as well as all the files they require can be managed at a central location (the so-called repository) with little effort—such as in a database or by a web service.

Basic Principles

In repository mode, List & Label does not save and load the files used in a report on its own. Instead of file paths and names, unique repository IDs are used. You will need to implement the *IRepository* interface yourself and pass it to the *ListLabel* object. From this point onwards, List & Label will query your user-defined repository for the file content belonging to a repository ID, or transmit the ID along with the corresponding file content to the repository to be saved. Whether the files in the repository are managed by an SQL database, a web service, or some other storage solution depends entirely on your *IRepository* implementation. In this case, loading and saving of entries in the repository takes place exclusively via streams.

The following schematic diagram shows how a report with the ID "123" is loaded via an IRepository implementation managing an internal SQL database:



Implementation

All functions to which the file path of the project was previously passed are now assigned the repository ID instead:

```
// Without repository mode:
LL.Design(LlProject.List, "C:\Reports\Invoice.lst")

// With repository mode – also applies to Export() and/or Print():
LL.FileRepository = new MyCustomRepository(…);
LL.Design(LlProject.List, "repository://{53F875F0-6177-8AD5-01B44E3A9867}")
```

In repository mode, "files" become "repository items", and filenames become "repository IDs". In addition to its ID, each repository item possesses a type, a time stamp, and a descriptor (string with variable length which contains internal information). Hence, in addition to the file content, your repository implementation will also need to be able to save and retrieve at least these four pieces of information for each repository item.

You will find a simple repository implementation in the ASP.NET sample projects (class *SQLiteFileRepository*) which provides a repository with an SQLite database for data storage.
For detailed descriptions of the methods to be implemented in the *IRepository* interface, please refer to the .NET online help (combit.ListLabel26.chm).

Tips

- Use the class *RepositoryImportUtil* to import existing local files and/or create new projects in the repository.
- You can use the *RepositoryItemDescriptor* class to change the display name shown in the selection dialogs in the Designer instead of the internal repository ID.
- Accessing the repository takes place sequentially. However, if you use the same repository object for multiple ListLabel instances, or if the data storage used is not thread-safe, then synchronization is required.

2.4 Usage in Web Applications

List & Label can also be used within web applications, albeit with a number of limitations. For ASP.NET-based web applications, List & Label contains components for displaying reports in the web browser, which also support complex features such as drill-down, as well as an independent version of the Designer with which reports can be designed on the server with the familiar List & Label Designer (Web Designer).

2.4.1 Web Reporting

Printing within a web application is basically the same as an export e.g. to PDF format, where all dialogs are suppressed. How this is done in general is described in section "Export Without User Interaction". After the report is generated that way, the browser of the user can be directed to the created file by the usual mechanisms.

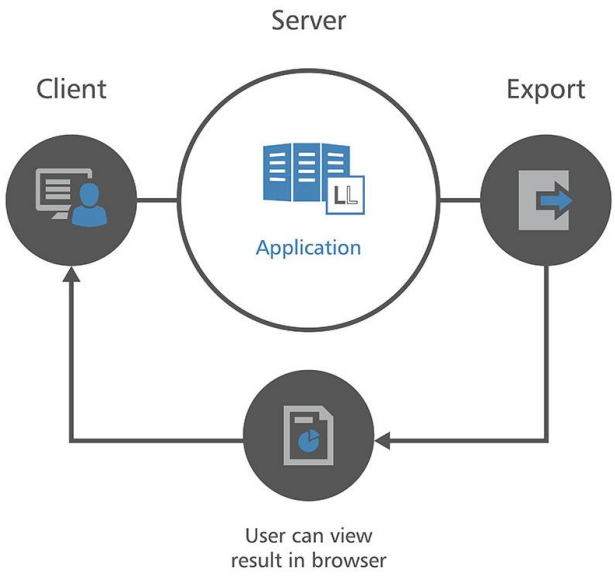
Generally, the project files are created within a client application and then published together with the web application. For editing projects on the server, the Web Designer is available. The provided web reporting sample for C# and VB.NET shows how to do this.

Also, on the server at least one printer driver has to be installed, so that printing can be successful. List & Label closely works together with the Windows GDI and therefore requires a printer device context for all operations. The Microsoft XPS printer driver that is included in Windows might also be used as a reference printer.

Important:

As the process/service is executed in the context of the IIS, it is important that the used ApplicationPool is configured accordingly to have access to the installed printer driver. From IIS versions 7.0/7.5 on (from Windows 7/Server 2008 R2 on), however, the standard identity is the 'ApplicationPoolIdentity'. This is a so-called virtual account, that might not have access to the system resources like the printer driver. If printer access cannot be granted to the Application Pool identity, the defined identity must be reconfigured to e.g. 'NetworkService' or a normal user account.

The following image visualizes the principle:



2.4.2 Web Designer

The Web Designer is a special browser-independent version of the Designer which can be used to design reports on a web server. It is installed as a regular Windows program on the client. In this way, the Web Designer does not appear as a part of the website, but instead as an independent desktop program which is launched from your website.

Basic Concepts

- The List & Label Web Designer needs to be distributed together with your own application. To do so, you will have to store the setup file at a location (of your own choosing) on the web server. The users of the Designer will then download it and install it on their local computer. The Designer launches as an independent program outside the web browser!
- You will need to place *DesignerControl* on a website under ASP.NET WebForms or call the command *@Html.ListLabelMvcWebDesigner()* under ASP.NET MVC from a view. This ensures that the web browser performing the call launches the Web Designer or prompts the user to install it.
- On the server, you will only need to set a few parameters and add and assign the List & Label data provider as usual. The Web Designer will internally ensure that the data from the data source on the web server is forwarded to the Designer on the client for the real-time data preview.

Integration

To integrate the Web Designer into your application, proceed as follows. For more details, please refer to the application examples provided for ASP.NET.

Step 1: Add Web Designer to Application

Copy the setup file of the Web Designer ("LL26WebDesignerSetup.exe") to any one of the web application's folders so that clients are able to download it from there. The setup file is located in the List & Label installation folder under "Redistributable files".

When switching to a new List & Label version (also for service packs), the setup file for the Web Designer on the server will need to be updated. The clients will then automatically prompt the user to perform an update the next time the application is launched.

Finally, add a reference to the assembly "combit.ListLabel26.Web.dll". How the Web Designer call is inserted differs according to the web framework used, you will find the necessary code in the programming examples in the List & Label installation folder under "Samples\Microsoft .NET\ASP.NET":

Under ASP.NET WebForms:

```
…\C# WebDesigner and AjaxViewer\Designer.aspx
```

Under ASP.NET MVC:

```
…\C# MVC Web Reporting Sample\MVC Web Reporting
Sample\Views\Sample\WebDesignerLauncher.cshtml
…\C# MVC Web Reporting Sample\MVC Web Reporting
Sample\Controllers\SampleController.cs
```

Step 2: Configuring Data Provider for Real-Time Data Preview

Assign your data provider to the ListLabel object and/or the DataSource property of the WebDesignerOptions object as usual. No direct connection is created between the Web Designer and the data source; the Web Designer communicates with your web application exclusively via HTTP(S), wherein a server-side module of the Web Designer forwards the data from your data source to the Web Designer. Hence, no additional adjustments are necessary on the data source side (or for your database server), such as having to enable connections in your firewall.

The OnRequestDataProvider event from List & Label versions earlier than 22 is no longer required.

Step 3: Expand the Application Configuration

The new Web Designer always uses ASP.NET MVC for the internal processing of HTTP requests. Hence, the necessary routes will need to be registered in the *Application_Start*() function of your application (*Global.asax* file), even if your application is not based on ASP.NET MVC. Add the following lines:

```
WebDesignerConfig.RegisterRoutes(RouteTable.Routes);
WebDesignerConfig.WebDesignerSetupFile =
Server.MapPath("~/Webdesigner/LL26WebDesignerSetup.exe");
```

Important: If you are also using MVC/Web API, the *WebDesignerConfig.RegisterRoutes* function must be registered before your own routes.

The second instruction specifies the local data path of the Web Designer setup which is downloaded if the client has not yet installed it. Distribute the Web Designer setup together with your web application and modify this link accordingly.

The basic setup of the Web Designer is now completed.

Step 4 (Recommended): Enable Drilldown, Table of Contents etc.

By default, all features of the (Desktop) Designer requiring additional files (apart from the main project) are disabled in the Web Designer. This is done because it cannot be guaranteed that the selected local data paths will be valid both on the client and on the server. For example, this applies to drilldown projects, project includes, pictures and PDF files that are not embedded etc.

These limitations do not apply when the repository mode of List & Label is used, with which the files from one or more reports are managed by a simple, virtual file system - the repository.

How to use the List & Label repository mode is described in chapter "Repository Mode for Distributed (Web) Applications". Using List & Label to retroactively convert a web application to repository mode can be rather time-consuming and complicated. Therefore, it is recommended that repository mode be used with new web applications from the very beginning despite the higher level of complexity - even if the features available without a repository initially satisfy your requirements. Hence, the included ASP.NET examples all use the repository.

Limitations in the Web Designer

General limitations in the Web Designer as compared to the normal Designer:

- no interactive formula functions (AskString$, AskStringChoice$, LibraryPath$, ProjectPath$, ApplicationPath$)

- no possibility to use local file paths or formulas as source for HTML objects
- no OLE objects
- no fax/mail features
- no custom DesignerActions
- no custom-added variables that change their value for each page
- no objects drawn using callback

There are also additional restrictions in the Web Designer if the repository mode is not used:

- no drilldown
- no project includes
- no report sections (*.toc, *.gtc, *.idx)
- pictures and PDF files must be embedded in the project file
- no external shapefiles

Troubleshooting

Older versions of IIS may not work properly with the MVC framework. If this happens, try adding the following entries to the *web.config* file under <system.webServer> \ <handlers >:

```
<remove name="ExtensionlessUrlHandler-ISAPI-4.0_32bit" />
<remove name="ExtensionlessUrlHandler-ISAPI-4.0_64bit" />
<remove name="ExtensionlessUrlHandler-Integrated-4.0" />
<add name="ExtensionlessUrlHandler-ISAPI-4.0_32bit" path="*."
verb="GET,HEAD,POST,DEBUG,PUT,DELETE,PATCH,OPTIONS" modules="IsapiModule"
scriptProcessor="%windir%\Microsoft.NET\Framework\v4.0.30319\aspnet_isapi.dl
l" preCondition="classicMode,runtimeVersionv4.0,bitness32"
responseBufferLimit="0" />
<add name="ExtensionlessUrlHandler-ISAPI-4.0_64bit" path="*."
verb="GET,HEAD,POST,DEBUG,PUT,DELETE,PATCH,OPTIONS" modules="IsapiModule"
scriptProcessor="%windir%\Microsoft.NET\Framework64\v4.0.30319\aspnet_isapi.
dll" preCondition="classicMode,runtimeVersionv4.0,bitness64"
responseBufferLimit="0" />
<add name="ExtensionlessUrlHandler-Integrated-4.0" path="*."
verb="GET,HEAD,POST,DEBUG,PUT,DELETE,PATCH,OPTIONS"
type="System.Web.Handlers.TransferRequestHandler"
preCondition="integratedMode,runtimeVersionv4.0" />
```

Additional Information

By default, the Web Designer communicates via URLs in the following format:
/WebDesigner/…
/WebDesignerHandler/…

If these routes are already being used by your own application, you may use the optional parameter of the *WebDesignerConfig.RegisterRoutes()* command to assign the Web Designer other routes.

2.4.3 HTML5 Viewer

The HTML5 Viewer is available in ASP.NET WebForms and ASP.NET MVC applications and supports all popular browsers. Therefore, it is applicable on clients with different operating systems. It allows interactive input through Drilldown and report parameters.

Basic Concepts

In ASP.NET WebForms you place the Html5ViewerControl on the website. In ASP.NET MVC you can use the HtmlHelp extension method.

On the server side you set the DataSource, the project file, the temp path and some optional properties in an event that is called by the HTML5 Viewer.

Integration

To integrate the HTML5 Viewer into your application proceed as follows. Further details can be found in the provided application samples for ASP.NET.

Step 1: Add HTML5 Viewer to your application

Under ASP.NET WebForms:

In an ASP.NET WebForms application just add the Html5ViewerControl from the combit.Reporting.Web namespace of the combit.ListLabel26.Web.dll. Please also see the example in the List & Label installation directory under "Samples\Microsoft .NET\ASP.NET\C# Web Reporting Sample\ HTML5Viewer.aspx".

```
<%@ Register TagPrefix="combit" Namespace="combit.Reporting.Web"
Assembly="combit.ListLabel26.Web" %>
…
<combit:Html5ViewerControl ID="Html5ViewerControl1"
runat="server"></combit:Html5ViewerControl>
```

Under ASP.NET MVC:

```
@using combit.Reporting.Web
@Html.Html5Viewer((string)ViewBag.InstanceName, new Html5ViewerOptions {}
);
```

Please also see the example in the List & Label installation directory under "Samples\Microsoft .NET\ASP.NET\C# MVC Web Reporting Sample\MVC Web Reporting Sample\Views\Home\ HTML5Viewer.cshtml".

Step 2: Setup project file and other data on the server side

In the file global.asax.cs the routes and the event handler for the HTML5 Viewer must be registered.

```
protected void Application_Start(Object sender, EventArgs e)
{
   Html5ViewerConfig.RegisterRoutes(RouteTable.Routes);
...Html5ViewerConfig.OnListLabelRequest += new
EventHandler<ListLabelRequestEventArgs>(Services_OnListLabelRequest);
```

Important: If you use MVC/Web API yourself, the Html5ViewerConfig.RegisterRoutes() function must be registered before your own routes.

In the event handler the data required for the HTML5 Viewer will be passed.

```
void Services_OnListLabelRequest(object sender, ListLabelRequestEventArgs e)
{
   ListLabel ll = new ListLabel();
   ll.DataSource =
DataAccess.SampleData.CreateProviderCollection(reportName);
   ll.AutoProjectFile = _reportsPath + e.ReportName;
   e.ExportPath = Path.GetTempPath();  // set temp directory;
    e.NewInstance = ll;
}
```

The name of the selected report can be set in ASP.NET WebForms by using the ReportName property of the Html5ViewerControl object. In ASP.NET MVC it can directly passed as a parameter of the HtmlHelper extension method @Html.Html5Viewer(). In the OnListLabelRequest event it is then available again under "e.ReportName", so that the corresponding data source and project file can be loaded.

Options

The Html5ViewerOptions class that you assign to the Options property of the Html5ViewerControl or pass as a parameter of the HtmlHelper extension method @Html.Html5Viewer() enables you to define where required jQuery and jQuery Mobile files should be loaded from (CDNType), which CSS files should be loaded (ThemeCSSUrl, ThemeIconsCSSUrl) or which jQuery Mobile theme should be used (DataTheme). For more information about these formatting options please see the jQuery Mobile help available online.

2.5 Examples

The examples in this paragraph show how some typical tasks can be solved. The code can serve as a template for your own implementations.

For the sake of clarity, the usual error handling is omitted here. All exceptions will be caught directly in the development environment. For "real" applications we recommend exception handling as described in section "Error Handling With Exceptions".

2.5.1 Simple Label

To print a label, use the project type LlProject.Label. If a data source with multiple tables, such as a DataSet, is connected, the desired data for the label can be selected by the DataMember property.

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();

// Products as data source
LL.DataMember = "Products";

// Select label as project type
LL.AutoProjectType = LlProject.Label;

// Call Designer
LL.Design();

// Print
LL.Print();
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()

' Products as data source
LL.DataMember = "Products"

' Select label as project type
LL.AutoProjectType = LlProject.Label

' Call Designer
LL.Design()

' Print
LL.Print()
LL.Dispose()
```

2.5.2 Simple List

Print and design of simple lists is the "default" and can be started with just a few lines of code:

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();

// Call Designer
LL.Design();

// Print
LL.Print();
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()

' Call Designer
LL.Design()

' Print
LL.Print()
LL.Dispose()
```

2.5.3 Invoice Merge

An invoice merge is an implicit merge print. The head or parent data contains one record for each document which is linked 1:n with the detail or child data. To design and to print such a document the parent table has to be passed to List & Label by the DataMember property. Furthermore the AutoMasterMode property has to be set to AsVariables as shown in the following example:

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();

// Order data as variables
LL.DataMember = "InvoiceHeader";
LL.AutoMasterMode = LlAutoMasterMode.AsVariables;

// Call Designer
LL.Design();
```

```
// Print
LL.Print();
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()

' Order data as variables
LL.DataMember = "Invoice head"
LL.AutoMasterMode = LlAutoMasterMode.AsVariables

' Call Designer
LL.Design()

' Print
LL.Print()
LL.Dispose();
```

2.5.4 Print Card With Simple Placeholders

Printing a full-page project which simply contains placeholders set by the application is achieved easily by binding it to a suitable object:

C#:

```
public class DataSource
{
    public string Text1 { get; set; }
    public double Number1 { get; set; }
    ...
}

// Prepare data source
object dataSource = new DataSource { Text1 = "Test", Number1 = 1.234 };
ListLabel LL = new ListLabel();
LL.DataSource = new ObjectDataProvider(dataSource);
LL.AutoProjectType = LlProject.Card;

// Call Designer
LL.Design();

// Print
LL.Print();
LL.Dispose();
```

VB.NET:

```
Public Class DataSource
       Dim _text1 As String
       Dim _number As Double

       Public Property Text1() As String
              Get
                     Return _text1
              End Get
              Set(ByVal value As String)
                     _text1 = value
              End Set
       End Property
       Public Property Number1() As Double
              Get
                     Return _number
              End Get
              Set(ByVal value As Double)
                     _number = value
              End Set
       End Property
End Class

' Prepare data source
Dim dataSource As Object = New DataSource()
dataSource.Text1 = "Test"
dataSource.Number1 = 1.234
Dim LL As New ListLabel()
LL.DataSource = New ObjectDataProvider(dataSource)
LL.AutoProjectType = LlProject.Card

' Call Designer
LL.Design()

' Print
LL.Print()
LL.Dispose()
```

2.5.5 Sub Reports

Structuring of reports by using the report container is a pure Designer feature. Therefore there is no difference from the "normal" list as shown in section "Simple List"

For using sub tables, it is required that parent and child data are relationally linked. So the first step is to design a table element for the parent table in the report container. The next step will be to add a sub element with the child data by the toolbar in the "Objects" window.

At print time the corresponding child sub report will be automatically added for each record of the parent table. For example, the List & Label Sample Application (in the

start menu's root level) demonstrates this under Design > Extended Samples > Sub reports and relations.

2.5.6 Charts

The chart function is also automatically supported by the report container. See section "Simple List".

The List & Label Sample Application (in the start menu's root level) contains a variety of different chart samples under Design > Extended Samples.

2.5.7 Cross Tables

Not surprisingly, cross tables will be implemented the same way as described in section "Simple List".

The List & Label Sample Application (in the start menu's root level) contains a variety of different cross table samples under Design > Extended Samples.

2.5.8 Database Independent Contents

Data is not always available in a database or a DataSet. It can be required to output further data in addition to the data from the data source, such as a user name within the application, the project name or similar information. In some cases, no suitable data provider seems to be available at first. These cases will be examined in the following paragraphs.

Pass Additional Contents

If only a few variables or fields are to be added to the data of the data source, there are two possibilities:

- If the data is constant during the runtime of the report, it can just be added prior to the design or print call by using LL.Variables.Add.
- If the data changes from page to page or even from line to line, the information can be passed within the AutoDefineNewPage or AutoDefineNewLine events by using LL.Fields.Add or LL.Variables.Add.

The following example shows both approaches:

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();

// Define additional data fields
LL.Variables.Add("AdditionalData.UserName", GetCurrentUserName());
LL.Variables.Add("AdditionalData.ProjectName ", GetCurrentProjectName());
…

// Add event handling for own fields
LL.AutoDefineNewLine += new AutoDefineNewLineHandler(LL_AutoDefineNewLine);
```

```
// Call Designer
LL.Design();

// Print
LL.Print();
LL.Dispose();

…

void LL_AutoDefineNewLine(object sender, AutoDefineNewLineEventArgs e)
{
    // Switch to next record if necessary
    // GetCurrentFieldValue is function of your application
    // which returns the content of a data field.
    LL.Fields.Add("AdditionalData.AdditionalField", GetCurrentFieldValue());
}
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()

' Define additional data fields
LL.Variables.Add("AdditionalData.UserName ", GetCurrentUserName())
LL.Variables.Add("AdditionalData.ProjectName ", GetCurrentProjectName())
…

' Call Designer
LL.Design()

' Print
LL.Print()
LL.Dispose()

…

Sub LL_AutoDefineNewLine(sender As Object, e As AutoDefineNewLineEventArgs)_
Handles LL.AutoDefineNewLine
    ' Switch to next record if necessary
    ' GetCurrentFieldValue is function of your application
    ' which returns the content of a data field.
    LL.Fields.Add("AdditionalData.AdditionalField ", GetCurrentFieldValue())
End Sub
```

Suppress Data From a Data Source

Particular fields or variables that are not needed (e.g. ID fields which aren't required for printing) can be suppressed by using the AutoDefineField and AutoDefineVariable events.

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();

// Add event handling for suppressing fields
LL.AutoDefineField += new AutoDefineElementHandler(LL_AutoDefineField);

// Call Designer
LL.Design();

// Print
LL.Print();
LL.Dispose();

…

void LL_AutoDefineField(object sender, AutoDefineElementEventArgs e)
{
    if (e.Name.EndsWith("ID"))
        e.Suppress = true;
}
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()

' Call Designer
LL.Design()

' Print
LL.Print()
LL.Dispose()

…

Sub LL_AutoDefineField(sender As Object, e As AutoDefineElementEventArgs)_
Handles LL.AutoDefineNewField
       If e.Name.EndsWith("ID") Then
               e.Suppress = True
       End If
End Sub
```

Custom Data Structures / Contents

For data content apparently not directly supported, a suitable provider is found in most cases anyway. Business data from applications can generally be passed through the object data provider, if the data is present in comma-separated form, the data provider

from the "Dataprovider" sample can be used. Many other data sources support the serialization to XML, so that the XmlDataProvider can be used.

Your own class that implements the IDataProvider interface can be used, too, of course. A good starting point is the DataProvider sample which demonstrates a simple CSV data provider. Often one of the existing classes can be used as base class. If for example a data source is to be connected that implements the IDbConnection interface, it can be inherited from DbConnectionDataProvider. Only the Init method has to be overwritten, where the available tables and relations have to be provided. The component help for .NET provides an example of how this is done for SQL Server data with the SqlConnectionDataProvider. Most database systems provide similar mechanisms.

At github.com/combit/NETDataProviders you'll find Open Source implementations for a range of other data sources like MySQL, PostgreSQL, DB2 and Oracle. These may be a good starting point for your own implemetation as well.

2.5.9 Export

The export formats can be completely controlled "remotely" so that no interaction by the user is required anymore. Additionally, the selection of formats can be restricted as required or desired for the specific report.

Export Without User Interaction

This can be easily done by using the ExportConfiguration class of the ListLabel component.

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();

// Set target and path (here: PDF) and project file
ExportConfiguration expConfig = new ExportConfiguration(LlExportTarget.Pdf,
"<Target filename with path>", "<Project filename with path>");

// Show result
expConfig.ShowResult = true;

// Start export
LL.Export(expConfig);
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()

' Set target and path (here: PDF) and project file
```

```
Dim expConfig As New ExportConfiguration(LlExportTarget.Pdf, "<Target
filename with path>", "<Project filename with path>")

' Show result
expConfig.ShowResult = True

' Start export
LL.Export(expConfig)
LL.Dispose()
```

Many other options can be set using the ExportOptions enumeration of the ListLabel component.

Restriction of Export Formats

If only specific export formats should be available for the end user, the list of formats can be restricted exactly to these formats. This is possible with the option LlOptionString.Exports_Allowed. A list of all available formats can be found in section "Export".

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();

// Only allow PDF and preview
LL.Core.LlSetOptionString(LlOptionString.Exports_Allowed, "PDF;PRV");

// Print
LL.Print();
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()

' Only allow PDF and preview
LL.Core.LlSetOptionString(LlOptionString.Exports_Allowed, "PDF;PRV")

' Print
LL.Print()
LL.Dispose()
```

2.5.10 Extend Designer by Custom Function

The following example shows how a function can be added that allows querying a registry key within a report. The result of the function could be used in appearance conditions for objects for example. Of course the properties of the DesignerFunction

class can also be set directly in the properties window of the development environment.

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();

// Initialize function
DesignerFunction RegQuery = new DesignerFunction();
RegQuery.FunctionName = "RegQuery";
RegQuery.GroupName = "Registry";
RegQuery.MinimalParameters = 1;
RegQuery.MaximumParameters = 1;
RegQuery.ResultType = LlParamType.String;
RegQuery.EvaluateFunction += new
EvaluateFunctionHandler(RegQuery_EvaluateFunction);

// Add function
LL.DesignerFunctions.Add(RegQuery);

LL.Design();
LL.Dispose();

…

void RegQuery_EvaluateFunction(object sender, EvaluateFunctionEventArgs e)
{
    // Get registry key
    RegistryKey key = Registry.CurrentUser.OpenSubKey(@"Software\combit\");
    e.ResultValue = key.GetValue(e.Parameter1.ToString()).ToString();
}
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()

' Initialize function
Dim RegQuery As New DesignerFunction()
RegQuery.FunctionName = "RegQuery"
RegQuery.GroupName = "Registry"
RegQuery.MinimalParameters = 1
RegQuery.MaximumParameters = 1
RegQuery.ResultType = LlParamType.String

' Add function
LL.DesignerFunctions.Add(RegQuery)

LL.Design()
LL.Dispose()
```

```
…

Sub RegQuery_EvaluateFunction(sender As Object,
      e As EvaluateFunctionEventArgs) Handles RegQuery.EvaluateFunction
    ' Get registry key
    Dim key As RegistryKey
    RegistryKey = Registry.CurrentUser.OpenSubKey("Software\combit\")
    e.ResultValue = key.GetValue(e.Parameter1.ToString()).ToString()
End Sub
```

2.5.11 Join and Convert Preview Files

The preview format can be used as the output format if, for example, multiple reports should be joined to one or if archiving the output in the form of a PDF is desired in addition to the direct output. The following example shows some options of the PreviewFile class.

C#:

```
// Open preview files, first file/coversheet with write access
PreviewFile cover = new PreviewFile(@"<Path>\frontpage.ll", false);
PreviewFile report = new PreviewFile(@"<Path>\report.ll", true);

// Append report to first file/coversheet
cover.Append(report);

// Print complete report
cover.Print();

// Convert report to PDF
cover.ConvertTo(@"<Path>\report.pdf");

// Release preview files
report.Dispose();
cover.Dispose();
```

VB.NET:

```
' Open preview files, first file/coversheet with write access
Dim cover As New PreviewFile("<Path>\frontpage.ll", False)
Dim report As New PreviewFile("<Path>\report.ll", True)

' Append report to first file/coversheet
cover.Append(report)

' Print complete report
cover.Print()
```

```
' Convert report to PDF
cover.ConvertTo("<Path>\report.pdf")

' Release preview files
report.Dispose()
cover.Dispose()
```

2.5.12 Sending E-Mail

Sending e-Mail can also be controlled via the list of export options (see section "Export Without User Interaction") if export and sending should be done in one step. An example showing that is the "Export" sample for C# and VB.NET.

However, independent of the previous export, it is also possible to send any files via e-Mail by using the MailJob class. This is especially interesting when generating a PDF file from the preview file as a source (see section "Join and Convert Preview Files") and the PDF file is supposed to be sent via e-mail.

C#:

```
// Instantiate mail job
MailJob mailJob = new MailJob();

// Set options
mailJob.AttachmentList.Add(@"<Path>\report.pdf");
mailJob.To = "info@combit.net";
mailJob.Subject = "Here is the report";
mailJob.Body = "Please note the attachment.";
mailJob.Provider = "XMAPI";
mailJob.ShowDialog = true;

// Send e-Mail
mailJob.Send();
mailJob.Dispose();
```

VB.NET:

```
' Instantiate mail job
Dim mailJob As New MailJob()

' Set options
mailJob.AttachmentList.Add("<Path>\report.pdf")
mailJob.To = "info@combit.net"
mailJob.Subject = " Here is the report "
mailJob.Body = " Please note the attachment."
mailJob.Provider = "XMAPI"
mailJob.ShowDialog = True

' Send e-Mail
mailJob.Send()
```

```
mailJob.Dispose()
```

2.5.13 Store Project Files in a Database

Project files can also be stored in a database. Besides the option to unpack these directly from the database and to store them in the local file system, this job can be passed to List & Label as well. The Print and Design methods have both overloads that allow passing a stream directly.

When using these overloads, a few important changes in how these methods work are to be obeyed. The background for these changes is the missing local file context and therefore the missing possibility to create new files:

- It is not possible to create a new project in the Designer
- The menu items File > Save as and File > Open are not available
- Project includes are deactivated
- Drilldown is not available
- The Designer function "ProjectPath$" is not available

In the case of designing it can happen of course that the passed stream is being modified. In this case you have to write the updated stream into the database after designing.

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();
byte[] report = GetReportFromDatabase();
MemoryStream memStream = new MemoryStream(report);

LL.Print(LlProject.List, memStream);
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()
Dim report As Byte() = GetReportFromDatabase()
Dim memStream As New MemoryStream(report)

LL.Print(LlProject.List, memStream)
LL.Dispose()
```

2.5.14 Network Printing

When printing in the network, keep the following two points in mind:

- Preview files are usually created in the same directory as the project file by the name of the project file and the extension "LL". If two users want to print the same file to preview, the second user receives an error message. This can be avoided by setting *LlPreviewSetTempPath()* (see example below).
- The same applies for printer settings files. These also will – with the currently selected extension – be searched for or created in the directory of the project file. *LlSetPrinterDefaultsDir()* should be used here. This setting is of particular importance if the available printers vary from workstation to workstation. That's one of the reasons why you shouldn't redistribute printer settings files to your users.

C#:

```
ListLabel LL = new ListLabel();
LL.DataSource = CreateDataSet();

// Set local temporary path
LL.Core.LlPreviewSetTempPath(Path.GetTempPath());

// Printer settings should be created in user-specific sub directory
// so changes will be stored permanently
LL.Core.LlSetPrinterDefaultsDir(<Path>);

LL.Print();
LL.Dispose();
```

VB.NET:

```
Dim LL As New ListLabel()
LL.DataSource = CreateDataSet()

' Set local temporary path
LL.Core.LlPreviewSetTempPath(Path.GetTempPath())

' Printer settings should be created in user-specific sub directory
' so changes will be stored permanently
LL.Core.LlSetPrinterDefaultsDir(<Path>)

LL.Print()
LL.Dispose()
```

Using the stream overloads of the Print and Design methods is an alternative here. These, for example, "automatically" take care of storing the printer settings in the passed stream. Hints as well as an example can be found in section "Store Project Files in a Database".

3. Programming With the OCX Component

Parallel with the VCL component and .NET assembly, List & Label includes the OCX component(s) for integration into your IDE. The following section exclusively discusses working with this component. If you do not work with the OCX component, you can skip this section.

3.1 Integration of the Component

The component is integrated by using the cmll26o.ocx file containing the control. This file is located in the "Redistributable Files" directory. For more information on how to install the OCX in your IDE, refer to the online help documentation for your development environment.

After integrating the control, an icon should appear automatically in the component area of the toolbar or toolbox of your IDE.

Now you can begin customizing List & Label to suit your needs by configuring the available properties and implementing the required programming logic. There are two different ways to do this:

- The simple print and design methods
- A separate, iterative print loop

The first option is described below. The iterative approach is for the most part the same as the direct use of the DLL and is thus covered by the general description of the List & Label API.

3.2 Simple Print and Design Methods

3.2.1 Working Principle

The print and design methods of the OCX Control implement a standardized print loop, which can be used directly with most of the simple applications (applications working with only one table). To print multiple tables, a separate print loop is used (see chapter "Printing Relational Data"). In the case of this method, the data is passed within the events *CmndDefineVariables* and *CmndDefineFields* to List & Label. This allows any data source to be connected individually. The event arguments allow access to useful information such as user data that has been copied, the *Design* mode etc. The property *pbLastRecord* is used to notify the print loop that the last data record has been reached. As long as this is not the case, each event is invoked repeatedly in order to retrieve the data.

The Design method shows the Designer in a modal pop-up window on top of your application window.

You can specify additional options in the *CmndSetPrintOptions* event. Internally, this event is triggered after invoking *LlPrintWithBoxStart()* but before the actual printing.

One very simple implementation of the print method looks like this:

```
Private Sub ListLabel1_CmndDefineVariables(ByVal nUserData As Long, ByVal
bDummy As Long, pnProgressInPerc As Long, pbLastRecord As Long)

 Dim i As Integer

 For i = 0 To Recordset.Fields.Count - 1

   Select Recordset.Fields(i).Type
    Case 3, 4, 6, 7: para = LL_NUMERIC: content$ = Recordset.Fields(i)
    Case 8: para = LL_DATE_MS: a! = CDate(Recordset.Fields(i)): content$ =
a!:
    Case 1: para = LL_BOOLEAN: content$ = Recordset.Fields(i)
    Case Else: para = LL_TEXT: content$ = Recordset.Fields(i)
   End Select

   nRet = LL.LlDefineVariableExt(Recordset.Fields(i).Name, content$, para)
 Next i

 If bDummy = 0 Then
   pnProgressInPerc = Form1.Data1.Recordset.PercentPosition
   Recordset.MoveNext
 End If
End Sub
```

3.2.2 Using the UserData Parameter

The *Print* and *Design* methods allow a *UserData* parameter of the type "integer" to be passed. This parameter makes it possible to prepare various data in the event for List & Label. For example, using the parameter, it would be possible to provide data for invoice printing as well as for a list of customers.

3.3 Transferring Unbound Variables and Fields

Transferring variables and fields follows the regular List & Label principle. There are three "API variations" available for the registration.

API	Description
LlDefineVariable	Defines a variable of the type LL_TEXT and its content.
LlDefineVariableExt	As described above; additionally can be transferred along with the List & Label data type.
LlDefineVariableExtHandle	As described above, except that the content must now be a handle.

One example in which a "text" type variable is registered looks like this:

```
LL.LlDefineVariableExt("MyVariable", "Content", LL_TEXT)
```

You can find the constants for the List & Label data types in the cmll26.bas (VB) unit located in your List & Label installation directory.

3.3.1 Pictures

To transfer picture files saved on your system, use

```
LL.LlDefineVariableExt ("Picture", <file path>, LL_DRAWING)
```

Graphics are transferred using the "API variation" *LlDefineVariableExtHandle()*. For more information about this function, refer to chapter "API Reference".

3.3.2 Barcodes

Barcodes are transferred by using the constant *LL_BARCODE...*. One example in which an EAN13-type barcode variable is registered looks like this:

```
LL.LlDefineVariableExt('EAN13', '123456789012',LL_BARCODE_EAN13);
```

3.4 Language Selection

The List & Label Designer is available in numerous languages. Thus, the component provides full-scale support in the implementation of multilingual desktop applications. There are two ways for telling List & Label which language to use.

- Assigning the "Language" property the appropriate language:

```
LL.Language = CMBTLANG_ENGLISH
```

- Set the language directly in the component.

Note: You need the appropriate language kit in order to display the respective language. For a list of the kits we currently offer and their prices, please visit our online shop at www.combit.com.

3.5 Working With Events

List & Label offers a variety of callbacks, which have been implemented as events in the List & Label OCX component.

For a detailed description of the available events, please refer to the online help included with the OCX component.

3.6 Displaying a Preview File

A separate component exists for displaying previews. To use this component, you must include the file cmll26v.ocx in your IDE. The component offers special ways to use the List & Label preview format. For example, you can start a PDF export from the control. You can also customize the control to meet your specific needs by displaying/hiding the toolbar buttons with the properties of the control. It is also possible to respond to the click events of the buttons and store any separate handling routines that may be necessary.

3.7 Working With Preview Files

The List & Label Storage API allows access to the LL preview files. You can query general information or the specific pages, merge several files and save user data. To use the Storage API, you must include the Unit cmls26.bas in your project or invoke the functions from the OCX Control.

3.7.1 Opening a Preview File

You can open the preview file with the *LlStgsysStorageOpen()* function. General information about the file can now be accessed using a whole series of other functions.

Visual Basic:

```
Dim hStgOrg As Long

hStgOrg = LL.LlStgsysStorageOpen("C:\Test.ll", "", False, True)
```

3.7.2 Merging Multiple Preview Files

You can merge multiple preview files. To do this, you must first open the target file. Since write access is required, you must pass a "false" value for the second parameter, ReadOnly. Using the *LlStgSysAppend()* function you can then merge the files.

Visual Basic:

```
Dim hStgOrg As Long
Dim hStgAppend As Long

hStgOrg = LL.LlStgsysStorageOpen("C:\Test1.ll", "", False, True)
hStgAppend = LL.LlStgsysStorageOpen("C:\Test2.ll", "", False, True)
```

```
LL.LlStgsysAppend hStgOrg, hStgAppend

LL.LlStgsysStorageClose hStgOrg
LL.LlStgsysStorageClose hStgAppend
```

3.7.3 Debugging

You can enable debugging for the OCX component by setting the *DebugMode* property in the source code to "1", e.g.:

```
LL.LlSetDebug = 1
```

For more information about the Debwin debugging tool, please refer to the chapter "Debug Tool Debwin".

3.8 Extending the Designer

List & Label offers a variety of options for extending the Designer. These include, among other things, the various events of the component such as a menu operation and the various features that are available. Yet there are numerous other possibilities...

3.8.1 Using the Formula Wizard to Add Your Own Functions

The formula wizard and its features are among the most important and powerful capabilities of the Designer. Using a special List & Label component for the OCX, you can also integrate fully customized functions into the Designer. To use the control, you must include the file cmll26fx.ocx in your IDE.

To add a new function, insert this component on a form at the time of design. You can now set the necessary parameters in the Properties window of this component.

Property	Description
Name	The unique name of the designer function.
Description	An additional description of the function for the formula wizard.
GroupName	The group in which the function is displayed in the formula wizard.
Visible	Indicates whether or not the function will be displayed in the wizard.
MinimumParameters	The minimum number of parameters. Permissible values between 0 and 4.
MaximumParameters	The maximum number of parameters. Here too, permissible values between 0 and 4. The value must be equal to or greater than the minimum number. Increasing the number results in optional parameters.

ResultType	The data type of the return value.

Using the properties, you can customize the configuration of the new Designer function. The parameters for the design functions are defined in the source code. This may look as follows:

Visual Basic:

```
Private Sub InitializeDesFunction()
       Dim param1 As DesignerFunctionsParameter
       Dim param2 As DesignerFunctionsParameter

       Set param1 = DesFunc_Add.Parameter1
       param1.Description = "First Value"
       param1.Type = LlParamType.ParamType_Double

       Set param2 = DesFunc_Add.Parameter2
       param2.Description = "Second Value"
       param2.Type = LlParamType.ParamType_Double

       DesFunc_Add.ParentComponent = ListLabel1
End Sub
```

In order to bring the function to life, you have to handle the event DesFunc_Add_EvaluateFunction. Using the event arguments, you gain access to the parameters entered by the user. For example, to return the sum of the two parameters, use the following lines:

Visual Basic:

```
Private Sub DesFunc_Add_EvaluateFunction(ResultValue As Variant,
               ResultType As CMLL26FXLibCtl.LlParamType,
               DecimalPositions As Long,
               ByVal Parameters As Long, ByVal Parameter1 As Variant,
               ByVal Parameter2 As Variant, ByVal Parameter3 As Variant,
               ByVal Parameter4 As Variant)

       ResultValue = CDbl(Parameter1) + CDbl(Parameter2)
       ResultType = ParamType_Double

End Sub
```

Two additional events also give you the option to further modify the function. DesFunc_Add_CheckFunctionSyntax lets you perform a syntax check. Here you can check the data types of the parameters and, for example, make sure the parameters fall within a certain range. DesFunc_Add_ParameterAutoComplete allows you to define several suggested values for the AutoComplete feature of the formula wizard.

3.8.2 Adding Your Own Objects to the Designer

Similar to the way you add functions to the Designer, you can also define your own object types and register them in List & Label. The user can access the new objects in the usual way on the left toolbar and menu.

Here there is also a special component for adding objects. To use this, you must include the file cmll26ox.ocx in your IDE.

After adding this component to your form, you can define the properties of the new object in the Properties window of the component.

The table provides an overview:

Property	Description
Name	The unique name of the object.
Description	This description appears in the Designer. It is allowed to contain spaces but should not be more than 30 characters long.
Icon	The icon of the object that will be shown in the toolbar and menu in Designer. It should be a 16x16 pixel, 16-color icon.

The component has three types of events. First, the DesObj_Picture_CreateDesignerObject event is triggered when the user creates a new object. If desired, you can have an initial dialog appear on the user's screen. For example, this can be a wizard that helps the user to more easily configure the new object. If it is not necessary to use this in a particular case, simply skip handling the event.

The DesObj_Picture_EditDesignerObject event is triggered when the user double-clicks the newly inserted object or selects "Properties" from the context menu.

After you are finished editing the object, you are prompted by List & Label to show the object. The DesObj_Picture_DrawDesignerObject event is triggered for this purpose. Using the usual methods, you can now draw in the workspace. Here, of course, it is also possible or useful to access the stored object properties.

3.9 The Viewer OCX Control

3.9.1 Overview

The CMLL26V.OCX control can be used to view, export or print List & Label preview files in your own environment.

It can, for example, be inserted into your own application or into an Internet page.

When printing, the projects will be fitted into the page, automatically taking account of the "physical page" flag and the page orientation to create the best possible printout result.

If a URL is given instead of a file name, the control will try to load the file into a temporary cache on the local hard disk if the URLMON.DLL is registered on the system, see below.

Please note, that a browser is required that supports ActiveX Controls, for example MS Internet Explorer.

3.9.2 Registration

The control can be registered in the usual way, for example with the command "REGSVR32 CMLL26V.OCX", by the programming environment or by your setup program. It cannot be used without registration. For further information, please refer to the file Redist.txt in your List & Label installation directory.

3.9.3 Properties

AsyncDownload [in, out] BOOL: This is an option to improve the screen update. If the download is not asynchronous, then the screen around the OCX will not be updated until the download is finished. On the other hand, you must be careful with the async download, as you might not be able to set properties like "Page" until the download is finished (see event LoadFinished). After setting this option, read the value again to check whether this feature is supported. This has no effect on local files. Default: FALSE

Enabled [in, out] BOOL: Defines whether the control is enabled or disabled. This will have an effect on the user interface. Default: TRUE

BackColor [in, out] OLE_COLOR: Defines the background color, i.e. the color that is painted on:

- the whole background if no preview file can be displayed, and
- the background outside the paper. Default: RGB(192, 192, 192) [light gray]

FileURL [in, out] BSTR: The most important property. Defines the name of the preview file. This can be a file name as well as a URL. Default: <empty>

Pages [out] Long: The total number of pages in the preview

CurrentPage [in, out] Long: Sets or reads the current page index (1..pages). Default: 1

ToolbarEnabled [in, out] BOOL: Defines whether the toolbar should be displayed. The toolbar is not necessary, as all of its functions can be called by methods (as an example, see LLVIEW26.EXE and its menu items). So there is no problem at all in defining your own toolbar. Default: TRUE

ToolbarButtons [out] LPDISPATCH: Returns a ToolbarButtons object that can be used to get or set the status of each toolbar button. The object has the following methods:

- ***GetButtonState([in] nButtonID) LONG*** Returns the button state for the passed TLB: constant.

Value	Meaning	Constant
-1	Hidden	TLBS_PRV_HIDE
0	Default	TLBS_PRV_DEFAULT
1	Enabled	TLBS_PRV_ENABLED
2	Disabled	TLBS_PRV_DISABLED

Example:

```
Dim oTlb as ToolbarButtons
Set oTlb = LlViewCtrl1.ToolbarButtons
MsgBox oTlb.GetButtonState(TLB_PRV_FILEEXIT)
```

- ***SetButtonState([in] nButtonID, [in] nButtonState)*** Sets the button state for the passed TLB_ constant. See above for valid state values.

Example:

```
Dim oTlb as ToolbarButtons
Set oTlb = LlViewCtrl1.ToolbarButtons
oTlb.SetButtonState TLB_PRV_FILEEXIT, TLBS_PRV_HIDE
```

The ID constants can also be found in the file MENUID.TXT of your List & Label installation.

ShowThumbnails[in, out] BOOL: Defines whether the thumbnail bar is visible in the preview control. Default: TRUE

SaveAsFilePath [in, out] BSTR: Defines the default path for the "Save as..." dialog. When using the SaveAs method the user-defined file name will be returned.

CanClose[out] BOOL: Must be used to query the state of the control before closing the hosting form/page. If the result is FALSE, the control must not be destroyed.

Version [out] LONG: Returns the version number of the OCX control (MAKELONG(lo,hi)).

3.9.4 Methods

GotoFirst: Shows the first page

GotoPrev: Shows the previous page (if possible)

GotoNext: Shows the next page (if possible)

GotoLast: Shows the last page

ZoomTimes2: Zooms in with a factor of 2

ZoomRevert: Resets to the previous zoom state (the zoom states are on a stack, where ZoomRevert pops the last one off before resizing to that zoom state).

ZoomReset: Resets the zoom state to "fit to client window".

SetZoom ([in] long nPercentage) : Sets the zoom to the passed factor (1-30). Use the factor -100 for page width.

PrintPage ([in] Long nPage, [in] Long hDC): Prints the page (nPage = 1..page). If hDC is NULL, a printer dialog will be shown.

PrintCurrentPage ([in] Long hDC): Prints the current page. If hDC is NULL, a printer dialog will be shown.

PrintAllPages ([in] BOOL bShowPrintOptions): Prints all pages of the project. If bShowPrintOptions is TRUE, a printer dialog will be shown.

RefreshToolbar: Refreshes the toolbar

SendTo: Starts "Send"

SaveAs: Starts "Save As"

GetOptionString([in] BSTR sOption) BSTR: Returns mail settings. See the chapter on project parameters for more information. The available options are documented in chapter "Send Export Results via E-Mail".

SetOptionString([in] BSTR sOption, [in] BSTR sValue) BSTR: Sets mail settings. See the chapter on project parameters for more information. The available options are documented in chapter "Send Export Results via E-Mail". Additionally, the control supports the following options:

Print.NoProgressDlg: Can be used to suppress the progress dialog during printing.

Value	Meaning
0	The progress dialog is displayed during printing.
1	The progress dialog is suppressed.
Default	0

3.9.5 Events

BtnPress

Syntax:

```
BtnPress(short nID): BOOL
```

Task:

Tells you that a toolbar button has been pressed.

Parameter:

nID: See MENUID.TXT for valid IDs

Return value:

TRUE if the action should not be performed by the OCX.

PageChanged

Syntax:

```
PageChanged
```

Task:

Tells you that the current page has been changed.

Parameter:

-

Return value:

-

LoadFinished

Syntax:

```
LoadFinished(BOOL bSuccessful)
```

Task:

Notifies you that the file is on the local disk. Important in case of asynchronous download.

This does not guarantee that the downloaded file is correct - you should use the property 'pages' to check: a value of 0 indicates a failure.

Parameter:

bSuccess: *TRUE* if the file has been transferred; *FALSE* if the download failed

Return value:

-

3.9.6 Visual C++ Hint

Visual C++ (at least V 5.0) notifies you with an "Assertion Failed" message in occcont.cpp that something is incorrect. This assertion is only in the debug build and results from the ATL library we used for this control.

3.9.7 CAB Files Packaging

A CAB file is supplied with List & Label.

3.9.8 Inserting the OCX Into Your Internet Page

As stated above, the control can be inserted into an Internet page. Properties can be set via <PARAM> tag or scripts.

4. Programming With the VCL Component

In parallel with the OCX component and .NET assembly, List & Label includes the VCL component(s) for integration into the IDE from Embarcadero. The following section exclusively discusses working with this component. If you do not work with the VCL component, you can skip this section.

4.1 Integration of the Component

The component is integrated using a package. If you are using a version of Delphi earlier than version 6, use the package ListLabel26PreDelphi6.dpk. For Delphi versions 6 and higher, use the package ListLabel26.dpk. You will find both files in your List & Label installation directory under "Samples\Delphi". For more information on installing the packages in your IDE, refer to the online help documentation for your development environment.

After installing the package, several icons are automatically created in the component area of the toolbar.

Now you can begin customizing List & Label to suit your needs by configuring the available properties and implementing the required programming logic. There are three different ways to do this:

- Data binding
- The simple print and design methods
- A separate, iterative print loop

The first two options are described below. The iterative approach is for the most part the same as the direct use of the DLL and is thus covered by the general description of the List & Label API.

4.2 Data Binding

An extra control exists for data binding using the List & Label VCL component. This control inherits all of the properties from the "normal" component and adds options for direct data binding. Using the DataSource property, you can now specify a data source of the type TDataSource.

4.2.1 Binding List & Label to a Data Source

The data binding is performed with the DataSource property. You can either program in the assignment of these or use the Properties window in your IDE. If you have already created a DataSource in your form, you can select it from the Properties window. The necessary link is created automatically.

You can now implement the program code for starting the design and print. To do this, you can, for example, include the Print or Design method call in the Click event of a

new button, without any additional parameters. The data from the assigned source can be made available automatically.

```
// Show Designer
DBL26_1.AutoDesign('Invoice');

// Execute print
DBL26_1.AutoPrint('Invoice','');
```

If you want to modify the standard flow of data-bound printing, there are several properties you can use. These begin with "Auto..." and are found in the data section of the Properties window.

Property	Description
AutoProjectFile	File name of the print project being used
AutoDestination	Print format, for example printer, preview, PDF, HTML etc.
AutoProjectType	Type of print project (list, index cards, label)
AutoFileAlsoNew	Creating a new project when opening Designer enabled
AutoShowPrintOptions	Show print options when starting print
AutoShowSelectFile	Show the file selection dialog for printing and Designer
AutoBoxType	Type of progress box

4.2.2 Working With Master Detail Records

In conjunction with data binding and list projects, List & Label can automatically analyze and transfer relationships existing between multiple tables.

The type of data transfer is defined using the *AutoMasterMode* property. The underlying enumeration provides for the following values:

- None: No master-detail relations are analyzed.
- AsFields: Master and detail data are registered in parallel as fields. This makes it possible to realize groups, statistics and overviews.
- AsVariables: The master data is registered in the form of variables and the detail data in the form of fields, respectively. After each master data set, the project is reset internally with *LlPrintResetProjectState()*. This makes it possible to use a single print job for printing in a row several identical reports containing different data, for example, multiple invoices.

Please also note the examples included for data binding.

4.2.3 Additional Options for Data Binding

There are different events you can use for influencing the process for data binding of the component. The table provides an overview:

Event	Description
AutoDefineNewPage	The event is invoked for each new page and allows additional variables to be registered for this page. The property IsDesignMode of the event arguments indicates whether design mode is being used.
AutoDefineNewLine	This event is invoked for each new line, before the data-bound fields are registered automatically. In the same was as you do with AutoDefineNewPage, you can register additional fields here.
AutoDefineVariable	This event is invoked for each variable that is automatically created using data binding. Using the Name and Value properties of the event arguments, you can manipulate the names and content of each individual variable before passing it for printing.
AutoDefineField	Analogous to AutoDefineVariable for fields
AutoDefineTable	This event is invoked for each table that has been registered via *LlDbAddTable()*. You can suppress passing.
AutoDefineTableSortOrder	This event is invoked for each sort order that has been registered via *LlDbAddTableSortOrder()*. You can suppress passing.
AutoDefineTableRelation	This event is invoked for each DataRelation that has been registered via *LlDbAddTableRelation()*. You can suppress passing.

Please note that when using these events, you must cast the sender object to the respective component types if you want to work with the triggering component instance. Otherwise, you may encounter problems with DrillDown or the preview in the Designer.

Example:

```
procedure TForm1.DBL26_1AutoDefineNewPage(Sender: TObject;
  IsDesignMode: Boolean);
begin
  (sender as TDBL26_).LlDefineVariable('MyCustomVariableName',
'MyCustomVariableValue');
end;
```

4.3 Simple Print and Design Methods

4.3.1 Working Principle

The methods implement a standardized print loop that can be used directly for most of the simpler applications if you do not pass the data via DataBinding. In the case of this method, the data is passed within the events *DefineVariables* and *DefineFields* to List & Label. This allows any data source to be connected individually. The event arguments allow access to useful information such as user data that has been copied, the Design mode etc. The property *IsLastRecord* is used to notify the print loop that the last data record has been reached. As long as this is not the case, each event is invoked repeatedly in order to retrieve the data.

The *Design* method shows the Designer in a modal pop-up window on top of your application window.

You can specify additional options in the *LLSetPrintOptions* event. Internally, this event is triggered after invoking *LlPrintWithBoxStart()* but before the actual printing.

One very simple implementation of the *Print* method looks like this:

Delphi:

```
procedure TForm1.LLDefineVariables(Sender: TObject; UserData: Integer;
  IsDesignMode: Boolean; var Percentage: Integer;
  var IsLastRecord: Boolean; var EventResult: Integer);
var
  i: integer;
begin
     For i:= 0 to (DataSource.FieldCount-1) do
     begin
         LL.LlDefineVariableFromTField(DataSource.Fields[i]);
     end;
     if not IsDesignMode then
     begin
          Percentage:=Round(DataSource.RecNo/DataSource.RecordCount*100);
          DataSource.Next;
          if DataSource.EOF=True then IsLastRecord:=true;
     end;
end;
```

4.3.2 Using the UserData Parameter

The *Print* and *Design* methods allow a *UserData* parameter of the type "integer" to be passed. This parameter makes it possible to prepare various data in the event for List & Label. For example, using the parameter, it would be possible to provide data for invoice printing as well as for a list of customers.

4.4 Transferring Unbound Variables and Fields

Transferring variables and fields follows the regular List & Label principle. Three "API variations" can be used for the registration.

API	Description
LlDefineVariable()	Defines a variable of the type LL_TEXT and its content.
LlDefineVariableExt()	As described above; additionally can be transferred along with the List & Label data type.
LlDefineVariableExtHandle()	As described above, except that the content must now be a handle.

One example in which a "text" type variable is registered looks like this:

```
LL.LlDefineVariableExt('MyVariable', 'Content', LL_TEXT);
```

You can find the constants for the List & Label data types in the cmbtll26.pas unit located in your List & Label installation directory.

4.4.1 Pictures

To transfer pictures files saved on your system, use

```
LL.LlDefineVariableExt ('Picture', <data path>, LL_DRAWING);
```

Graphics in the memory (only for BMP, EMF) are transferred with the API *LlDefineVariableExtHandle()*. For example, to display the graphic as a bitmap or metafile, use the following call:

```
LL.LlDefineVariableExtHandle('Picture', BufferImage.picture.bitmap.handle,
LL_DRAWING_HBITMAP);
```

or

```
LL.LlDefineVariableExtHandle('Picture', BufferImage.picture.metafile.handle,
LL_DRAWING_HEMETA);
```

4.4.2 Barcodes

Barcodes are transferred by using the constant *LL_BARCODE...*. One example in which an EAN13-type barcode variable is registered looks like this:

```
LL.LlDefineVariableExt('EAN13', '123456789012',LL_BARCODE_EAN13);
```

4.5 Language Selection

The List & Label Designer is available in numerous languages. Thus, the component provides full-scale support in the implementation of multilingual desktop applications. There are two ways for telling List & Label which language to use:

1. Assigning the *Language* property the appropriate language:

```
LL.Language = ltEnglish;
```

2. Set the language directly in the component.

Note: You need the appropriate language kit in order to display the respective language. For a list of the kits currently available, please visit our online shop at www.combit.com.

4.6 Working With Events

List & Label offers a variety of callbacks, which have been implemented as events in the List & Label VCL component.

For a detailed description of the available events, please refer the online help included with the VCL component.

4.7 Displaying a Preview File

An extra control exists for displaying a preview file using the List & Label VCL component. This provides special ways to use the List & Label preview format. For example, you can start a PDF export from the control. You can also customize the control to meet your specific needs by displaying/hiding the toolbar buttons with the properties of the control. It is also possible to respond to the click events of the buttons and store any separate handling routines that may be necessary.

4.8 Working With Preview Files

The List & Label Storage API allows access to the LL preview files. You can query general information or the specific pages, merge several files and save user data. To use the Storage API, you have to integrate the unit cmbtls26.pas into your project.

4.8.1 Opening a Preview File

You can open the preview file with the *LlStgsysStorageOpen()* function. General information about the file can now be accessed using a whole series of other functions.

Delphi:

```
var
  hStg: HLLSTG;
begin
  hStg := LlStgsysStorageOpen('c:\test.ll','',False, False)
end;
```

4.8.2 Merging Multiple Preview Files

You can merge multiple preview files. To do this, you must first open the target file. Since write access is required, you must pass a "false" value for the second parameter, *ReadOnly*. Using the *LlStgSysAppend()* function you can then merge the files.

Delphi:

```
var
  hStgOrg, hStgAppend: HLLSTG;
begin
  hStgOrg := LlStgsysStorageOpen('c:\test1.ll','',False, True);
  hStgAppend := LlStgsysStorageOpen('c:\test2.ll','',False, True);
  LlStgsysAppend(hStgOrg, hStgAppend);
  LlStgsysStorageClose(hStgOrg);
  LlStgsysStorageClose(hStgAppend);
end;
```

4.8.3 Debugging

The debugging of the VCL component allows you to either directly activate the component by setting the property *DebugMode* to "1" or alternatively in the source code. For example:

```
LL.DebugMode := LL_DEBUG_CMBTLL;
```

For more information about the Debwin debugging tool, please refer to the chapter "Debug Tool Debwin".

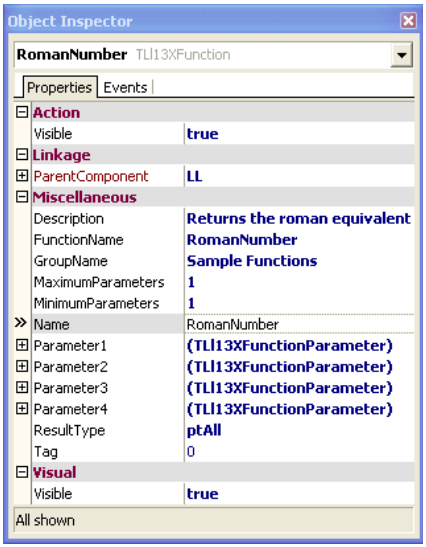
4.9 Extending the Designer

List & Label offers a variety of options for extending the Designer. These include, among other things, the various events of the component such as a menu operation and the various features that are available. Yet there are numerous other possibilities.

4.9.1 Using the Formula Wizard to Add Your Own Functions

The formula wizard and its features are among the most important and powerful capabilities of the Designer. Using a special List & Label VCL component, you can also integrate fully customized functions into the Designer.

To add a new function, insert this component on a form in the development environment. You can now set the necessary parameters in the Properties window of this component:



Property	Description
Name	The unique name of the designer function.
Description	An additional description of the function for the formula wizard.
GroupName	The group in which the function is displayed in the formula wizard.
Visible	Indicates whether or not the function will be displayed in the wizard.
MinimumParameters	The minimum number of parameters. Permissible values between 0 and 4.
MaximumParameters	The maximum number of parameters. Here too, permissible values between 0 and 4. The value must be equal to or greater than the minimum number. Increasing the number results in optional parameters.
Parameter1 – 4	The configuration for each of the four parameters can be customized.

Type	The data type of the parameter.
Description	A description of the parameter for the tooltip help in Designer.
ResultType	The data type of the return value.

Using the properties, you can customize the configuration of the new Designer function. In order to bring the function to life, you have to handle the event *OnEvaluateFunction*. Using the event arguments, you gain access to the parameters entered by the user. For example, to return the Roman numeral, use the following lines:

Delphi:

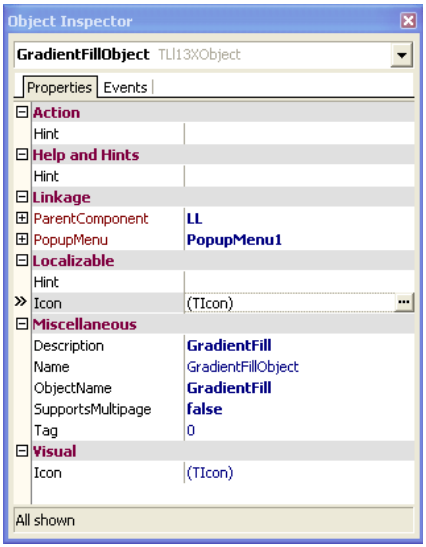
```
procedure TDesExtForm.RomanNumberEvaluateFunction(Sender: TObject;
       var ResultType: TLl26XFunctionParameterType;
       var ResultValue: OleVariant;
       var DecimalPositions: Integer; const ParameterCount: Integer;
       const Parameter1, Parameter2, Parameter3, Parameter4: OleVariant);
begin
       ResultValue:=ToRoman(Parameter1);
end;
```

Two additional events also give you the option to further modify the function. *OnCheckFunctionSyntax* lets you perform a syntax check. Here you can check the data types of the parameters and, for example, make sure the parameters fall within a certain range. Using *OnParameterAutoComplete* you can define several suggested values for the AutoComplete feature of the formula wizard.

4.9.2 Adding Your Own Objects to the Designer

Similar to the way you add functions to Designer, you can also define your own object types and register them in List & Label. The user can access the new objects in the usual way on the left toolbar and menu.

Here there is also a special component for adding objects:



After adding this component to your form, you can define the properties of the new object in the Properties window of the component. The table provides an overview:

Property	Description
Name	The unique name of the object.
Description	This description appears in the Designer. It is allowed to contain spaces but should not be more than 30 characters long.
Icon	The icon of the object that will be shown in the toolbar and menu in the Designer. It should be a 16x16 pixel icon.

The component has three types of events. First, the *OnInitialCreation* event is triggered when the user creates a new object. If desired, you can have an initial dialog appear on the user's screen. For example, this can be a wizard that helps the user to more easily configure the new object. If it is not necessary to use this in a particular case, simply skip handling the event.

The following lines initialize the object as soon as the new object is placed on the workspace for the first time.

Delphi:

```
procedure TDesExtForm.GradientFillObjectInitialCreation(Sender: TObject;
      ParentHandle: Cardinal);
```

```
begin
       with Sender as TLl26XObject do
       begin
              Properties.AddProperty('Color1', '255');
              Properties.AddProperty('Color2', '65280');
       end;
end;
```

The *OnEdit* event is triggered when the user double-clicks the newly inserted object or selects "Properties" from the context menu.

After you are finished editing the object, you are prompted by List & Label to show the object. The *OnDraw* event is triggered for this purpose. The event arguments are responsible for generating the *TCanvas* as well as a *TRect* for the object. Using the usual methods, you can now draw in the workspace. Here, of course, it is also possible or useful to access the stored object properties.

5. Programming Using the API

Besides reading this chapter, we recommend to have a look at the source code samples that are provided with List & Label for various programming languages to get started quickly. You will find them in the "Samples" folder in the List & Label start menu. Declaration files for many programming languages are additionally available in order to ease the integration of List & Label, even if there are no samples available. In this case, a look at the samples for other programming labguages might help to set up the neccessary code.

The source code snippets in this chapter are using C/C++ as programming languages, but the syntax should be easily adaptable to other languages, too.

5.1 Programming Interface

5.1.1 Dynamic Link Libraries

Basics

A DLL (Dynamic Link Library) is a collection of routines in a file which is loaded on demand by the Windows kernel.

The DLL principle allows the routines (procedures) contained within it to be "bound" (linked) to the executable at the time the application is run. Furthermore, several applications can use the DLL routines without requiring multiple copies to be installed on the system.

Of course, procedures of one DLL can also call procedures of other DLLs, as is regularly done by Windows.

List & Label uses different DLLs for specific jobs.

Usage of a DLL

Because of this method of linking, Windows has to be able to find and load the DLL file. To be accessible, the DLL has to be placed either in the path where the calling application is, the directory of Windows or its system path, or in any path contained in the system search path.

The same is valid for the DLLs List & Label uses.

These DLLs must be copied into a path in which your program is installed, or which complies with the requirements stated above.

The List & Label DLL and its dependent DLLs can be installed for Side-By-Side use in your application's directory.

For compatibility reasons concerning Windows 7 and Windows 8, no files should be placed in the Windows directory.

See chapter "Redistribution: Shipping the Application" for further information.

Linking With Import Libraries

To use the API functions, your source code must include a declaration file for cmll26.dll (C++: cmbtll26.h), which needs to be done after the "#include <windows.h>" statement resp. the precompiled header file, as Windows data types are used in the declarations. Additionally, you need to link the corresponding LIB file (C++: cmll26.lib). In order to use the API for managing preview files (see chapter "Managing Preview Files") you need to include the declaration file for cmls26.dll (C++: cmbtls26.h) and if neccessary link to the corresponding LIB file (C++: cmls26.lib).

Dynamic loading of the DLLs is also possible – you will find a description in the article "HOWTO: Loading the List & Label DLLs Dynamically in C/C++ and Delphi" in our knowledgebase.

Important Remarks on the Function Parameters of DLLs

Returning strings from the DLL to the application is always performed by returning a pointer to a memory area and, as a further parameter, an integer value for the length. The buffer size (in characters) must be passed so that buffer overflows are impossible. The string is always \0-terminated. If, for example, the buffer is too short, the returned string will be shortened and may result in incomplete return values. This is indicated by the API function's return value. *LL_ERR_BUFFERTOOSMALL* will be returned in these cases.

The parameters are checked for correctness as far as possible. While developing a program it is therefore worth switching on the debug mode (see *LlSetDebug()*) and checking the return values. You can switch off the parameter check later using *LL_OPTION_NOPARAMETERCHECK*.

Please note that with Delphi the routines require "\0"-terminated strings just like the Windows API functions. It may be necessary to use the Pascal-string to C-string conversion routines.

With Visual Basic, it may in some cases be necessary to remove the '\0'-termination. Parameters are passed with ByVal. It is recommended that you initialize buffer/strings with…

```
Dim lpszBuffer As String * 255
```

to a specific size (here 255 bytes). This is also possible using…

```
lpszBuffer$ = space$(255)
```

but requires more time and memory. It is important to reserve the memory so that the DLL does not write in unused areas - an unhandled exception would be the result otherwise.

In addition, it must be noted that some functions cannot be supported with Visual Basic; these are, however, not normally required. You can find out which functions these are in the corresponding chapters. If a parameter value is NULL (C) or nil (Pascal),

you should use "" (empty string) or 0 as value in Visual Basic, depending on the data type of the parameter. The OCX control does not need ANSIZ buffers, as it supports the native String data types.

Please note with C or C++ the C convention in source texts of having to enter, for example, the '\' in path entries twice: "c:\\temp.lbl" instead of "c:\temp.lbl".

Example:

```
INT    nSize = 16000;
LPWSTR pszBuffer = new TCHAR[nSize];
INT    nResult = <API>(hJob,...,pszBuffer,nSize);

if (nResult == LL_ERR_BUFFERTOOSMALL)
    {
    nSize = <API>(hJob,...,NULL,0);

    ASSERT(nSize > 0);
    delete[] pszBuffer;
    pszBuffer = new TCHAR[nSize];
    nResult = <API>(hJob,...,pszBuffer,nSize);
    }

...

delete[] pszBuffer;
```

5.1.2 General Notes About the Return Value

- Zero (or with some functions a positive return value) generally means that the function was successful, though there are some exceptions to this rule.
- A negative return value, apart from exceptional cases, indicates an error, which shows the reason with the corresponding error constants.

5.2 Programming Basics

This chapter should provide a quick and easy start into programming List & Label. Thus, only the programming basics will be covered. For a look at more advanced topics, please read the corresponding chapters later on.

5.2.1 Database Independent Concept

List & Label works database-independently when being programmed via the API, meaning List & Label does not access the database itself and does not have its own database drivers. This offers you an enormous number of advantages.

The advantages:

- No unnecessary overheads due to duplicate database drivers, resulting in less memory consumption so that your application can run faster.
- Less risk of problems with additional database drivers.
- More flexible, as this allows data control in one place only.
- Can be used without any database.
- Allows access to exotic databases.
- You can mix database data and "virtual" data created by your program.
- The data can be manipulated before printing.

The disadvantage:

- You really have to write some code. List & Label needs to be fed with data. This is very simple and needs little coding in standard cases.

5.2.2 The List & Label Job

To enable List & Label to distinguish between the different applications that are printing with it, a so-called job management is necessary: each application using a functionality of List & Label (print, design etc.) has to open a job first (*LlJobOpen()*,*LlJobOpenLCID()*) and pass the returned job handle as parameter to (nearly) all following calls of List & Label functions.

If you develop using one of the enclosed components, the job management is handled automatically by the control. For this reason, the job handle parameter must be omitted in calls of functions of these components.

```
HLLJOB hJob = LlJobOpen(CMBTLANG_ENGLISH);
...
LlDefineVariable(hJob, "Firstname", "George");
LlDefineVariable(hJob, "Lastname", "Smith");
...
```

5.2.3 Variables, Fields and Data Types

The delivery and definition of variables and their contents is performed with the List & Label function *LlDefineVariable(Ext)()*, the delivery and definition of fields and their contents is performed with the List & Label function *LlDefineField(Ext)()*. Regarding the names of fields and variables please refer to the section "Hints on Table, Variable and Field Names".

List & Label allows the specification of the following variable and field types. As the APIs expect string parameters to be passed, you may need to convert the actual values to strings before passing them to List & Label.

Please note the hints regarding NULL values in section "Passing NULL Values".

```
HLLJOB hJob = LlJobOpen(CMBTLANG_ENGLISH);
...
LlDefineVariable(hJob, "Firstname", "George");
LlDefineVariable(hJob, "Lastname", "Smith");
LlDefineVariableExt(hJob, "ISBN", "40|15589|97531", LL_BARCODE_EAN13, NULL);
LlDefineVariableExt (hJob, "Photo", "c:\\dwg\\test.bmp", LL_DRAWING, NULL);
...
```

Text

Constant:

LL_TEXT

Content e.g.:

"abcdefg"

Hints:

The text can contain special characters to handle word wraps. These characters are:

Value	Meaning
LL_CHAR_NEWLINE, 0x0d, 0x0a	*Text object*: Becomes a blank, if "word wrapping" is not activated in the Designer. *Table field*: Wrap is forced: "This will"+chr$(*LL_CHAR_NEWLINE*) +"be wrapped"
LL_CHAR_PHANTOMSPACE	The character is ignored, if no wrapping is chosen in the Designer. In this way, other characters can be assigned to a wrap: "This is a word-"+chr$(*LL_CHAR_PHANTOMSPACE*)+"wrap" is wrapped when needed.
LL_CHAR_LOCK	Is put in front of tabs or blanks and means that no wrapping may occur: "Not"+chr$(*LL_CHAR_LOCK*)+" here please"

The codes of these characters can be changed with *LL_OPTION_xxx-REPRESENTATIONCODE*.

Numeric

Constant:

LL_NUMERIC

Content e.g.:

"3.1415", "2.5e3"

Hint:

Exponential syntax is allowed (2.5e3 equals $2.5*10^3$). A thousands separator (as in "1,420,000.00") is not allowed.

Constant:

LL_NUMERIC_LOCALIZED

Content e.g.:

"1,255.00"

Hint:

The number is interpreted as a localized number, i.e. as a number formatted according to the local format settings. The OLE-API VarR8FromStr() is used internally.

Constant:

LL_NUMERIC_INTEGER

Content e.g.:

""5""

Hint:

Provides an integer value. The usage of integer values has a better performance in calculations. They will be displayed and printed without decimals.

Date

Constant:

LL_DATE

Content e.g.:

"2451158.5" (equals 12/11/1998 noon) or "5-1-2017" for *LL_DATE_MDY* format or "20170501" for *LL_DATE_YYYYMMDD* format

Hint:

Date values are usually expected in the Julian format. The Julian date specifies a certain date by giving the number of days that have passed since January, 1st -4713. The decimals represent the fraction of a day, which may be used to calculate hours, minutes and seconds.

Many programming languages also have a special data type for date values. The representation is usually analogous to the Julian date; however, a different start day is often used. This means that in this case an offset has to be added. To avoid this, at least for the languages Visual Basic, Visual FoxPro and Delphi, List & Label knows the following special date variants:

LL_DATE_OLE, LL_DATE_MS, LL_DATE_DELPHI_1, LL_DATE_DELPHI, LL_DATE_VFOXPRO

To make passing dates easier, there are some formats that do not require a Julian format:

LL_DATE_DMY, LL_DATE_MDY, LL_DATE_YMD, LL_DATE_YYYYMMDD.

With these, the value of a date variable can be passed directly as a string to List & Label, there is no need for your program to convert it to the Julian date - List & Label performs this task for you.

The day, month and year numbers must be separated by a dot ('.'), slash ('/') or minus sign ('-') in the first three formats.

Constant:

LL_DATE_LOCALIZED

Hint:

The date is interpreted as localized date, e.g. "12/31/2017". The OLE-API VarDateFromStr() is used internally.

Boolean

Constant:

LL_BOOLEAN

Content e.g.:

"T"

Hint:

"T","Y","J","t","y","j","1" all represent "true", all other values represent "false".

RTF Formatted Text

Constant:

LL_RTF

Content e.g.:

"{\rtf1\ansi[...]Hello {\b World}!\par}"

Hint:

The variable content must start with "{\rtf" and contain RTF-formatted text.

Important: The rendering of RTF contents in variables and fields is optimized for contents created with Microsoft's RTF control. These can be generated e.g. by using the Windows Wordpad application. Contents generated by Microsoft Word may not conform to the Windows RTF control's RTF standard and should not be used.

HTML Formatted Text

Constant:

LL_HTML

Content e.g.:

"<html><body>Hello World</body></html>"

Hint:

List & Label uses a custom component for HTML rendering. This component supports a restricted CSS subset. The correct rendering of entire web pages is not the main intent – the component rather offers a quick and easy way to render simple HTML streams.

Drawing

Constant:

LL_DRAWING

Content e.g.:

"c:\temp\sunny.jpg"

Hint:

The variable content represents the name of a graphics file (C/C++-programmers note: use a double "\\" with path specifications)

Constant:

LL_DRAWING_HMETA, LL_DRAWING_HEMETA, LL_DRAWING_HBITMAP, LL_-DRAWING_HICON

Hint:

Variable content is a handle to a graphic of the respective type in the memory (can only be defined using *LlDefineVariableExtHandle()* or *LlDefineFieldExtHandle()*)

Barcode

Constant:

LL_BARCODE

Contents:

"Barcode Text"

Hint:

The variable contents are the text to be printed in the barcode. The format and character range of the barcodes are described in the online help.

Constant:

All constants starting with LL_BARCODE_...

User Object

Constant:

LL_DRAWING_USEROBJ, LL_DRAWING_USEROBJ_DLG

Hint:

This object is drawn by the application itself in a callback/event procedure. For *LL_DRAWING_USEROBJ_DLG* the programmer can supply a custom properties dialog for the object available in the Designer.

The use of this variable type is a more advanced topic and is described later in this manual.

5.3 Invoking the Designer

5.3.1 Basic Scheme

In pseudo code, calling the Designer is done as follows (functions marked with '*' are optional calls):

```
<open Job>
       (LlJobOpen, LlJobOpenLCID)
<define List & Label-settings>*
       (LlSetOption,
        LlSetOptionString,
        LlSetDebug,
        LlSetFileExtensions,
        LlSetNotificationMessage,
        LlSetNotificationCallback)
<which file?>*
       (LlSelectFileDlgTitleEx)
<define variables>
       (LlDefineVariableStart,
        LlDefineVariable,
        LlDefineVariableExt,
        LlDefineVariableExtHandle)
<define fields>*       (only LL_PROJECT_LIST)
       (LlDefineFieldStart,
        LlDefineField,
        LlDefineFieldExt,
        LlDefineFieldExtHandle)
<disable functions>*
       (LlDesignerProhibitAction,
        LlDesignerProhibitFunction)
<call designer>
       (LlDefineLayout)
<close Job>
       (LlJobClose)
```

It is sufficient for job management to get the job at the beginning of the program and to release it at the end; this job is then used both for Designer calls and printing. Normally a job handle can be retained for the whole lifetime of the application, so that it only has to be released at the end.

We recommend making global settings valid for all List & Label calls after LlJobOpen()/LlJobOpenLCID() and making local settings such as disabling menu items directly before calling the Designer or printing.

5.3.2 Annotations

If setting of certain options is required, then this must of course be done before calling the Designer.

Normally the user is asked (by a file selection dialog) which file he would like to process. Let's assume in our example that a label is to be processed:

It is important that the buffer receiving the filename is pre-initialized - either to an empty string (""), or to a file name suggestion (which also sets the path!):

```
TCHAR aczProjectFile [_MAX_PATH];
_tcscpy(aczProjectFile,"c:\\mylabel.lbl");
LlSelectFileDlgTitle(hJob, hWindow, "Choose label", LL_PROJECT_LABEL,
       aczProjectFile, _MAX_PATH, NULL);
```

Of course this can also be done with an individual dialog box, or a file name can be passed to the Designer directly if the user should not be given the option of choosing.

List & Label must be informed of the possible variables in order to make them available to the user during design time or print time. Otherwise, the user could only use fixed text in the object definitions.

First of all, the variable buffer is cleared (in case variables have been previously defined. The call is also recommended to make sure that the variable buffer is empty, and no variables remain from the previous print job that might be meaningless in this next task):

```
LlDefineVariableStart(hJob);
```

Now the variables can be declared in various ways. If the Designer knows sample data for a variable, then these are used instead of the variable name in the preview window in order to guarantee a more realistic preview display.

```
LlDefineVariable(hJob, "forename", "George")
LlDefineVariable(hJob, "lastname", "Smith");
```

And so the expression

'forename +" "+lastname'

is transformed to

'George Smith'

If list objects are also required - in a 'report' or list project (*LL_PROJECT_LIST*) - then the programmer must also make the necessary fields available. This is done analogously to the above (barcode fields and drawings are also possible table columns), except that the function names contain "Field" instead of "Variable":

```
LlDefineFieldStart(hJob);
LlDefineField(hJob, "book");
LlDefineField(hJob, "ISBN");
LlDefineFieldExt(hJob, "ISBN", "40|15589|97531", LL_BARCODE_EAN13, NULL);
LlDefineFieldExt(hJob, "photo", "c:\\dwg\\test.bmp", LL_DRAWING, NULL);
```

Before calling *LlDefineLayout(),* menu items can be deleted by *LlDesignerProhibitAction()* or blocked so that the Designer cannot be minimized. The latter can be done by calling:

```
LlDesignerProhibitAction(hJob, LL_SYSCOMMAND_MINIMIZE);
```

Now everything has been defined sufficiently for the user to edit his project and the Designer can be started:

```
LlDefineLayout(hJob, hWindow, "test title", LL_PROJECT_LABEL, "test.lbl");
```

For a list, change the constant to *LL_PROJECT_LIST,* or for a file card project to *LL_-PROJECT_CARD*.

Please note that in the Designer you will generally see the data of only one record (multiple times) in the Designer. It is possible to provide a real data preview in the designer. See chapter "Direct Print and Export From the Designer" for further information.

Checking for error codes is generally recommended.

5.4 The Print Process

5.4.1 Supplying Data

List & Label also works database-independently in this mode. This means that the application is (i.e. you as a programmer are) responsible for supplying the data. You tell List & Label, by calling a function, which data (fields) are available in your application (e.g. "A Field called <Name>, a field called <Lastname> etc.") and which content this field should have. Where you get the data from at print time is totally irrelevant for List & Label. In most cases you probably perform a read access to a record field in a database.

To integrate the Designer into your application, you need to tell List & Label about the available data fields by calling a function for each of your data fields that may be placed in the form. With this call, you may also optionally declare the field type (e.g. text, numeric, boolean, date etc.) to List & Label, which is e.g. relevant for the correct treatment of your fields and variables in formulas within the Designer. You may pass a sample content for each field, which is used during the design in the layout preview. If you want to support the real data preview in the Designer, please follow the instructions in chapter "Direct Print and Export From the Designer".

During printing, the passing of data works analogously, apart from the fact that you have to supply the real data content instead of the sample content. This has to be done for all fields used, while you are iterating all records you want to print out.

5.4.2 Real Data Preview or Print?

In principle the printing loop always looks the same, whether you print to preview (*LL_PRINT_PREVIEW*), printer (*LL_PRINT_NORMAL*) or file (*LL_PRINT_FILE*). The only

difference is a parameter setting at the beginning of the print (see *LlPrint[WithBox]-Start()*). You may, however, leave the target choice to the end user (*LL_PRINT_EXPORT*) by giving him the option to choose the print target within the print dialog called by *LlPrintOptionsDialog()*.

5.4.3 Basic Procedure

First of all a List & Label job is opened (*LlJobOpen()* or *LlJobOpenLCID()*) and, if necessary, global List & Label options are set (*LlSetOption()*). Now List & Label has to be informed that printing should start (*LlPrintWithBoxStart()*). With this call, the project file to be used for printing is passed to List & Label. At this point, the project is opened and parsed by List & Label. During this process a syntax check is performed on all variables, fields and formulas used. This requires List & Label to know all variables and fields you are making available to your end users. For this reason, you must define all variables and fields using the *LlDefineVariable()* and *LlDefineField()* functions before calling *LlPrintWithBoxStart()*.

As only the names and types are important at this point, not the contents, you may call the same routine you use to define all variables and fields for the Designer (e.g. with a sample content, or the content of the first record).

A print usually proceeds as follows (functions with '*' are optional calls which are not necessarily required):

```
<open Job>
       (LlJobOpen, LlJobOpenLCID)
<define List & Label-settings>*
       (LlSetOption,
        LlSetOptionString,
        LlSetFileExtensions,
        LlSetNotificationMessage,
        LlSetNotificationCallback)
<print>                (see below)
<close Job>
       (LlJobClose)
```

Printing Labels and File Cards

For a label or file card print (*LL_PROJECT_LABEL*, *LL_PROJECT_CARD*), the <print> part looks as follows:

```
<define all possible variables>
       (LlDefineVariableStart,
        LlDefineVariable,
        LlDefineVariableExt,
        LlDefineVariableExtHandle)
<define options>*
       (LlSetPrinterDefaultsDir)
<start print>
```

```
       (LlPrintStart,
        LlPrintWithBoxStart)
<define print options>*
       (LlPrintSetOption,
        LlPrintSetOptionString,
        LlPreviewSetTempPath)
<let user change options>*
       (LlPrintOptionsDialog,
        LlPrintOptionsDialogTitle,
        LlPrintSelectOffsetEx,
        [LlPrinterSetup])
<define constant variables>
       (LlDefineVariable,
        LlDefineVariableExt,
        LlDefineVariableExtHandle)
<get printer info for progress-box>*
       (LlPrintGetOption,
        LlPrintGetOptionString,
        LlPrintGetPrinterInfo)
<skip unwanted labels>*

<print while data left and no error or user abort>
{
       <give progress-status>*
              (LlPrintSetBoxText,
               LlPrintGetCurrentPage,
               LlPrintGetOption)
       <define variables>
              (LlDefineVariable,
               LlDefineVariableExt,
               LlDefineVariableExtHandle)
       <print objects>
              (LlPrint)
       <no warning, no user abort: next data record>
}

<end print>
       (LlPrintEnd)
```

Printing Lists

And for printing a report (*LL_PROJECT_LIST*):

```
<define all possible variables>
       (LlDefineVariableStart,
        LlDefineVariable,
        LlDefineVariableExt,
        LlDefineVariableExtHandle)
<define all possible fields>
       (LlDefineFieldStart,
        LlDefineField,
        LlDefineFieldExt,
```

```
        LlDefineFieldExtHandle)
<define options>*
       (LlSetPrinterDefaultsDir)
<start print>
       (LlPrintStart,
        LlPrintWithBoxStart)
<define options>
       (LlPrintSetOption,
        LlPrintSetOptionString,
        LlSetPrinterDefaultsDir,
        LlPreviewSetTempPath)
<let user change options>*
       (LlPrintOptionsDialog,
        LlPrintOptionsDialogTitle,
        LlPrintSelectOffsetEx,
        [LlPrinterSetup])
<define constant variables>
       (LlDefineVariable,
        LlDefineVariableExt,
        LlDefineVariableExtHandle)
<print variables> (print all objects)
        LlPrint
<while "page full" warning (LL_WRN_REPEAT_DATA) do>
        LlPrint

<repeat >
{
       <define fields>
               (LlDefineField,
                LlDefineFieldExt,
                LlDefineFieldExtHandle)
       <print row>
               (LlPrintFields)
       <while "page full" warning (LL_WRN_REPEAT_DATA) do>
               <define page specific variables>*
                       (LlDefineVariable,
                        LlDefineVariableExt,
                        LlDefineVariableExtHandle)
               <re-print>
                       (LlPrint)
                       (LlPrintFields)
       <goto next data record>
       <give progress report>*
               (LlPrintSetBoxText,
                LlPrintGetCurrentPage,
                LlPrintGetOption)
}
<until
 -error or
 -no data records left or
 -user abort
>
```

```
<Print final footer and all linked objects>
      (LlPrintFieldsEnd)
<while "page full"-warning (LL_WRN_REPEAT_DATA) do>
      (LlPrintFieldsEnd)
<end print>
      (LlPrintEnd)
```

5.4.4 Annotations

Starting Print: Reading the Project File

Before the print can be started, it is important to know which project is to be loaded and which variables are to be made available.

After the optional dialog where the user can choose the project file, *LlSelectFileDlgTitleEx()*, all variables which are to be included in this project must be defined. If List & Label evaluates an expression in which there is an unknown variable, then it ends the loading process and passes back the corresponding error code. The definition of variables is programmed in the same way as the definitions before calling the Designer.

In the case of a list project (*LL_PROJECT_LIST*), the corresponding fields must also be defined in order to avoid an error.

Once you have called

```
LlPrintWithBoxStart(hJob, LL_PROJECT_LABEL, aczProjectFile, LL_PRINT_NORMAL,
              LL_BOXTYPE_BRIDGEMETER, hWindow, "my test");
```

and no error is returned from this function, List & Label has read the definition of the project and is ready to print. The printer is, however, not initialized yet - this is done with the first function call which starts the printing job.

If you want to allow the user to change the print parameters, the corresponding dialog is called using:

```
LlPrintOptionsDialog(hJob, hWindow, "Print Parameter");
```

With *LlSetOption()* and *LlSetOptionString()* standard values for that particular printout can be given, for example

```
LlPrintSetOption(hJob, LL_OPTION_COPIES, LL_COPIES_HIDE);
```

suppresses the "copies" query in the print options dialog.

If "Save options permanently" has been checked in the dialog then the chosen printer setting is saved in a so-called "printer definition file". Initially, the printer and layout is

determined in the Designer (menu: Project > Page Layout). If this file is not found, then the Windows standard printer is used. Further information on this can be found in chapter "List & Label Files".

List Projects: Important Things to Note

Variables - in the case of list projects - are values which remain constant for one page, and fields are the record-dependent data. These are printed using *LlPrintFields()*.

When calling *LlPrint()* the objects that are not lists are printed, as well as the list headers (if the option *LL_OPTION_DELAYTABLEHEADERLINE* is not set, otherwise the table header will be delayed until the first data line is to be printed). List & Label then expects the records to be defined.

With every *LlPrintFields()* it is tested whether the data record to be printed fits onto the current page. If this is not the case, *LL_WRN_REPEAT_DATA* is returned, which indicates that a new page should be started. In this case don't increment the record pointer.

When the table is full, the variables for the next page must be defined before calling *LlPrint()*, as with this *LlPrint()* any linked objects are printed, the new page is started and - see above - the objects on the new page including list headers printed.

A forced page break is possible by calling *LlPrint()* at any time, which ends the present page if this has already been partially filled.

Copies

"Copies" can mean two different kinds of copies:

a) Copies of labels usually do not mean that multiple pages should be printed, but the copies should be located on labels next to each other.

To support this, get the number of copies before the first *LlPrint()* call so that you know how many copies should be printed of each label, and set the copies for List & Label to 1, so that List & Label does not use printer copies too.

```
// user can change the number of copies...:
LlPrintOptionsDialog(hJob,hWnd,"Printing...");

nCopies = LlPrintGetOption(hJob,LL_PRNOPT_COPIES);
LlPrintSetOption(hJob,LL_PRNOPT_COPIES,1);
```

Then, print the requested number of labels:

```
for (nCopy = 1; (nCopy < nCopies) && (nError == 0); ++nCopy)
{
       nError = LlPrint(hJob);
       // support AUTOMULTIPAGE (usually memos in file cards)
       while (nError == LL_WRN_REPEAT_DATA)
```

```
                nError = LlPrint(hJob);
}
```

b) True copies of the sheets, that is, identical pages. This kind of copies is directly handled by List & Label, so no special handling from the developer is necessary.

Speed Optimization

a) Application optimization

At first, variable definitions which are to be constant during printing, can be pulled out of the print loop. If you want to always print your company name in the letter head with lists, it's best to define it outside the loop before *LlPrintWithBoxStart().*

b) Is the variable / field used?

You can also query which variables or fields are used in the expressions. If the number of potential variables or fields is much bigger than the actually used number or getting the data values is complex (sub queries, calculations, etc.) using these functions is worth it. Calling *LlGetUsedIdentifiers()* returns all variables and fields used in the project. *LlGetUsedIdentifiersEx()* furthermore allows to differentiate between the type (variable or field).

You should call this function before print start and later only pass the fields or variables from your data source which will actually be used.

c) Global "Dummy"-job

Some of the system libraries (e.g. riched20.dll) used by List & Label seem to cause resource losses under certain circumstances. These are very small but incur with every load and unload of the DLL.

These DLLs are loaded or unloaded by List & Label with every open or close of the "first" job. Therefore you should avoid a frequent *LlJobOpen()* / *LlJobClose()* in your application or to start a dummy job at start and keep it open until the end. The permanent loading and unloading of the DLLs is avoided and besides the achieved speed optimization also the resource losses won't occur anymore.

5.5 Printing Relational Data

List & Label offers a convenient way of designing projects with multiple relationally linked database tables (hierarchical reports). The report container is also the easiest way for the user to work with multiple tables, charts crosstabs or charts in table columns. The rest of this chapter handles working with *LL_PROJECT_LIST* type projects*. LL_PROJECT_LABEL* or *LL_PROJECT_CARD* type projects support exactly one table and an arbitrary number of sort orders for this table that can be set and retrieved just as for *LL_PROJECT_LIST* projects.

In the following we use "table" as a synonym for a group of matching fields in the List & Label-Designer or in the "Objects" tool window. You are not restricted to "real" databases - it is also possible to display a class array or dynamically created data in

a "table", or all member variables of a class. In the List & Label-Designer you will work with just one "report container object". This object can contain multiple tables, crosstabs and charts.

Once you have added single tables with *LlDbAddTable(),* your users can edit the structure in the "Objects" tool window. You will find further information on how to design the report container object in the corresponding chapter of the Designer manual. This chapter focuses on how to control such designs.

Examples of how to use multiple tables for the most common programming languages are included in the installation.

5.5.1 Using a Custom Print Loop

If your programming environment is not able to work with COM interfaces, you can support relational data by coding your own print loop. A few features (e.g. multiple report containers on one page) are not available. If you have the possibility, we recommend using the ILLDataProvider interface as described in chapter "Using the ILLDataProvider Interface".

API Functions Needed

The name of the API functions needed to control this functionality begin with *LlDb…* or *LlPrintDb…*. You can add tables (*LlDbAddTable()*), define sortings for the tables (*LlDbAddTableSortOrder()*) and define relations between tables (*LlDbAddTableRelation()*).

At print time you can query the currently active table (*LlPrintDbGetCurrentTable()*) as well as the currently active relation and sort order (*LlPrintDbGetCurrentTableSortOrder()*, *LlPrintDbGetCurrentTableRelation()*). You will find detailed descriptions later in this chapter.

Calling the Designer

First all tables have to be declared to List & Label, so that they can be inserted into the project:

```
LlDbAddTable(hJob, "", ""); // delete existing tables
LlDbAddTable(hJob, "Orders", "ORDERS");
LlDbAddTable(hJob, "OrderDetails", "ORDER DETAILS");
```

The first parameter is the usual job handle of the List & Label job. The second parameter is the table ID, which will be returned during printout by *LlPrintDbGetCurrentTable()*. The third parameter is the display name of the table in the Designer. If you pass NULL or an empty string, the table ID will be used as display name as well.

A special role is assigned to the table name "LLStaticTable". This is reserved and can be used for the insertion of 'static' contents (fixed texts or contents of variables, chart signatures etc.). This type of static table is then available as "Free content"

data source and can only be filled with data lines by the user in the Designer. You must react accordingly to the table in your code - a detailed explanation is provided in the Printing subchapter.

In the next step the relations between the tables will be defined. List & Label does not directly differ between different types relationships (n:m, 1:n) – you declare a relation with a relation ID which can be queried at print time:

```
LlDbAddTableRelation(hJob, "OrderDetails", "Orders",
       "Orders2OrderDetails", NULL);
```

With this command, you have established a relationship between the child table "OrderDetails" and the parent table "Orders". In this case only the ID of the relation was passed and will be displayed in the Designer.

Finally you can pass sort orders for the tables. Again, you define a unique ID for every sort order that can then be queried at print time:

```
LlDbAddTableSortOrder(hJob, "Orders", "OrderDate ASC",
       "Order Date [+]");
LlDbAddTableSortOrder(hJob, "Orders", "OrderDate DESC",
       "Order Date [-]");
```

This allows the user to choose one of these sort orders (as well as the default "unsorted") in the Designer. If you use *LlDbAddTableEx()* to define the tables, you can also support multiple (stacked) sortings.

The remaining action when calling the Designer is analogous to the "normal" call, i.e. the complete scheme for calling the Designer with multiple tables looks like this:

```
<open job>
       (LlJobOpen, LlJobOpenLCID)
<define List & Label-settings>
       (LlSetOption,
        LlSetOptionString,
        LlSetDebug,
        LlSetFileExtensions,
        LlSetNotificationMessage,
        LlSetNotificationCallback)
<which file?>
       LlSelectFileDlgTitleEx
<define data structure>
       (LlDbAddTable,
        LlDbAddTableRelation,
        LlDbAddTableSortOrder)
<define variables>
       (LlDefineVariableStart,
        LlDefineVariable,
```

```
        LlDefineVariableExt,
        LlDefineVariableExtHandle)
<define fields>
       (LlDefineFieldStart,
        LlDefineField,
        LlDefineFieldExt,
        LlDefineFieldExtHandle)
<disable funtions>
       (LlDesignerProhibitAction,
        LlDesignerProhibitFunction)
<call designer>
       (LlDefineLayout)
<close job>
       (LlJobClose)
```

Make sure that you pass all field names in the form of "<tableid>.<fieldname>" in order to enable List & Label can connect these to their corresponding table (e.g. "Orders.OrderID").

If you want to add fields of a 1:1 relation, please refer to chapter "Handling 1:1 Relations".

Controlling the Print Engine

The control of hierarchical reports with List & Label occurs more or less analogously to the print flow in the last chapter. With the function *LlPrintDbGetCurrentTable()* you can query which table's values are to be passed – as usual with *LlDefineField[Ext]()* and *LlPrintFields()*. Depending on the layout, there are two specific cases:

- Tables can be consecutive (multiple tables following each other at the same level)
- The user could have added a sub-table to the current table

We will deal with these cases in the next two sections.

Multiple Independent Tables on the Same Level

An example for this would be a list of customers followed by a chart of employees. Both tables can be independent. The print loop for this looks very similar to the print loop in the last chapter – with one difference. Usually, you tell List & Label that a table is finished (no more data) by calling *LlPrintFieldsEnd()*. Now you may get the return value *LL_WRN_TABLECHANGE*, meaning that there is another table to print in the layout.

We suggest splitting your print loop into different subroutines.

The first part declares the data and the structure, starts the print job and initializes the first page so that printing of a table can be started. For ease of reading, the optional part of the print loop is not shown here, as it has already been shown in the last chapter.

```
<define data structure>
       (LlDbAddTable,
        LlDbAddTableRelation,
        LlDbAddTableSortOrder)
<define all possible variables>
       (LlDefineVariableStart,
        LlDefineVariable,
        LlDefineVariableExt,
        LlDefineVariableExtHandle)
<define all possible fields>
       (LlDefineFieldStart,
        LlDefineField,
        LlDefineFieldExt,
        LlDefineFieldExtHandle)
        LlSetPrinterDefaultsDir
<begin print>
       (LlPrintStart,
        LlPrintWithBoxStart)
<define options>
       (LlPrintSetOption,
        LlPrintSetOptionString,
        LlPreviewSetTempPath)
<define fixed variables>
       (LlDefineVariable,
        LlDefineVariableExt,
        LlDefineVariableExtHandle)
<print variables>       (print all objects)
              (LlPrint)
<as long as warning repeat>
              (LlPrint)
```

The second part of the print loop needs an auxiliary function. This function prints the data of a single (database) table

```
function PrintTable(DataTable Dataobject)
{
       // DataTable is an adequate object for data access, e.g. a
       // table of a database, a class array or similar

       <repeat>
       {
              <define fields of DataTable>
              (LlDefineField,
               LlDefineFieldExt,
               LlDefineFieldExtHandle)
              <print line>
              (LlPrintFields)
              <as long as warning repeat >
                     (LlPrint,
                      LlPrintFields)
```

```
                <next data record in DataTable>
        }
        <until last data record in DataTable reached>

        <print footer line>
                (Ret = LlPrintFieldsEnd)
        <as long as warning "page full" repeat>
                (Ret = LlPrintFieldsEnd)
        <result = Ret>
}
```

The return value specifies whether another table follows (*LlPrintFieldsEnd()* returns *LL_WRN_TABLECHANGE*) or if the print can be finished (return value 0).

With this function, the second part of the print – the part after the initialization of the first page – can be coded as follows:

```
<repeat>
{
        <get current table name >
                (LlPrintDbGetCurrentTable)
        <get current sorting>
                (LlPrintDbGetCurrentTableSortOrder)
        <generate a corresponding DataTable object>
        <Ret=PrintTable(DataTable)>
}
<until Ret <> LL_WRN_TABLECHANGE>

<finish printout>
        (LlPrintEnd)
```

If you have declared the "LLStaticTable" table for free contents and *LlPrintDbGetCurrentTable()* provides this table as the current table, your printing loop must react to it by printing a single data line via LlPrintFields(). In the above example, you could simply generate a DataTable object with just one data record for the case of "LLStaticTable", and printing will then automatically run correctly.

This code already allows an arbitrary sequence of multiple tables in series. In the following chapter, we will expand it to print sub-tables as well.

Simple 1:n Relations

The typical example for this case is the previously discussed 1:n relation order – order details. After each record with order data, the order details for that data shall be printed.

The printing of a data line is triggered by *LlPrintFields()*. Analogously to the behavior of *LlPrintFieldsEnd()* in the last section, the function returns *LL_WRN_TABLECHANGE* if the user has placed a sub-table, and you then have to respond.

You can ask for the table relation with *LlPrintDbGetCurrentRelation()* and for the name of the child table with *LlPrintDbGetCurrentTableName()*. With this information, you can invoke the auxiliary function *PrintTable()* from the last section again. This call must be placed directly after *LlPrintFields()* – thus from the function *PrintTable()* itself. The function must be changed in order to call itself recursively:

```
function PrintTable(DataTable data object)
{
       // DataTable is an adequate object for data access, e.g. a
       // table of a database, a class array or similar

       <repeat>
       {
              <define fields of DataTable>
                     (LlDefineField,
                      LlDefineFieldExt,
                      LlDefineFieldExtHandle)
              <print row>
                     (LlPrintFields)
              <as long as warning repeat>
                     (LlPrint,
                      Ret = LlPrintFields)
              <as long as Ret = LL_WRN_TABLECHANGE repeat>
              {
                     <get current table name>
                            (LlPrintDbGetCurrentTable)
                     <get current relation>
                            (LlPrintDbGetCurrentTableRelation)
                     <get current sorting>
                            (LlPrintDbGetCurrentTableSortOrder)
                     <generate an appropriate DataTable child object>
                     <Ret = PrintTable(child DataTable)>
              }

              <next record in DataTable>
       }
       <until last record in DataTable is reached>

       <print footer line>
              (Ret = LlPrintFieldsEnd)
       < as long as warning "page full" repeat >
              (Ret = LlPrintFieldsEnd)
       <result = Ret>
}
```

Any sequence of tables and sub-tables can be printed with this code. The recursion ensures that it works properly with any "depth", i.e. this code can control arbitrary multilevel relations.

The Recursive Print Loop

For a complete print loop which is supporting sequence tables und sub-tables, there is nothing more to do. The code from the last two sections makes sure that the complete tree of the table structure is printed.

So only the finishing touches have to be added – e.g. to display a progress bar. The structure of a layout can be quite complex. Thus it is not possible to just take the current position inside the data source as percentage. This approach does not work as soon as the user puts two tables in series. Therefore List & Label allows you to get the count of tables at the root level (*LlPrintDbGetRootTableCount()*). Whenever you display a data record from the root level, you can update the progress bar.

The following holds for the maximum available percentage of a table:

```
INT nMaxPerc = 100/LlPrintDbGetRootTableCount();
```

If you index the root tables from 0.. *LlPrintDbGetRootTableCount()-1*, you can calculate the total percentage as

```
INT nPercTotal = nMaxPerc*nIndexCurrentTable+(nPerc/100*nMaxPerc);
```

where *nPerc* is the percentage position in the current table. To properly update the progress bar, you can adapt the function *PrintTable()* from the last section. The current depth of the recursion can be determined with another input parameter – if this parameter is 0, a "root" data record is printed and the progress bar can be updated:

```
function PrintTable(DataTable data object, depth of recursion depth)
{
       <repeat>
       {
              <define fields of DataTable>
              ...
              <if depth==0 update progress bar>
                     (LlPrintDbGetRootTableCount,
                      LlPrintSetBoxText)
              <print row>
                      (LlPrintFields)
              <as long as warning repeat >
                     (LlPrint,
                      Ret = LlPrintFields)
              <repeat until Ret <> LL_WRN_TABLECHANGE >
              {
                     ...
                     <generate an appropriate DataTable child object>
                     <Ret = PrintTable(child DataTable, depth+1)>
              }
```

```
                ...
        }
}
```

Supplying Master Data as Variables

In the case of an order with the relevant order details, it could be desirable to offer the "master" data, i.e. in this example the data of the table orders, as variables. So the addressee could e.g. be displayed in a text object and the order details in a table object. Therefore, at the root level of the table object you need to have access to the sub-tables of the master table. In order to achieve this, call *LlDbSetMasterTable()*. The necessary calls are

```
LlDbAddTable(hJob, "Orders", "");
LlDbAddTable(hJob, "OrderDetails", "");
LlDbAddTableRelation(hJob, "OrderDetails", "Orders",
        "Orders2OrderDetails", NULL);
LlDbSetMasterTable(hJob, "Orders");
```

The print loop is analogous to the description above, but you have to make the appropriate child table available on the top level (see chapter "Multiple Independent Tables on the Same Level"):

```
<repeat>
{
        <get current table name>
                (LlPrintDbGetCurrentTable)
        <get current sorting>
                (LlPrintDbGetCurrentTableSortOrder)
        <get current relation>
                (LlPrintDbGetCurrentTableRelation)
        <if relation empty>
                <generate an appropriate DataTable object>
        <else>
                <generate an appropriate child DataTable object>
        <Ret = PrintTable(DataTable)>
}
<until Ret <> LL_WRN_TABLECHANGE>

<close printing>
        (LlPrintEnd)
```

5.5.2 Using the ILLDataProvider Interface

Instead of a manual implementation of the print loop, you can also use the ILL-DataProvider Interface. This is the most flexible and convenient way for the developer to use the API.

Advantages

By using the interface instead of an individual implementation, most of the print logic can be mapped automatically directly within List & Label.

- Generally better reusability and thus maintainability of the code.
- Many of the advanced features that are already available in .NET are also available in this way.
- Use of several report containers next to each other.
- Nesting of tables.
- Increased performance due to delayed loading of content.

Prerequisites

First, the general requirements for programming via API apply. Since the ILLDataProvider interface is based on a COM interface, the programming language used must support the use of Microsoft's Component Object Model. You may then implement the ILLDataProvider interface depending on the required functionality.

Calling the Designer

The call of the designer is similar to the call in general API programming. If the option LL_OPTION_SUPPORT_DELAYEDFIELDEFINITION is set, the steps marked with * do not need to be executed in advance. List & Label then queries sort orders, variables and fields at the required time via the data provider itself.

```
// Initialization
<create instance of your own ILLDataProvider implementation>
<create print job, set parameters and the data provider>
       (LlJobOpen, LlSetOption)
<define data structure>
       (LlDbAddTable,
 LlDbAddTableRelation,
 LlDbAddTableSortOrder*)
<define all possible variables>
       (LlDefineVariableStart,
        LlDefineVariable*,
        LlDefineVariableExt*,
        LlDefineVariableExtHandle*)
<define all possible fields>
       (LlDefineFieldStart,
        LlDefineField*,
        LlDefineFieldExt*,
        LlDefineFieldExtHandle*)

// Job, Designer
<start Designer>
       (LlDefineLayout)

// Deinitialization
<detach Data provider and close job>
```

```
       (LlSetOption,
        LlJobClose)
```

Controlling the printing process

Initialization and deinitialization are carried out in the same way as the Designer call. The print loop itself is as short as possible, since the actual logic is now within List & Label.

```
// Initialization
       (...)

// Job, print loop
<Start printing>
       (LlPrintWithBoxStart)
<repeat>
{
(LlPrint)
}

<Exit print>
       (LlPrintEnd)

// Deinitialization
<detach Data provider and close job>
       (LlSetOption,
        LlJobClose)
```

Required API functions and interface

With the supplied Visual C++ example "Print and Design Reports (SQL data source)" you will find an executable example for the implementation.

The pseudo code for designer call and print loop shows easily that the set of required API functions differs only slightly from the classic API programming. However, some parts are relocated into the data provider implementation.

In the general initialization part after LlJobOpen, the data provider instance must first be made known and, optionally, the delayed loading must be activated. At this point, you would also register the callbacks for preview and drilldown.

```
auto pDP = (ILLDataProvider*) new MyDataProviderObject;
::LlSetOption(hJob,LL_OPTION_ILLDATAPROVIDER,(LPARAM)pDP);
::LlSetOption(hJob, LL_OPTION_SUPPORT_DELAYEDFIELDDEFINITION, 1);
```

To create the data provider instance, you need at least one class that implements the ILLDataProvider interface. In addition, QueryInterface, AddRef and Release from IUnknown must also be implemented. List & Label distinguishes internally between

root objects and nodes. This distinction can now be made either across several classes (see example below) or, for simplicity's sake, within one class.

Methods that are not implemented return E_NOTIMPL.

```
#define SMI STDMETHODIMP
class DPBase : public ILLDataProvider
{
…

// From ILLDataProvider
SMI OpenTable(LPCWSTR pszTableName, IUnknown** ppUnkOfNewDP) = 0;
SMI OpenChildTable(LPCWSTR pszRelation, IUnknown** ppUnkOfNewDP) = 0;
SMI GetRowCount(INT* pnRows) = 0;
SMI DefineDelayedInfo(enDefineDelayedInfo nInfo) = 0;
SMI MoveNext() = 0;
SMI DefineRow(enDefineRowMode, const VARIANT* arvRelations) = 0;
SMI Dispose() = 0;
SMI SetUsedIdentifiers(const VARIANT* arvFieldRestriction) = 0;
SMI ApplySortOrder(LPCWSTR pszSortOrder) = 0;
SMI ApplyFilter(const VARIANT* arvFields, const VARIANT* arvValues) = 0;
SMI ApplyAdvancedFilter(LPCWSTR pszFilter, const VARIANT* arvValues) = 0;
SMI SetOption(enOptionIndex nIndex, const VARIANT * vValue) = 0;
SMI GetOption(enOptionIndex nIndex, VARIANT * vValue) = 0;

…
};
class DPRoot : public DPBase{ … };
class DPNode : public DPBase{ … };
```

OpenTable (ILLDataProvider)

Syntax:

```
HRESULT OpenTable(LPCWSTR pszTableName, IUnknown** ppUnkOfNewDP);
```

Task:

Only used at root level. Requests the creation of a new node below the root and returns the new interface to List & Label.

Parameter:

pszTableName: The requested table name

ppUnkOfNewDP: Target address for the new object

Return value:

E_NOTIMPL for node objects, otherwise S_OK or E_FAIL in case of an error

Hints:

Required as an entry point for the data provider. List & Label delegates the actual work to the created nodes and subnodes.

Example:

```
*ppUnkOfNewDataProvider = new DPNode(_hJob, pszTableName);
return S_OK;
```

See also:

OpenChildTable (ILLDataProvider)

OpenChildTable (ILLDataProvider)

Syntax:

```
HRESULT OpenChildTable(LPCWSTR pszRelation, IUnknown** ppUnkOfNewDP);
```

Task:

Only used at node level. Requests the creation of a new sub-node and returns the new interface to List & Label.

Parameter:

pszRelation: The requested relation name or table name

ppUnkOfNewDP: Target address for the new object

Return value:

E_NOTIMPL for root objects, otherwise S_OK or E_FAIL in case of error

Hints:

The data provider is able to create the requested object hierarchies controlled by List & Label using OpenChildTable.

Example:

```
*ppUnkOfNewDataProvider = new DPNode (_hJob, this, pszRelation);
return S_OK;
```

See also:

OpenTable (ILLDataProvider)

GetRowCount (ILLDataProvider)

Syntax:

```
HRESULT GetRowCount(INT* pnRows);
```

Task:

Only used at node level. Requests the number of data lines of the data object. If no record number is available due to performance limitations, simply return S_FALSE. In this case, List & Label does not support a progress bar.

Parameter:

pnRows: Target address for the number of data rows

Return value:

E_NOTIMPL for root objects, otherwise S_OK or S_FALSE or E_FAIL in case of an error

Example:

```
*pnRows = _count;
return S_OK;
```

See also:

SetOption (ILLDataProvider), OPTION_HINT_MAXROWS

DefineDelayedInfo (ILLDataProvider)

Syntax:

```
HRESULT DefineDelayedInfo(enDefineDelayedInfo nInfo);
```

Task:

Only used at node level. If the option LL_OPTION_SUPPORT_DELAYEDFIELDEFINITION is set, then List & Label requests the definition of the sort orders via this method.

Parameter:

nInfo: One of the following values

DELAYEDINFO_SORTORDERS_DESIGNING, asks for the sort orders at design time.

DELAYEDINFO_SORTORDERS_PRINTING, asks for the sort orders at print time.

Return value:

E_NOTIMPL for root objects, otherwise S_OK or E_FAIL in case of error

Hints:

Empty sort orders can also be returned with S_OK.

Example:

```
for (auto& column_rec : pTableRec->_columns)
   DefineSortOrders(pTableRec, column_rec, nInfo);
return S_OK;
```

See also:

LIDbAddTableSortOrder, LIDbAddTableSortOrderEx

MoveNext (ILLDataProvider)

Syntax:

```
HRESULT MoveNext();
```

Task:

Only used at node level. List & Label requests to move the cursor of the data object to the next row.

Parameter:

None

Return value:

E_NOTIMPL for root objects and E_FAIL in the event of an error. Usually S_OK if successful or S_FALSE if no more data is available.

Hints:

In order for a cursor to be moved, the data object usually has to already exist completely. In the case of an SQL provider, this means that the required query has already been assembled and initiated at this point in time. You should initialize your enumerator exactly now (and not earlier) as MoveNext () is called initially to get the first row of data.

DefineRow (ILLDataProvider)

Syntax:

```
HRESULT DefineRow(enDefineRowMode nMode, const VARIANT* arvRelations);
```

Task:

Only used at node level. List & Label requests to pass the data of the current record.

Parameter:

```
enum enDefineRowMode
{
        ROWMODE_DEFAULT = 0, // internal, not yet queried
        ROWMODE_OWN_COLUMNS = 1, // bit 0
        ROWMODE_1TO1_COLUMNS = 2, // bit 1
        ROWMODE_ALL_COLUMNS = 3, // bit 0 | bit 1
        ROWMODE_COLUMN_MASK = 0x0f,
        ROWMODE_DATA_PRINT_SYNTAXPARSING = 0x00, and
        ROWMODE_DATA_DESIGN = 0x10,
        ROWMODE_DATA_PRINT_REALDATA = 0x20,
```

```
            ROWMODE_DATA_MASK = 0xf0,
            ROWMODE_FIELD = 0x100,
    };
```

nMode: Bitmask for the type of data

arvRelations: Not used

Return value:

E_NOTIMPL for root objects and E_FAIL in the event of an error. Usually S_OK.

Hints:

Pass the requested fields and variables to List & Label here.

See also:

LIDefineFieldExt, LIDefineVariableExt

Dispose (ILLDataProvider)

Syntax:

```
HRESULT Dispose();
```

Task:

Only used at node level. Resources such as database connections can be released here.

Parameter:

None

Return value:

E_NOTIMPL for root objects. Usually S_OK.

SetUsedIdentifiers (ILLDataProvider)

Syntax:

```
HRESULT SetUsedIdentifiers(const VARIANT* arvFieldRestriction);
```

Task:

Only used at node level. List & Label informs the provider which fields are required. Use this method, for example, to restrict an underlying SQL (select) statement in the data object.

Parameter:

arvFieldRestriction: BSTR Variant Array with the maximum number of fields to be requested.

Return value:

E_NOTIMPL for root objects, otherwise S_OK

Hints:

If no used identifiers are set here, there is no restriction to set - all fields are then requested.

Example:

```
_usedIdentifiers.clear();// to be used when building query later
for (int i = 0;
     i   <   int(arvFieldRestriction->parray->rgsabound[0].cElements);
     ++i)
{
        VARIANT vItem;
        long lIndex = i;
        SafeArrayGetElement(arvFieldRestriction->parray,
                 &lIndex, &vItem);
        std::wstring ws(vItem.bstrVal, SysStringLen(vItem.bstrVal));
        _usedIdentifiers.insert(ws.c_str());
}
```

See also:

LIGetUsedIdentifiers

ApplySortOrder (ILLDataProvider)

Syntax:

```
HRESULT ApplySortOrder(LPCWSTR pszSortOrder);
```

Task:

Only used at node level. List & Label sets the required sort order via this method. Use it, for example, to add an ORDER BY clause to your SQL statement.

Parameter:

pszSortOrder: Tab separated string with the requested sort orders.

Return value:

E_NOTIMPL for root objects, otherwise S_OK

See also:

LIGetUsedIdentifiers

ApplyFilter (ILLDataProvider)

Syntax:

```
HRESULT ApplyFilter(const VARIANT* arvFields,
                    const VARIANT* arvValues);
```

Task:

Only used at node level. In the drilldown case, List & Label transfers a field list and its contents. For example, you can use WHERE to restrict an underlying SQL statement in the data object.

Parameter:

arvFields: VARIANT field with the field names

arvValues: VARIANT field of equal size with the field contents

Return value:

E_NOTIMPL for root objects, otherwise S_OK or E_FAIL in case of error

Hints:

Empty filter requests can also be answered with S_OK.

Example:

```
SAFEARRAY* pArray = arvFields->parray;
SAFEARRAY* pValArray = arvValues->parray;
for (ULONG i = 0; i < pArray->rgsabound[0].cElements; ++i)
{
   CSafeVARIANT vHelper, vValHelper;
   long     lResIndex[2] = { i, 1 };
   ::SafeArrayGetElement(pArray, lResIndex, &vHelper);
   ::SafeArrayGetElement(pValArray, lResIndex, &vValHelper);
   _sqlparms._ddwhere += xsprintf(L"%ls%ls = %ls",
         _sqlparms._ddwhere.empty() ? L"WHERE " : L"AND ",
         (LPCTSTR)MakeDBColumnString(this, vHelper));
         (LPCTSTR)MakeDBValueString(this, vHelper, vValHelper));
}
return S_OK;
```

SetOption (ILLDataProvider)

Syntax:

```
HRESULT SetOption (enOptionIndex nIndex, const VARIANT* vValue);
```

Task:

Only used at node level. List & Label transfers additional information about the status of the data provider. These can be used in the implementation for optimizations.

Parameter:

nIndex: One of the following values

OPTION_HINT_MAXROWS, is set, for example, to restrict an underlying SQL statement in the data object to vValue rows.

OPTION_HINT_IS_INFO_QUERY, is set if the data provider only queries structure information to dynamically fill the field names in the variable tree in List & Label Designer, for example.

All other constant values are currently only for internal use and can be ignored.

vValue: The variable value

Return value:

E_NOTIMPL for root objects, otherwise S_OK

GetOption (ILLDataProvider)

Syntax:

```
HRESULT GetOption (enOptionIndex nIndex, VARIANT* vValue);
```

Task:

Only used at node level. List & Label queries additional information with this.

Parameter:

nIndex: One of the following values

OPTION_SCHEME_AND_DEFAULTS, is used to optimize schema queries. By default, this option must be ignored first or the result in vValue must be set to OPTION_SCHEMAROWUSAGEMODE_NONE.

OPTION_SUPPORTED_AS_1_TO_1_RELATION, is used to determine whether a given relation can also be resolved "backwards" as a 1:1 relation. In this case, the name of the relation is passed with vValue and the return value is to be returned with S_FALSE or S_OK.

vValue: Pointer to variant for data exchange

Return value:

E_NOTIMPL for root objects, otherwise dependent on nIndex.

5.5.3 Handling 1:1 Relations

When reporting 1:1 relations, the data is usually combined by a database query with a SQL JOIN, so that the data is available in one table. If this is not the case or if you do not want this, you can display the 1:1 relations in the list of variables below the fields

of the parent table. To accomplish this, you have to declare the fields in a special syntax.

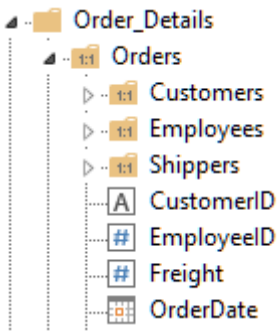
1:1 Relations Without a Key Field Definition

If the key fields for the relation are not relevant - if, for example, you are dealing with a trivial, single 1:1 relation between the two connected tables - you can declare the fields as follows:

<parent table>:<linked table>.<field name>, e.g.

OrderDetails:Orders.OrderDate

This adds a folder with the field OrderDate to the list of variables below the OrderDetails hierarchy:



Of course, you must make sure that when printing the OrderDetails table, you fill this field with the corresponding value for each record.

1:1 Relation With Key Field Definition

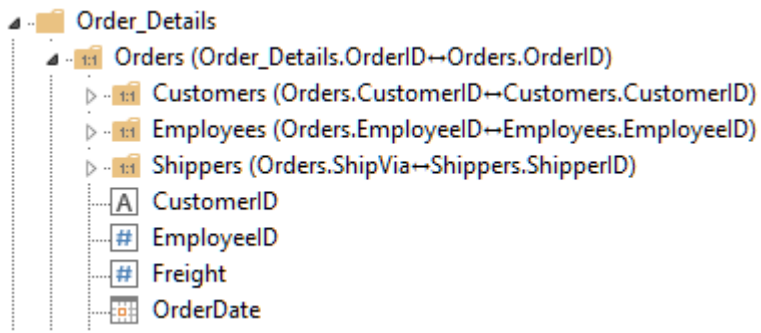
In case of multiple 1:1 connections, it might be important for the user to see which of the key fields are linking the tables together. In this case you can declare the fields as follows:

<parent table>.<key field parent table>@<linked table>.<key field linked table>:<field name>, e.g.

OrderDetails.OrderID@Orders.OrderID:OrderDate

(SQL equivalent: "SELECT OrderDate FROM Orders WHERE OrderDetails.OrderID= Orders.OrderID")

Now the key field declaration is displayed in the tool window list of variables next to the table name:



Again, remember to update the field contents when the parent table is printed!

Performance Hints

When using 1:1 relations, it is very important to check whether the user has actually placed a field of the linked table. You can do this by using the wildcard option with *LlPrintIsFieldUsed()*. If you want to check whether a field of the 1:1 linked table Orders inside the table OrderDetails is being used, you can call

```
LlPrintIsFieldUsed(hJob, "OrderDetails.OrderID@Orders.OrderID*");
```

If the result is 0, no field of the table orders is being used and it is not necessary to update the values.

5.6 Callbacks and Notifications

This chapter is only required if you are not working with one of the components (.NET/VCL/OCX). If you are using one of these components, you may skip this chapter.

5.6.1 Overview

The following principle is to be understood by the expressions "callbacks and notifications": when List & Label needs information then it just asks your program. You don't have to pre-program all answers, just those for which you explicitly wish a modified behavior.

For example, there are objects which are program definable (user objects, see next chapter) which are handled by List & Label as a "black box". And when List & Label has to print such an object it turns to your program to ask it to carry out this function. This allows your application to extend the support for special objects, for example graphics with formats not implemented in List & Label. This is easier than a whole COM interface, because you need to supply only one function for the job.

Using the callback function you can add data to a page (important for labels: when needed you can add information like page no., print date or similar onto a page outside of the labels) which is controlled by the programmer (and consequently cannot be

removed by the user in the Designer). Objects can be hidden (this can also be done with *LlPrintEnableObject()* or the appearance condition of an object).

All this is possible if you implement one of the following:

- a callback routine is defined and its address is passed on to List & Label by *LlSetNotificationCallback()*, or
- you react to messages sent by List & Label via Windows messages. These are sent by List & Label to the window which is stated with *LlDefineLayout()* and *LlPrintWithBoxStart()*.

In both cases you obtain detailed information about the function which is to be carried out.

The rest of this chapter describes how to implement such a callback routine. For an overview of all available callbacks, see chapter "Callback Reference".

5.6.2 User Objects

As List & Label cannot draw all possible objects - be they spline objects, statistic graphs or drawings with an unknown format - a function has been built into List & Label to offer the programmer so-called user objects, as even metafile variables cannot cover all areas.

If you have defined a variable in your program with

```
LlDefineVariableExt(hJob, <Name>, <Content>, LL_DRAWING_USEROBJ, NULL);
```

the user can define an object in the Designer which is connected to this variable. This takes place analogously to normal *LL_DRAWING* variables.

When List & Label needs to print this object, it calls your program using the callback *LL_CMND_DRAW_USEROBJ* to pass this task on to your program, as List & Label has no idea what kind of "visual" action needs to be taken.

The same can be done for table fields, so that the user has the capability of including an user object in a table:

```
LlDefineFieldExt(hJob, <Name>, <Content>, LL_DRAWING_USEROBJ, NULL);
```

For variables, but not for fields, it is also possible to define user objects whose parameters can be changed by the user in the Designer, just like the object properties of List & Label's own objects. These objects are defined with the *LL_DRAWING_USEROBJ_DLG* type:

```
LlDefineVariableExt(hJob, <Name>, <Content>, LL_DRAWING_USEROBJ_DLG, NULL);
```

(This means that editable user objects cannot be inserted in tables. Only non-editable user objects can be used as table field.)

If the user selects the image in the Designer and clicks on the Variable's "Properties" sub item in the property window, the callback *LL_EDIT_USEROBJ* is invoked to request a dialog in which the parameters belonging to the object can be changed. These parameters are automatically stored in the project definition file with the other object information and passed with the callback *LL_DRAW_USEROBJ* for evaluation, so that your program does not have to take care of further storage of the parameters.

Note that you should only work with DIB (device independent bitmaps) in callback objects, DDB (device dependent bitmaps) can be incompatible with the respective printer DC.

5.6.3 Definition of a Callback Routine

A callback routine is defined like a normal Windows callback. For special features such as compiler switches, please refer to your compiler manual.

The general form of a callback is, in C

```
LRESULT CALLBACK _extern LLCallback(INT nMsg, LPARAM lParam, UINT_PTR
lUserParam);
```

You can pass the routine pointer immediately:

```
LlSetNotificationCallback(hJob, LLCallback);
```

From now on your routine will be called by List & Label if necessary.

At the end of the program it is important that the callback is set to NULL, otherwise your system will raise an unhandled exception.

```
LlSetNotificationCallback(hJob, NULL);
```

5.6.4 Passing Data to the Callback Routine

The value of the nMsg parameter determines the different functions. The values are the constants which begin with *LL_CMND_xxxx*, e.g. *LL_CMND_TABLEFIELD*, *LL_NTFY_xxxx* or *LL_INFO*.

Depending on the task your program has to perform, the parameter lParam has different meanings. The individual meanings are described later on. They are mostly structures (records) which lParam points to, so the value must therefore be cast by a type conversion to a structure pointer:

```
LRESULT CALLBACK _extern
       LLCallback(INT wParam, LPARAM lParam, UINT_PTR lUserParam)
{
       PSCLLTABLEFIELD pSCF;

       switch (wParam)
       {
               case LL_CMND_TABLEFIELD:
                       pSCF = (PSCLLTABLEFIELD)lParam;
                       // do something using pSCF;
                       break;
       }
       return(0);
}
```

The function must always return a defined value. Unless stated otherwise, this value should be zero.

lUserParam is the value passed by

```
LlSetOption(hJob, LL_OPTION_CALLBACKPARAMETER, <Value>)
```

This can be used to store an object pointer ("this", "self") in object-oriented languages.

5.6.5 Passing Data by Messages

Every message has three parameters: nMsg, wParam and lParam in the following definition of your message callback (this is called a window procedure, but is nothing but a callback!)

```
LRESULT WINAPI MyWndProc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM
lParam);
```

The message value which List & Label uses can be queried by *LlGetNotificationMessage()*. If the default setting is not suitable for your purposes (by definition a unique value) another can be chosen with *LlSetNotificationMessage()*.

wParam is once again our function index and *lParam* points to an intermediate structure of the type scLlCallback:

```
struct scLlCallback
{
       int      _nSize;
       LPARAM   _lParam;
       LRESULT  _lResult;
       UINT_PTR _lUserParameter;
}
```

The necessary *_lParam* (as parameter value) and *_lResult* (as return value) are stored in this structure.

```
nLLMessage = LlGetNotificationMessage(hJob);
//....
//...in the window procedure...
if (wMsg == nLLMessage)
{
       PSCCALLBACK      pSC;
       PSCLLTABLEFIELD pSCF;

       pSC = (PSCCALLBACK)lParam;
       switch (wParam)
       {
               case LL_CMND_TABLEFIELD:
                       pSCF = (PSCLLTABLEFIELD)pSC->_lParam;
                       // do something;
                       pSC._lResult = 0;
                       break;
       }
}
```

_lUserParam is the value passed by

```
LlSetOption(hJob, LL_OPTION_CALLBACKPARAMETER, <value>)
```

This can be used to store an object pointer ("this", "self") in object oriented languages.

When no special return value is needed, the *_lResult* field doesn't need to be changed, as it is set to 0 by default.

5.6.6 Further Hints

Some callback structures for drawing operations contain two device contexts. Both are identical and kept only for backward compatibility reasons..

If you select a GDI object in this DC or make other changes, e.g. change the mapping mode, you should reverse the changes before ending the routine.

Tip: the Windows API functions *SaveDC()*, *RestoreDC()* can help considerably for complex changes.

5.7 Advanced Programming

This chapter is only required if you are not working with one of the components (.NET/VCL/OCX). If you are using one of these components, you may skip this chapter.

5.7.1 Direct Print and Export From the Designer

Introduction

It is possible to provide the Designer with data for preview, so that the user sees the report as it will be when printed. Furthermore, there is the possibility of printing or exporting directly from the Designer.

For C++ there is already a fully functional sample source code available. You will find it in the "Samples" program group for "Visual C++", its title is "Designer Preview and Drilldown ".

Your development environment must comply with the following conditions, so that this feature can be supported:

- It can respond to callbacks (refer to chapter "Callbacks and Notifications")
- It can start a thread with a print procedure and supports synchronization elements such as Mutex, Critical Section or similar.

The work to be undertaken by your code includes execution of your usual real data print/ export routine, however – at least for the preview – in a separate thread. For this purpose, information about the pending task (start, abort, end and query status) is supplied via a callback. A pointer to a *scLIDesignerPrintJob* structure will be passed, specifing all neccessary information for the respective task. There are only a few changes necessary compared to a regular print/export.

Preparation

In order to enable the direct preview or export functionality within the designer, you have to set one or both of the following options:

- *LL_OPTION_DESIGNERPREVIEWPARAMETER* for preview of real data
- *LL_OPTION_DESIGNEREXPORTPARAMETER* for the export from the designer

The value that you pass via these options can be defined by yourself, for example a pointer to an internal data structure or object. You receive this back unchanged in the callback (*scLIDesignerPrintJob._nUserParam*). Important for List & Label is that it is not 0 or -1.

Via callback *LL_NTFY_DESIGNERPRINTJOB* List & Label informs you about the task that has to be performed. This callback will always be called in the context of the designer thread (this is the thread, from which *LIDefineLayout()* was called).

When you use structure members, e.g. like *_nUserParam*, please ensure that the thread has evaluated or copied them before you pass control back to List & Label, as the structure will no longer be valid then – this is true for all Callbacks!

Tasks

Now to the individual tasks, that will be put to you via the callback being indicated by various values of *scLIDesignerPrintJob._nFunction*. The symbolic constants for them all start with *LL_DESIGNERPRINTCALLBACK...*:

Start Event (..._PREVIEW_START/..._EXPORT_START)

Should you receive this Event, you will have to create a thread and pass the start parameters to it.

This thread creates a new List & Label- job, and causes the new job to basically lead to the execution of "a standard" print loop. In addition to the normal print loop you will have to perform the following modifications:

- Before print start set the option *LL_OPTIONSTR_ORIGINALPROJECTFILENAME* to the path, as delivered by the structure.
- After starting the print job, use *LlPrintSetOption(hJob ,LL_PRNOPT_LASTPAGE, _nPages)* to set the maximum number of pages to be printed, as passed by the callback structure.
- After each *LlPrint()*, check if the number of pages has exceeded this value, if so call function *LlPrintAbort()*. This optimization can and should also be used for normal print jobs. In this case you should not abort, but end printing normally using *LlPrintEnd()*.
- Indicate the thread status using the provided event *_hEvent* of the callback structure to List & Label, once at the start and once at the end of the thread. It is important to synchronize the signaling of the event in a way that makes sure that the following call to QUEST_JOBSTATE delivers the correct state RUNNING/STOPPED. As you're signaling the event from the thread, you cannot use the thread status, but will have to use a corresponding variable. This process status has to be handled individually for each thread.
- Delete the remaining project data after printing

Additionally the following points have to be considered:

Preview

- Pass the window handle you've received in the callback structure via *LlAssociatePreviewControl(hJob,hWnd,1)* to List & Label before calling *LlPrint(WithBox)Start*, so that the print job is informed where the data should be presented.
- After completing printing, that is after *LlPrintEnd()* call *LlAssociatePreviewControl(hJob,NULL,1)*, so that preview control regains control of the preview data. If a print error occurs, the last parameter has to be 0, so that the preview control is empty and not showing the last project.

Export

- Use *LlPrintWithBoxStart()*, so that a status box can be displayed. For the parent window handle use additionally the window handle passed by the callback structure.
- If the user selects a direct export from the Ribbon's menu, the _pszExportFormat member of the struct is set. In this case, call *LlPrintSetOptionString(hJob, LL_PRNOPTSTR_EXPORT, _pszExportFormat)* to preset the required export format and do not show the print options dialog (*LlPrintOptionsDialog()*) if _bWithoutDialog is TRUE.

Abort Event (..._PREVIEW_ABORT/..._EXPORT_ABORT)

If you receive this event, simply call *LlPrintAbort()* to abort the print job for the preview/export thread. The print loop of the thread ensures correct handling.

Finalize Event (..._PREVIEW_FINALIZE/..._EXPORT_FINALIZE)

Will always be called, so that you can release internal data structures.

Status Query Event (..._PREVIEW_QUEST_JOBSTATE/..._EXPORT_QUEST_JOBSTATE)

Is used to keep List & Label toolbar icons and menu options up to date. Return *LL_DESIGNERPRINTTHREAD_STATE_RUNNING* when your thread is running, otherwise return *LL_DESIGNERPRINTTHREAD_STATE_STOPPED.*

Activity

Of course you can support multiple start events. Before each start, List & Label checks if a print thread is running, and stops it if necessary with an abort event.

Designer-Thread	Print-Thread
Start-Event: • Copies the start parameter of the callback • Starts the print thread and waits on signal that it is ready (Event)	
	starts: • sets process status internally to RUNNING • indicates change of state per *SetEvent(hEvent)* to List & Label • indicate readiness
returns to List & Label	

From now on both the designer and preview/export run in parallel.

Normal designer execution.	• create new job

Abort • calls *LlPrintAbort()* for the print job and returns Status Query • returns the value of the process status Completion • calls *LlPrintAbort()* if necessary and waits for thread to end	• starts print loop with changes already mentioned above When printing is complete: • set internal process state to STOPPED • indicate state change per *SetEvent(hEvent)* to List & Label • end job • delete project file

It is advisable to use a unique structure for both output types, and then provide the address of the structure using *LL_OPTION_DESIGNERPREVIEWPARAMETER* and *LL_OPTION_DESIGNEREXPORTPARAMETER* to List & Label. This structure should contain:

- a pointer to a object that manages the data source (if necessary i.e. possible)
- a synchronization object (CRITICAL_SECTION)
- the thread handle of the thread in progress
- the job handle of the worker thread
- variables as copies of the start parameter

If your data source only allows single threaded access, you must set *LL_OPTION_DESIGNERPRINT_SINGLETHREADED* to *TRUE*. This will be used by List & Label, so that during the preview calculation no export is possible, and vice versa.

5.7.2 Drilldown Reports in Preview

Drilldown reporting means navigation in hierarchical data through different detail levels.

Initially only the top level will be printed to preview (for example "Customers"). With a click on a customer, a new report (for example "Orders") will be opened, that contains detail information for this record. In this manner, you "drill down" through different levels until you reach for example the products a customer has ordered in a specific order. One of the advantages is the performance gain by means of specialization. Drilldown is available in preview. Drilldown can be defined for table rows and table fields.

Using drilldown requires that your development system can handle callbacks or window notifications (see chapter "Callbacks and Notifications")

The .NET and VCL components in databound mode support drilldown automatically if a data source is used that supports hierarchies and can be reset. Most DataProviders comply with this requirement. If using the component in this manner, you can skip this chapter.

For C++ there is already a fully functional sample source code available. You'll find it in the "Samples" program group for "Visual C++", its title is "Designer Preview and Drilldown".

For other development systems, it is suggested to implement drilldown with different threads, so tasks can be done in the background. In this case, you will need to be able to start a thread with the printing procedure and you will need synchronisation elements like mutex or critical section.

Your task is to initiate a real data print job with a corresponding filtered data source. For this purpose, information about the task (start and end of a drilldown report) is available in the callback. Only minor changes to the normal print job routines are necessary.

Preparations

To enable drilldown in List & Label set the option *LL_OPTION_DRILLDOWNPARAMETER* to a value unequal to 0.

Please note that this option has to be set for each LL-job that should support drilldown:

```
// activate Drilldown for current LL-Job
::LlSetOption(hJob, LL_OPTION_DRILLDOWNPARAMETER,
              (LPARAM)&oMyDrillDownParameters);
```

To deactivate drilldown for this LL-job set the option to NULL:

```
// deactivate Drilldown for current LL-Job
::LlSetOption(hJob, LL_OPTION_DRILLDOWNPARAMETER, NULL);
```

The parameter passed with this option can be used freely, for example as a pointer to an internal data structure or objecets. This parameter will be passed unchanged in the callback for your use (*scLIDrillDownJob._nUserParam)*. Please make sure the parameter is not 0 or NULL unless you want to deactivate drilldown.

Via the callback *LL_NTFY_VIEWERDRILLDOWN* (for further description, please see chapter "Callbacks and Notifications") List & Label informs about the current task. This callback will always be called in the context of the preview thread, regardless if initiated from designer or preview print.

When you use structure members, e.g. like *_nUserParam*, please ensure that the thread has evaluated or copied them before you pass control back to List & Label, as the structure will no longer be valid then – this is true for all Callbacks!

Tasks

Now to the individual tasks, that will be put to you via the callback being indicated by various values of *scLlDrillDownJob._nFunction*:

5.7.2.1.1 Start Event (LL_DRILLDOWN_START)

If this event is fired, you can create a thread and pass on the drilldown parameters. If your development systems does not support threads, you should use the main thread for the print job. In this case your program will not be usable for this period.

The return value of the callback (resp. the *_lReply* member of the *scLlCallback*-structure) should be set to a unique number, so you can assign the drilldown reports to the corresponding thread.

Example:

```
…
LRESULT        lResult = 0;
case LL_DRILLDOWN_START:
{
       scLlDrillDownJob* pDDJob = (scLlDrillDownJob*)lParam;
       … StartSubreport(pDDJob); …
       // generate new Drilldown-JobID
       lResult = ++m_nUniqueDrillDownJobID;
}
case LL_DRILLDOWN_FINALIZE:
{
       scLlDrillDownJob* pDDJob = (scLlDrillDownJob*)lParam;
       if (pDDJob->_nID == 0)
       {
              // clean up
       }
       else
       {
              // clean up the corresponding job
       }
}
…
return (lResult);
}
…
```

After copying the parameters, this thread creates a new List & Label job that uses the regular print loop. The following differences have to be made before calling *LlPrintStart()*:

- set the option *LL_OPTIONSTR_PREVIEWFILENAME* to the path, that has been passed in the structure with *_pszPreviewFileName*

Example:

```
// set preview filename
::LlSetOptionString(pMyDrillDownParameters->m_hLlJob,
               LL_OPTIONSTR_PREVIEWFILENAME,
               pMyDrillDownParameters ->m_sPreviewFileName);
```

- Pass on the *_hAttachInfo* to List & Label that has been passed with the callback structure, so the print job is informed whrere the data should be displayed.

Example:

```
// attach viewer
::LlAssociatePreviewControl(pMyDrillDownParameters->m_hLlJob,
                (HWND)pMyDrillDownParameters->_hAttachInfo,
               LL_ASSOCIATEPREVIEWCONTROLFLAG_DELETE_ON_CLOSE |
               LL_ASSOCIATEPREVIEWCONTROLFLAG_HANDLE_IS_ATTACHINFO);
```

5.7.2.1.2 Finalize Event (LL_DRILLDOWN_FINALIZE)

This event will be called for drilldown jobs, that have been canceled, so you can release internal data structures. Additionally it is suggested, that active print jobs should be aborted by calling *LlPrintAbort().*

If the *_nID* member of the *scLlDrillDownJob* structure that has been passed by List & Label is 0 all active drilldown jobs can be ended and released. This happens if ending the preview.

5.7.2.1.3 Preparing the Data Source

To provide the correct data for the drilldown report minor changes to the printing loop are neccessary.

Relation(s)

Drilldown can only be used when relations are declared. For drilldown reports use the function *LlDbAddTableRelationEx().* This function has 2 additional parameters: *pszKeyField* and *pszParentKeyField* for the key field of the child table and the key field of the parent table, so a unique assignment can be made.

Further information can be found in the description of the function *LlDbAddTableRelationEx()*.

Please note, that the key fields must contain the table name as a prefix, for example "Customers.CustomerID".

Example:

Declare the relation between 'Customers' and 'Orders' table for drilldown using the northwind sample.

```
// add relation
…
CString sParentField = pMyDrillDownParameters->_pszSubreportTableID +
               _T(".")+pMyDrillDownParameters->_pszKeyField;
//Orders.CustomerID
CString sChildField = pMyDrillDownParameters->_pszTableID + _T(".") +
               pMyDrillDownParameters->_pszSubreportKeyField;
//Customers.OrderID
::LlDbAddTableRelationEx(hJob,
               pMyDrillDownParameters->_pszSubreportTableID,   // "Orders"
               pMyDrillDownParameters->_pszTableID,    // "Customers"
               pMyDrillDownParameters->_pszRelationID, _T(""),
               sParentField, sChildField);
…
```

Datasource

For each drilldown report the datasource should be filtered differently, because only the data related to the parent record that has been clicked is needed.

For example you want to create a drilldown structure from "Customers" to "Orders". In this case the parent table should show all customers. A click on one customer should show only the corresponding orders related to this special customer. Therefor the datasource must only contain the orders from this customer. All necessary information for filtering the child table can be found in the structure 'scLIDrillDownJob'.

5.7.3 Supporting the Report Parameter Pane in Preview

Out of the box, report parameter printing is already supported automatically. However, if you want to support a re-rendering from the preview window, you need to signal your support to List & Label by setting some additional options. For convenience, this feature uses the same callback as drilldown printing, so in most cases you can simply reuse your existing code and just make sure it works even if no drilldown filter is set in the passed structure (i.e. all table IDs and key field values are empty).

Preparations

To enable report parameter printing in List & Label set the option *LL_OPTION_REPORT_PARAMETERS_REALDATAJOBPARAMETER* to a value unequal to 0.

Please note that this option has to be set for each LL-job that should support report parameter printing:

```
// activate report parameter pane for current LL-Job
::LlSetOption(hJob, LL_OPTION_REPORT_PARAMETERS_REALDATAJOBPARAMETER,
              (LPARAM)&oMyReportParameters);
```

To deactivate report parameter printing for this LL-job set the option to NULL:

```
// deactivate report parameter pane for current LL-Job
::LlSetOption(hJob, LL_OPTION_REPORT_PARAMETERS_REALDATAJOBPARAMETER, NULL);
```

The parameter passed with this option can be used freely, for example as a pointer to an internal data structure or objects. This parameter will be passed unchanged in the callback for your use (*scLlDrillDownJob._nUserParam)*. Please make sure the parameter is not 0 or NULL unless you want to deactivate report parameter printing.

Via the callback *LL_NTFY_VIEWERDRILLDOWN* (for further description, please see chapter "Drilldown Reports in Preview") List & Label informs about the current task. This callback will always be called in the context of the preview thread, regardless if initiated from designer or preview print.

When you use structure members, e.g. like *_nUserParam*, please ensure that the thread has evaluated or copied them before you pass control back to List & Label, as the structure will no longer be valid then – this is true for all Callbacks!

5.7.4 Supporting Expandable Regions in Preview

If this feature is supported, elements in the report container can be expanded and collapsed dynamically in the preview window. For convenience, this feature uses the same callback as drilldown printing, so in most cases you can simply reuse your existing code and just make sure it works even if no drilldown filter is set in the passed structure (i.e. all table IDs and key field values are empty).

Preparations

To enable expandable regions in List & Label set the option *LL_OPTION_EXPANDABLE_REGIONS_REALDATAJOBPARAMETER* to a value unequal to 0.

Please note that this option has to be set for each LL-job that should support expandable regions:

```
// activate expandable regions for current LL-Job
::LlSetOption(hJob, LL_OPTION_EXPANDABLE_REGIONS_REALDATAJOBPARAMETER,
              (LPARAM)&oMyExpandableRegions);
```

To deactivate expandable regions for this LL-job set the option to NULL:

```
// deactivate expandable regions for current LL-Job
::LlSetOption(hJob, LL_OPTION_EXPANDABLE_REGIONS_REALDATAJOBPARAMETER,
NULL);
```

The parameter passed with this option can be used freely, for example as a pointer to an internal data structure or objecets. This parameter will be passed unchanged in the callback for your use (*scLlDrillDownJob._nUserParam)*. Please make sure the parameter is not 0 or NULL unless you want to deactivate expandable regions.

Via the callback *LL_NTFY_VIEWERDRILLDOWN* (for further description, please see chapter "Drilldown Reports in Preview") List & Label informs about the current task. This callback will always be called in the context of the preview thread, regardless if initiated from designer or preview print.

When you use structure members, e.g. like *_nUserParam*, please ensure that the thread has evaluated or copied them before you pass control back to List & Label, as the structure will no longer be valid then – this is true for all Callbacks!

5.7.5 Supporting Interactive Sorting in Preview

If this feature is supported, report container table header fields can be used to toggle between different sortings in the preview window. For convenience, this feature uses the same callback as drilldown printing, so in most cases you can simply reuse your existing code and just make sure it works even if no drilldown filter is set in the passed structure (i.e. all table IDs and key field values are empty).

Preparations

To enable interactive sortings in List & Label set the option *LL_OPTION_INTERACTIVESORTING_REALDATAJOBPARAMETER* to a value unequal to 0.

Please note that this option has to be set for each LL-job that should support interactive sortings:

```
// activate interactive sortings for current LL-Job
::LlSetOption(hJob, LL_OPTION_INTERACTIVESORTING_REALDATAJOBPARAMETER,
              (LPARAM)&oMyInteractiveSortings);
```

To deactivate interactive sortings for this LL-job set the option to NULL:

```
// deactivate interactive sortings for current LL-Job
::LlSetOption(hJob, LL_OPTION_INTERACTIVESORTING_REALDATAJOBPARAMETER,
NULL);
```

The parameter passed with this option can be used freely, for example as a pointer to an internal data structure or objecets. This parameter will be passed unchanged in the callback for your use (*scLIDrillDownJob._nUserParam)*. Please make sure the parameter is not 0 or NULL unless you want to deactivate expandable regions.

Via the callback *LL_NTFY_VIEWERDRILLDOWN* (for further description, please see chapter "Drilldown Reports in Preview") List & Label informs about the current task. This callback will always be called in the context of the preview thread, regardless if initiated from designer or preview print.

When you use structure members, e.g. like *_nUserParam*, please ensure that the thread has evaluated or copied them before you pass control back to List & Label, as the structure will no longer be valid then – this is true for all Callbacks!

5.7.6 Handling Chart and Crosstab Objects

The easiest way to work with charts and crosstabs is to insert them into the report container. See chapter "Printing Relational Data" for a detailed explanation.

However, for label and card projects it might be interesting to access these objects separately.

In the following chapter whenever "chart" is mentioned "crosstab" also applies.

Besides working with the report container, there are two different modes when handling chart objects. Choose the one you require via the option *LL_OPTION_USECHARTFIELDS*. In principle, printing charts is similar to printing tables, i.e. you first declare a data record you wish to pass to the chart object and pass this record afterwards.

Standard Mode (Default)

This mode can only be used with list projects and does not require any changes to existing projects. The chart objects are fed with the same data as the table objects, an *LIPrintFields()* sends data to both chart objects and table objects. However, the charts only accumulate the data and are not printed right away. This mode is included for compatibility reasons and can be used to easily fill charts that are linked to table objects.

Enhanced Mode

This mode is activated by setting the option *LL_OPTION_USECHARTFIELDS* to TRUE. In this case, besides variables and fields you have special chart fields available. These can be declared similarly to normal fields via API calls. This mode offers more flexibility than the standard mode; your users may

- use chart objects wherever they like (no printing order has to be obeyed)

- use chart objects in label / card projects.

LlPrintFields() in this mode does not have any influence on the charts, the analogous command in the enhanced mode is *LlPrintDeclareChartRow().* This API call passes the data currently defined to the chart objects. Which chart objects are addressed can be determined by the parameter:

Value	Meaning
LL_DECLARECHARTROW_FOR_OBJECTS	The data is passed to all chart objects not contained in table columns.
LL_DECLARECHARTROW_FOR_TABLECOLUMNS	The data is passed to all chart objects in table columns.

For charts within a label project, the following pseudo code would apply:

```
<print start>
       (LlPrintStart,
        LlPrintWithBoxStart)

<while
       - no error and not finished>
{
       <define variables>
       <while
       - no error or
       - not finished (ex. i = 1..12)>
       {
              <define chart fields (ex. Month = MonthName[i])>
              <send data to chart controls>
              (LlPrintDeclareChartRow(LL_DECLARECHARTROW_FOR_OBJECTS))
              }

       <print objects>
              (LlPrint)
       <no warning, no abortion: next record>
       }
<done>
       (LlPrintEnd)
```

Of course, all chart fields used must also be declared before calling the Designer, in order to enable your users to use them at all.

5.8 Using the DOM-API (Professional/Enterprise Edition Only)

This chapter is only required if you're not working with one of the components .NET/VCL, where a type safe object model for accessing the DOM functionality is available. If you are using one of these components, you may skip this chapter and turn to one of the DOM samples for a quick start.

In order to create project files dynamically for the runtime or to edit existing project files by code, you can use the List & Label DOM functions.

5.8.1 Basic Principles

Each "object" within a project file has its own handle ("DOM handle"). The functions of the DOM-API use this handle to uniquely identify objects. An "object" in this sense is any designer object, but also other elements such as auxiliary lines, project parameters etc. The DOM viewer included in the scope of supply enables a quick overview of all objects, their value and other properties. In addition, properties / values can be changed with the viewer, and saved in the project. The clipboard function enables any object or property to be copied to the clipboard for further use.

The functions relevant for the DOM-API are divided into 2 groups: first, project files can be loaded, created and saved. The functions *LlProjectOpen(), LlProjectClose()* and *LlProjectSave()* are available for this purpose. The function *LlDomGetProject()* (called immediately after *LlProjectOpen()*) returns the DOM handle for the project object. This then provides the basis for using the other functions.

DOM Functions

LlDomGetObject

With this function, important subobjects can be obtained from the project object. In order to obtain the object list, for example,

```
LlProjectOpen(hJob, LL_PROJECT_LIST, "c:\\filename.lst",
      LL_PRJOPEN_AM_READONLY);
HLLDOMOBJ hProj;
LlDomGetProject(hJob, &hProj);
HLLDOMOBJ hObjList;
INT nRet = LlDomGetObject(hProj, "Objects", &hObjList);
```

can be used. The other available objects correspond to the entries in the tree structure in the DOM viewer: "Layout", "ProjectParameters", "Settings", "SumVars" and "UserVars". A description of the individual objects with most properties can be found in the reference chapter; the emphasis here is on the principle of working with the DOM functions.

LlDomGetSubobjectCount

Serves to query the number of subobjects in the specified list. To query the number of objects in the project, for instance, use

```
INT nObjCount;
INT nRet = LlDomGetSubobjectCount(hObjList, &nObjCount);
```

LlDomGetSubobject

Returns the DOM handle of the specified subobject. In addition to the DOM handle for the list, parameters are the index (0-based) and a pointer for return of the handle. The code for a DOM handle to the first object in the projectfile is

```
HLLDOMOBJ hObj;
INT nRet = LlDomGetSubobject(hObjList, 0, &hObj);
```

LlDomCreateSubobject

Creates a new subobject in the specified list. Parameters are the list handle, the insertion position, the desired type and a handle pointer for the new object. In order to insert a new text object at the beginning of the object list, use

```
HLLDOMOBJ hObj;
INT nRet = LlDomCreateSubobject(hObjList, 0, _T("Text"), &hObj);
```

You can create the following objects within the object list with the help of these functions, for example:

Object type	Required third parameter
Line	"Line"
Rectangle	"Rectangle"
Ellipse	"Ellipse"
Drawing	"Drawing"
Text	"Text"
Template	"Template"

Barcode	"Barcode"
RTF	"RTFText"
HTML	"LLX:LLHTMLObject"
Report container (may contain tables, charts and crosstabs)	"ReportContainer"
Gauge	"Gauge"
PDF	"PDF"

Further possible values for other lists (e.g. field list within a table) can be found in the DOM Viewer's online help.

LlDomDeleteSubobject

Deletes the specified subobject. In order to delete the first object in the object list, for example, use the code

```
INT nRet = LlDomDeleteSubobject(hObjList, 0);
```

LlDomSetProperty

Allows you to set a property for the specified object. In order to allow the pagebreak for a text object, for example, you need

```
INT nRet = LlDomSetProperty(hObj, _T("AllowPageWrap"), _T("True"));
```

The transfer parameter for the value must be a valid List & Label formula. A special feature results for properties that contain character strings (e.g. the content of a text paragraph): character strings must be set in quotation marks within the Designer, to enable their use as a valid formula. Therefore, in order to transfer the fixed text "combit", the parameter "'combit'" must be used. This also applies for fixed font names, for example; once again, "'Verdana'" must be transferred, for example, not "Verdana".

Example code: LlDomSetProperty(hObj, _T("Contents"), _T("'") + sProjectTitle + _T("'"));

In order to set the values of nested properties, such as the color of a filling, the property name "<Parent property>.<Child property>" can be used, so for example

```
INT nRet = LlDomSetProperty(hObj, _T("Filling.Color"),
_T("LL.Color.Black"));
```

LlDomGetProperty

Reads out the value of a property. It is advisable to determine the necessary buffer length first of all by transferring a NULL buffer, as usual, and then to allocate an adequately large buffer:

```
INT nBufSize = LlDomGetProperty(hObj, _T("AllowPageWrap"), NULL, 0);
TCHAR* pszBuffer = new TCHAR[nBufSize];
INT nRet = LlDomGetProperty(hObj, _T("AllowPageWrap"), pszBuffer, nBufSize);
…
delete[] pszBuffer;
```

For simplification, objects (but not lists!) can also be "tunneled through" using the full stop as hierarchy separator, as for example:

...

```
//US: Get the page coordinates for the first page
LlDomGetProperty(hRegion, _T("Paper.Extent.Horizontal"),
       pszContainerPositionWidth, nBufSize);
```

Units

Many properties contain information on sizes, widths etc. These are - if transferred as fixed numbers - interpreted and returned as SCM units (1/1000 mm) and are therefore independent of the selected unit system. In order to place an object in a (fixed) position 5 mm from the left margin, you would use

```
INT nRet = LlDomSetProperty(hObj, _T("Position.Left"), _T("5000"));
```

If the property is to contain a formula rather than a fixed value, the function UnitFromSCM must be used, in order to be independent of the units. An inside margin with an indent of 10 mm on odd and 5 mm on even pages would be produced with

```
INT nRet = LlDomSetProperty(hObj, _T("Position.Left"), T("Cond(Odd(Page()),
       UnitFromSCM(10000), UnitFromSCM(5000))"));
```

5.8.2 Examples

Creating a Text Object

The following code creates a new project, inserts a text object inside which is a new paragraph with the content "DOM", and saves the project:

```
HLLJOB hJob = LlJobOpen(-1);

// Create new project
LlProjectOpen(hJob,LL_PROJECT_LIST,"c:\\simple.lst",
       LL_PRJOPEN_CD_CREATE_ALWAYS | LL_PRJOPEN_AM_READWRITE);

HLLDOMOBJ hProj;
LlDomGetProject(hJob, &hProj);

// Get object list
HLLDOMOBJ hObjList;
LlDomGetObject(hProj, "Objects", &hObjList);

// Create text object
HLLDOMOBJ hObj;
LlDomCreateSubobject(hObjList, 0, _T("Text"), &hObj);
LlDomSetProperty(hObj, _T("Name"), _T("My new Textobject"));

// Get paragraph list
HLLDOMOBJ hObjParagraphList;
LlDomGetObject(hObj, _T("Paragraphs"), &hObjParagraphList);

// Create new paragraph and create contents
HLLDOMOBJ hObjParagraph;
LlDomCreateSubobject(hObjParagraphList, 0, _T("Paragraph"), &hObjParagraph);
LlDomSetProperty(hObjParagraph, _T("Contents"), _T("'DOM'"));

// Save project
LlProjectSave(hJob, NULL);
LlProjectClose(hJob);

LlJobClose(hJob);
```

Creating a Table

This example shows the creation of a table object inside a report container and creates a new dataline and three columns inside it.

Please note that, even if you do not use the APIs to control the report container, you must create a report container with exactly one table.

```
HLLJOB hJob = LlJobOpen(-1);
// Create new project
LlProjectOpen(hJob, LL_PROJECT_LIST, "c:\\simple.lst",
       LL_PRJOPEN_CD_CREATE_ALWAYS | LL_PRJOPEN_AM_READWRITE);

HLLDOMOBJ hProj;
LlDomGetProject(hJob, &hProj);

// Get object list
HLLDOMOBJ hObjList;
LlDomGetObject(hProj, "Objects", &hObjList);

// Create report container and set properties
HLLDOMOBJ hObjReportContainer;
LlDomCreateSubobject(hObjList, 0,
_T("ReportContainer"),&hObjReportContainer);
LlDomSetProperty(hObjReportContainer,_T("Position.Left"), _T("27000"));
LlDomSetProperty(hObjReportContainer,_T("Position.Top"), _T("103500"));
LlDomSetProperty(hObjReportContainer,_T("Position.Width"), _T("153400"));
LlDomSetProperty(hObjReportContainer,_T("Position.Height"), _T("159500"));

// Get subobject list and create table inside it
HLLDOMOBJ hObjSubItems;
LlDomGetObject(hObjReportContainer, _T("SubItems"), & hObjSubItems);
HLLDOMOBJ hObjTable;
LlDomCreateSubobject(hObjSubItems, 0, _T("Table"), &hObjTable);

// Get line list
HLLDOMOBJ hObjTableLines;
LlDomGetObject(hObjTable , _T("Lines"), &hObjTableLines);

// Get data line list
HLLDOMOBJ hObjTableData;
LlDomGetObject(hObjTableLines , _T("Data"), &hObjTableData);

// Create new line definition
HLLDOMOBJ hObjTableLine;
LlDomCreateSubobject(hObjTableData, 0, _T("Line"), &hObjTableLine);
LlDomSetProperty(hObjTableLine,_T("Name"), _T("My new table line"));

// Get header list
HLLDOMOBJ hObjTableHeader;
LlDomGetObject(hObjTableLines , _T("Header"), &hObjTableHeader);

// Create new line definition
HLLDOMOBJ hObjTableHeaderLine;
LlDomCreateSubobject(hObjTableHeader, 0, _T("Line"), &hObjTableHeaderLine);

// Get field list for headers
HLLDOMOBJ hObjTableHeaderFields;
```

```
LlDomGetObject(hObjTableHeaderLine , _T("Fields"), &hObjTableHeaderFields);

// Get field list for data lines
HLLDOMOBJ hObjTableDataFields;
LlDomGetObject(hObjTableLine , _T("Fields"), &hObjTableDataFields);

TCHAR aczVarName[1024];
int nItemCount = 3;
for (int i=0; i < nItemCount; i++)
{
       sprintf(aczVarName, "'Var%d'", i);

       // Create new field in header and set properties
       HLLDOMOBJ hObjHeaderField;
       LlDomCreateSubobject(hObjTableHeaderFields, 0, _T("Text"),
              &hObjHeaderField);
       LlDomSetProperty(hObjHeaderField, _T("Contents"), aczVarName);
       LlDomSetProperty(hObjHeaderField,_T("Filling.Style"), _T("1"));
       LlDomSetProperty(hObjHeaderField,_T("Filling.Color"),
              _T( "RGB(204,204,255)"));
       LlDomSetProperty(hObjHeaderField,_T("Font.Bold"), _T("True"));
       LlDomSetProperty(hObjHeaderField,_T("Width"), _T("50000"));

       sprintf(aczVarName, "Var%d", i);

       // Create new field in data line and set properties
       HLLDOMOBJ hObjDataField;
       LlDomCreateSubobject(hObjTableDataFields, 0, _T("Text"),
              &hObjDataField);

       LlDomSetProperty(hObjDataField,_T("Contents"), aczVarName);
       LlDomSetProperty(hObjDataField,_T("Width"), _T("50000"));
}

// Save project
LlProjectSave(hJob, NULL);
LlProjectClose(hJob);
LlJobClose(hJob);
```

Setting the Project Parameters

The following code sets project parameters in an existing List & Label project for fax and sending mail:

```
HLLJOB hJob = LlJobOpen(-1);

LlProjectOpen(hJob, LL_PROJECT_LIST, "c:\\simple.lst",
       LL_PRJOPEN_CD_OPEN_EXISTING | LL_PRJOPEN_AM_READWRITE);

HLLDOMOBJ hProj;
```

```
LlDomGetProject(hJob, &hProj);

// Fax parameter:
LlDomSetProperty(hProj, _T("ProjectParameters.LL.FAX.RecipName.Contents"),
               _T("'sunshine agency'"));
LlDomSetProperty(hProj, _T("ProjectParameters.LL.FAX.RecipNumber.Contents"),
_T("'555-555 555'"));
LlDomSetProperty(hProj,
       _       T("ProjectParameters.LL.FAX.SenderCompany.Contents"),
               _T("'combit'"));
LlDomSetProperty(hProj,_T("ProjectParameters.LL.FAX.SenderName.Contents"),
               _T("John Q. Public'"));

// Mail parameter:
LlDomSetProperty(hProj, _T("ProjectParameters.LL.MAIL.Subject.Contents"),
               _T("'Your request'"));
LlDomSetProperty(hProj, _T("ProjectParameters.LL.MAIL.From.Contents"),
               _T("'info@combit.net'"));
LlDomSetProperty(hProj, _T("ProjectParameters.LL.MAIL.To.Contents"),
               _T("'info@sunshine-agency.net'"));

// Save project
LlProjectSave(hJob, NULL);
LlProjectClose(hJob);
LlJobClose(hJob);
```

6. API Reference

6.1 Function Reference

LlAssociatePreviewControl

Syntax:

```
INT LlAssociatePreviewControl(HLLJOB hJob, HWND hWndControl,
   UINT nFlags);
```

Task:

Associates a LL job to a preview control.

Parameters:

hJob: List & Label job handle

hWnd: window handle

nFlags:

Value	Meaning
LL_ASSOCIATEPREVIEW-CONTROLFLAG_DELETE_-ON_CLOSE	Automatically delete preview file when preview is closed
*LL*__ASSOCIATEPREVIEW-CONTROLFLAG_HANDLE_-IS_ATTACHINFO	Informs the API, that the passed window handle is a pointer to a structure containing drilldown information
LL_ASSOCIATEPREVIEW-CONTROLFLAG_PRV_-REPLACE	If LL_ASSOCIATEPREVIEWCONTROLFLAG_-HANDLE_IS_ATTACHINFO is not set: the current preview will be replaced by this preview
LL_ASSOCIATEPREVIEW-CONTROLFLAG_PRV_-ADD_TO_CONTROL_STACK	If LL_ASSOCIATEPREVIEWCONTROLFLAG_-HANDLE_IS_ATTACHINFO is not set: this preview is added to the current preview tab
LL_ASSOCIATEPREVIEW-CONTROLFLAG_PRV_-ADD_TO_CONTROL_IN_-TAB	If LL_ASSOCIATEPREVIEWCONTROLFLAG_-HANDLE_IS_ATTACHINFO is not set: this preview is added to the current preview window in a new tab

If necessary combined with 'or'.

Return Value:

Error code

Hints:

See chapter "Direct Print and Export From the Designer".

Example:

See chapter "Direct Print and Export From the Designer".

LlCreateSketch

Syntax:

```
INT LlCreateSketch (HLLJOB hJob, UINT nObjType, LPCTSTR lpszObjName);
```

Task:

Creates a sketch that can later be displayed in the file selection dialogs. The color depth can be set using *LL_OPTION_SKETCHCOLORDEPTH*

Parameter:

hJob: List & Label job handle

nObjType: Project type

Value	Meaning
LL_PROJECT_LABEL	for labels
LL_PROJECT_CARD	for cards
LL_PROJECT_LIST	for lists

lpszObjName: Pointer to project's file name (with path) and file extension

Return Value:

Error code

Hints:

This API function can be used to create the sketches on-the-fly automatically, so that you do not need to include the sketch files in your setup.

See also:

LlSelectFileDlgTitleEx

LlDbAddTable

Syntax:

```
INT LlDbAddTable(HLLJOB hJob, LPCTSTR pszTableID,
   LPCTSTR pszDisplayName);
```

Task:

Adds a table or schema for designing and printing. This table is available in the Designer and List & Label can request it at print time.

Parameter:

hJob: List & Label job handle

pszTableID: Unique name of the table. It is returned by the *LlPrintDbGetCurrentTable()* function at print time. If you pass an empty string or NULL, the table buffer will be deleted.

pszDisplayName: Name of the table as displayed in the Designer. If no name is given, the display name and the unique name are identical.

Return Value:

Error code

Hints:

If a table name contains a "." a schema will be used.

See the hints in chapter "Printing Relational Data".

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDbAddTable(hJob, "", NULL);
LlDbAddTable(hJob, "Orders", NULL);
LlDbAddTable(hJob, "OrderDetails", NULL);
LlDbAddTable(hJob, " HumanResources.Employee", NULL); // schema info
<... etc ...>
LlJobClose(hJob);
```

See also:

LlDbAddTableSortOrder, LlDbAddTableRelation, LlPrintDbGetCurrentTable, LlPrintDbGetCurrentTableSortOrder, LlPrintDbGetCurrentTableRelation

LlDbAddTableEx

Syntax:

```
INT LlDbAddTableEx(HLLJOB hJob, LPCTSTR pszTableID,
   LPCTSTR pszDisplayName, UINT nOptions);
```

Task:

Adds a table or schema for designing and printing and supports additional options. This table is available in the Designer and List & Label can request it at print time.

Parameter:

hJob: List & Label job handle

pszTableID: Unique name of the table. It is returned by the *LlPrintDbGetCurrentTable()* function at print time. If you pass an empty string or NULL, the table buffer will be deleted.

pszDisplayName: Name of the table as displayed in the Designer. If no name is given, the display name and the unique name are identical.

nOptions: a combination of one of the following flags:

Value	Meaning
LL_ADDTABLEOPT_-SUPPORTSSTACKEDSORTORDERS	Support stacked sort orders in the designer. If the user chooses multiple stacked sortings, these are returned tab separated in *LlPrintDbGetCurrentTableSortOrder()*
LL_ADDTABLEOPT_-SUPPORTSADVANCEDFILTERING	Support the translation of filter expressions to native syntax. See the documentation for the *LL_QUERY_EXPR2HOSTEXPRESSION* callback and *LlPrintDbGetCurrentTableFilter()*

Return Value:

Error code

Hints:

If a table name contains a "." a schema will be used.

See the hints in chapter "Printing Relational Data".

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDbAddTable(hJob, "", NULL);
LlDbAddTable(hJob, "Orders", NULL);
LlDbAddTable(hJob, "OrderDetails", NULL);
LlDbAddTable(hJob, " HumanResources.Employee", NULL); // schema info
<... etc ...>
LlJobClose(hJob);
```

See also:

LlDbAddTableSortOrder, LlDbAddTableRelation, LlPrintDbGetCurrentTable, LlPrintDbGetCurrentTableSortOrder, LlPrintDbGetCurrentTableRelation

LlDbAddTableRelation

Syntax:

```
INT LlDbAddTableRelation(HLLJOB hJob, LPCTSTR pszTableID,
   LPCTSTR pszParentTableID, LPCTSTR pszRelationID,
   LPCTSTR pszRelationDisplayName);
```

Task:

This method can be used to define relations between the tables added via *LlDbAddTable()*. List & Label does not directly distinguish between different relation types. You simply pass a relation and its ID, and you can query the current relation later at print time using *LlPrintDbGetCurrentTableRelation()*.

Parameter:

hJob: List & Label job handle

pszTableID: ID of the child table. Must be identical to the ID passed in *LlDbAddTable()*.

pszParentTableID: ID of the parent table. Must be identical to the ID passed in *LlDbAddTable()*.

pszRelationID: Unique ID of the table relation. It is returned by *LlPrintDbGetCurrentTableRelation()* at print time. Must be unique within a print.

pszRelationDisplayName: Name of the table relation as displayed in the Designer and it is not saved to the project file. If no name is given, the display name and the unique name are identical.

Return Value:

Error code

Hints:

See the hints in chapter "Printing Relational Data". Before using the call, the parent and child table must be passed with *LlDbAddTable()*.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDbAddTable(hJob, "Orders", NULL);
LlDbAddTable(hJob, "OrderDetails", NULL);
LlDbAddTableRelation(hJob, "OrderDetails", "Orders",
   "Orders2OrderDetails", NULL);
<... etc ...>
LlJobClose(hJob);
```

See also:

LlDbAddTable, LlDbAddTableSortOrder, LlPrintDbGetCurrentTable, LlPrintDbGet-CurrentTableSortOrder, LlPrintDbGetCurrentTableRelation

LlDbAddTableRelationEx

Syntax:

```
INT LlDbAddTableRelationEx(HLLJOB hJob, LPCTSTR pszTableID,
   LPCTSTR pszParentTableID, LPCTSTR pszRelationID,
```

```
LPCTSTR pszRelationDisplayName, LPCTSTR pszKeyField,
LPCTSTR pszParentKeyField);
```

Task:

This method can be used to define a relation between two tables added via *LlDbAddTable()*, especially for drilldown support. List & Label does not directly distinguish between different relation types. You simply pass a relation and its ID, and you can query the current relation later at print time using *LlPrintDbGetCurrentTableRelation()*.

Parameter:

hJob: List & Label job handle

pszTableID: ID of the child table. Must be identical to the ID passed in *LlDbAddTable()*.

pszParentTableID: ID of the parent table. Must be identical to the ID passed in *LlDbAddTable()*.

pszRelationID: ID of the table relation. It is returned by *LlPrintDbGetCurrentTableRelation()* at print time. Must be unique within a print.

pszRelationDisplayName: Name of the table relation as displayed in the Designer and it is not saved to the project file. If no name is given, the display name and the unique name are identical.

pszKeyField: Key field of the child table, multiple key fields can be added as tab separated list

pszParentKeyField: Key field of the parent table, multiple key fields can be added as tab separated list

Return Value:

Error code

Hints:

See the hints in chapter "Printing Relational Data". Before using the call, the parent and child table must be passed with *LlDbAddTable()*.

Example:

See chapter "Direct Print and Export From the Designer".

See also:

LlDbAddTable, LlDbAddTableSortOrder, LlPrintDbGetCurrentTable, LlPrint-DbGetCurrentTableSortOrder, LlPrintDbGetCurrentTableRelation

LlDbAddTableSortOrder

Syntax:

```
INT LlDbAddTableSortOrder(HLLJOB hJob, LPCTSTR pszTableID,
   LPCTSTR pszSortOrderID, LPCTSTR pszSortOrderDisplayName);
```

Task:

Defines available sort orders for the tables added via *LlDbAddTable()*. Each sorting has its unique ID that can be queried using *LlPrintDbGetCurrentTableSortOrder()* at print time.

Parameter:

hJob: List & Label job handle

pszTableID: Table ID to which this sort order applies. Must be identical to the ID passed in *LlDbAddTable().*

pszSortOrderID: Unique ID of the table sort order. It is returned by *LlPrintDbGetCurrentTableSortOrder()* at print time.

pszSortOrderDisplayName: Name of the table sort order as displayed in the Designer. If no name is given, the display name and the unique name are identical.

Return Value:

Error code

Hints:

See the hints in chapter "Printing Relational Data". Before using the call, the table must be passed with *LlDbAddTable()*.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDbAddTable(hJob, "Orders", NULL);
LlDbAddTableSortOrder(hJob, "Orders", "Name ASC", "Name [+]");
<... etc ...>
LlJobClose(hJob);
```

See also:

LlDbAddTable, LlDbAddTableRelation, LlPrintDbGetCurrentTable, LlPrint-DbGetCurrentTableSortOrder, LlPrintDbGetCurrentTableRelation

LlDbAddTableSortOrderEx

Syntax:

```
INT LlDbAddTableSortOrderEx(HLLJOB hJob, LPCTSTR pszTableID,
   LPCTSTR pszSortOrderID, LPCTSTR pszSortOrderDisplayName
   LPCTSTR pszField);
```

Task:

Defines available sort orders for the tables added via *LlDbAddTable()*. Each sorting has its unique ID that can be queried using *LlPrintDbGetCurrentTableSortOrder()* at print time. Additionally the fields that are relevant for the sorting can be passed separated by tabs for further use in the function *LlGetUsedIdentifiers().*

Parameter:

hJob: List & Label job handle

pszTableID: Table ID to which this sort order applies. Must be identical to the ID passed in *LlDbAddTable()*.

pszSortOrderID: Unique ID of the table sort order. It is returned by *LlPrintDbGetCurrentTableSortOrder()* at print time.

pszSortOrderDisplayName: Name of the table sort order as displayed in the Designer. If no name is given, the display name and the unique name are identical.

pszField: List of fields that are relevant for the sorting (separated by tabs) if they should be regarded in the function *LlGetUsedIdentifiers()*.

Return Value:

Error code

Hints:

See the hints in chapter "Printing Relational Data". Before using the call, the table must be passed with *LlDbAddTable()*.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDbAddTable(hJob, "Orders", NULL);
LlDbAddTableSortOrderEx(hJob, "Orders", "Name ASC", "Name [+]",
   "Orders.Name");
<... etc ...>
LlJobClose(hJob);
```

See also:

LlDbAddTableSortOrder, LlDbAddTable, LlDbAddTableRelation, LlPrintDbGetCurrentTable, LlPrintDbGetCurrentTableSortOrder, LlPrintDbGetCurrentTableRelation

LlDbSetMasterTable

Syntax:

```
INT LlDbSetMasterTable(HLLJOB hJob, LPCTSTR pszTableID);
```

Task:

If the master data is passed as variables, List & Label needs to know which table is the "master" table in order to be able to offer the suitable sub-tables in the table structure window. If you set the master table name using this method, all tables related to this table can be inserted at the root level of the report container object.

Parameter:

hJob: List & Label job handle

pszTableID: ID of the table which is used as the "master" table. Must be identical to the ID passed in *LlDbAddTable()*.

Return Value:

Error code

Hints:

See the hints in chapter "Printing Relational Data". Before using the call, the table must be passed with *LlDbAddTable()*.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDbAddTable(hJob, "Orders", NULL);
LlDbSetMasterTable(hJob, "Orders");
<... etc ...>
LlJobClose(hJob);
```

See also:

LlDbAddTable, LlDbAddTableRelation, LlPrintDbGetCurrentTable, LlPrint-DbGetCurrentTableSortOrder, LlPrintDbGetCurrentTableRelation

LlDebugOutput

Syntax:

```
void LlDebugOutput (INT nIndent, LPCTSTR pszText);
```

Task:

Prints the text in the debug window of the Debwin Tool or – depending on the parameter passed to *LlSetDebug()* - to the log file.

Parameter:

nIndent: Indentation of the following line(s)

pszText: Text to be printed

Hints:

The indentation is very handy for tracing calls to sub-procedures, but you need to make sure that every call with an indentation of +1 is matched by a call with the indentation of -1!

Example:

```
HLLJOB hJob;

LlSetDebug(LL_DEBUG_CMBTLL);
LlDebugOutput(+1,"Get version number:");
hJob = LlJobOpen(0);
v = LlGetVersion(VERSION_MAJOR);
LlJobClose(hJob);
LlDebugOutput(-1,"...done");
```

prints the following to the debug screen:

```
Get version number:
   @LlJobOpen(0)=1
   @LlGetVersion(1)=26
   @LlJobClose(1)
...done
```

See also:

LlSetDebug, Debwin

LlDefineChartFieldExt

Syntax:

```
INT LlDefineChartFieldExt (HLLJOB hJob, LPCTSTR lpszName,
   LPCTSTR lpszCont, INT lPara, LPVOID lpPara);
```

Task:

Defines a chart field and its contents.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to a string with the name of the field

lpszCont: Pointer to a string with the contents of the field

lPara: Field type (*LL_TEXT*, *LL_NUMERIC*, ...)

lpPara: For future extensions, must be NULL.

Return Value:

Error code

Hints:

Please note the general hints in the section "Variables and Fields in List & Label".

See also:

LlDefineChartFieldStart

LlDefineField

Syntax:

```
INT LlDefineField (HLLJOB hJob, LPCTSTR lpszName, LPCTSTR lpszCont);
```

Task:

Defines a list/table field and its contents.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to a string with the name of the field

lpszCont: Pointer to a string with the contents of the field

Return Value:

Error code

Hints:

Please note the general hints in the section "Variables and Fields in List & Label".

This function defines a text field and can be mixed with the other *LlDefineField...()* functions.

LlDefineField(...) is identical to LlDefineFieldExt(..., *LL_TEXT*, NULL).

List & Label predefines the following fields:

Field	Meaning
LL.CountDataThisPage	Numerical, footer field, defined data records per page
LL.CountData	Numerical, footer field, defined data records total
LL.CountPrintedDataThisPage	Numerical, footer field, printed data records per page
LL.CountPrintedData	Numerical, footer field, printed data records total
LL.FCountDataThisPage	Numerical, footer field, defined data records per page
LL.FCountData	Numerical, footer field, defined data records total
LL.FCountPrintedDataThisPage	Numerical, footer field, printed data records per page
LL.FCountPrintedData	Numerical, footer field, printed data records total

The difference between "defined" and "printed" data records is that the user can apply a record filter to the table so that the "defined" numbers increase with every data record sent from the program, but not necessarily the "printed" ones.

Example:

```
HLLJOB hJob;

hJob = LlJobOpen(0);
LlDefineFieldStart(hJob);
LlDefineField(hJob, "Name", "Smith");
LlDefineField(hJob, "Forename", "George");
LlDefineFieldExt(hJob, "Residence", "Cambridge", LL_TEXT, NULL);
LlDefineFieldExt(hJob, "Postal Code", "*CB5 9NB*", LL_BARCODE_3OF9);
<... etc ...>
LlJobClose(hJob);
```

See also:

LlDefineFieldStart, LlDefineFieldExt, LlDefineFieldExtHandle

LlDefineFieldExt

Syntax:

```
INT LlDefineFieldExt (HLLJOB hJob, LPCTSTR lpszName,
   LPCTSTR lpszCont, INT lPara, LPVOID lpPara);
```

Task:

Defines a list/table field and its contents.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to a string with the name of the field

lpszCont: Pointer to a string with the contents of the field

lPara: Field type (*LL_TEXT*, *LL_NUMERIC*, ...), if necessary combined with 'or' (see below).

lpPara: For future extensions, must be NULL.

Return Value:

Error code

Hints:

Please note the general hints in the section "Variables and Fields in List & Label".

List & Label predefines the fields listed in *LlDefineField()*.

lPara 'or'ed with *LL_TABLE_FOOTERFIELD* supplies field definitions **only** for the list footer. The footer is dynamically linked to the list body and is suitable for, e.g., dynamic calculations as the line for totals or sub-totals.

lPara 'or'ed with *LL_TABLE_HEADERFIELD* supplies field definitions **only** for the list header.

lPara 'or'ed with *LL_TABLE_GROUPFIELD* supplies field definitions **only** for the group.

lPara 'or'ed with *LL_TABLE_GROUPFOOTERFIELD* supplies field definitions **only** for the group footer.

lPara 'or'ed with *LL_TABLE_BODYFIELD* supplies field definitions **only** for the list body.

If none of these flags is used, the fields appear in all field selection dialogs in the table object.

Example:

```
HLLJOB hJob;

hJob = LlJobOpen(0);
LlDefineFieldStart(hJob);
LlDefineField(hJob, "Name", "Smith");
LlDefineField(hJob, "Forename", "George");
LlDefineFieldExt(hJob, "Residence", "Cambridge", LL_TEXT, NULL);
LlDefineFieldExt(job, "Number of entries per page",
   "1", LL_TABLE_FOOTERFIELD Or LL_TEXT, NULL)
LlDefineFieldExt(hJob, "Postal code",
   "*CB5 9NB*", LL_BARCODE_3OF9);
LlDefineFieldExt(hJob, "Photo",
   "c:\\photos\\norm.bmp", LL_DRAWING);
<... etc ...>
LlJobClose(hJob);
```

See also:

LlDefineFieldStart, LlDefineField, LlDefineFieldExtHandle

LlDefineFieldExtHandle

Syntax:

```
INT LlDefineFieldExtHandle (HLLJOB hJob, LPCTSTR lpszName,
   HANDLE hContents, INT lPara, LPVOID lpPara);
```

Task:

Defines a list field and its (handle) contents.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to a string with the name of the field

hContents: Handle to an object of type HMETAFILE, HENHMETAFILE, HICON or HBITMAP

lPara: *LL_DRAWING_HMETA*, *LL_DRAWING_HEMETA*, *LL_DRAWING_HICON* or *LL_DRAWING_HBITMAP*

lpPara: For further extensions, must be NULL.

Return Value:

Error code

Hints:

Please note the general hints in the section "Variables and Fields".

This function defines a text field and can be mixed with the other *LlDefineField...()* functions.

List & Label predefines the fields listed in *LlDefineField()*.

The metafile handle must be valid as long as it is needed, that is during the entire layout definition or until after *LlPrintFields()* or *LlPrint()* has finished.

After use the handle can or should be deleted with the normal API function.

Example:

```
HLLJOB hJob;
HMETAFILE hMeta;
HDC    hMetaDC;
INT    i;
hMetaDC = CreateMetaFile(NULL);  // Fieberkurve
selectObject(hMetaDC,GetStockObject(NULL_PEN));
Rectangle(hMetaDC, 0, 0, LL_META_MAXX, LL_METY_MAXY);
for (i = 0; i < 10; ++i)
{
   MoveTo(hMetaDC,0,MulDiv(i, LL_META_MAXY, 10));
   LineTo(hMetaDC,MulDiv(i, LL_META_MAXX, 100),
         MulDiv(i, LL_META_MAXY, 10);
}
MoveTo(hMetaDC, 0, MulDiv(((100*i) & 251) % 100, LL_META_MAXY,100));
for (i = 0; i < 10; ++i)
   LineTo(hMetaDC,MulDiv(i, LL_META_MAXX, 10),
         MulDiv(((100*i) & 251) % 100, LL_META_MAXY, 100));
hMeta = CloseMetaFile(hMetaDC);

hJob = LlJobOpen(0);
LlDefineFieldStart(hJob);
LlDefineField(hJob, "Name", "Normalverbraucher");
LlDefineField(hJob, "Vorname", "Otto");
LlDefineFieldExt(hJob, "Ort", "Konstanz", LL_TEXT, NULL);
LlDefineFieldExtHandle(hJob, "Erfolgs-Chart", hMeta,
      LL_DRAWING_HMETA, NULL);
<... etc ...>
LlJobClose(hJob);
DeleteObject(hMeta);
```

See also:

LlDefineFieldStart, LlDefineField, LlDefineFieldExt

LlDefineFieldStart

Syntax:

```
void LlDefineFieldStart (HLLJOB hJob);
```

Task:

Empties List & Label's internal field buffer in order to delete old field definitions.

Parameter:

hJob: List & Label job handle

Hints:

The hints for *LlDefineVariableStart()* also apply to this function.

If the function *LlPrintIsFieldUsed()* is used in your application, *LlDefineFieldStart()* may not be used after the call to *LlPrint[WithBox]Start()*, otherwise the "used" flag will be reset and *LlPrintIsFieldUsed()* returns always FALSE. We recommend the usage of *LlGetUsedIdentifiers* anyway.

Important: This function must not be called within the print loop!

Example:

```
HLLJOB hJob;

hJob = LlJobOpen(0);
LlDefineFieldStart(hJob);
LlDefineField(hJob, "Name", "Smith");
LlDefineField(hJob, "forename", "George");
<... etc ...>
LlJobClose(hJob);
```

See also:

LlDefineField, LlDefineFieldExt, LlDefineFieldExtHandle

LlDefineLayout

Syntax:

```
INT LlDefineLayout (HLLJOB hJob, HWND hWnd, LPCTSTR lpszTitle,
   UINT nObjType, LPCTSTR lpszObjName);
```

Task:

Calls the interactive Designer that will be displayed as a modal pop-up window overlapping your application window.

Parameter:

hJob: List & Label job handle

hWnd: Handle of the application window which will be disabled while the Designer is being displayed.

lpszTitle: Window title

nObjType: Project type

Value	Meaning
LL_PROJECT_LABEL	for labels
LL_PROJECT_CARD	for cards
LL_PROJECT_LIST	for lists

if necessary combined with 'or' with:

Value	Meaning
LL_FIXEDNAME	Deletes the menu items 'new' and 'load' (preferred: *LlDesignerProhibitAction()*)
LL_NOSAVEAS	Deletes the menu item 'save as' (preferred: *LlDesignerProhibitAction()*)
LL_NONAMEINTITLE	No file name of the current project in List & Label's main window title

lpszObjName: File name of the desired object with file extension

Return Value:

Error code

Hints:

The window handle is used to deactivate the calling program.

If this is not desired or possible, NULL can also be passed. In this case the calling program is responsible for closing the layout editor, should the user abort the main program. This is not recommended.

When the List & Label layout Designer is minimized, the calling program is also automatically minimized; when the Designer is subsequently restored, List & Label is also restored.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDefineVariableStart(hJob);
LlDefineVariable(hJob, "Name", "Smith");
LlDefineVariable(hJob, "forename", "George");
LlDefineVariable(hJob, "PIN", "40|08150|77500",
```

```
            LL_BARCODE_EAN13, NULL);
LlDefineLayout(hJob, hWndMain, "Test", LL_PROJECT_LABEL,
   "test.lbl")
LlJobClose(hJob);
```

See also:

LlDesignerProhibitAction, LlSetOption, LlSetFileExtensions

LlDefineSumVariable

Syntax:

```
INT LlDefineSumVariable (HLLJOB hJob, LPCTSTR lpszName,
   LPCTSTR lpszCont);
```

Task:

Defines a sum variable's contents.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to a string with the name of the variable

lpszCont: Pointer to a string with the contents of the variable

Return Value:

Error code

Hints:

The field content is interpreted as numerical value.

Use of this function usually conflicts with a user who can edit a layout, as the sum variable will not have the value he expects.

Example:

```
HLLJOB hJob;

hJob = LlJobOpen(0);
<... etc ...>
LlDefineSumVariable(hJob, "@Sum01", "14");
<... etc ...>
LlJobClose(hJob);
```

See also:

LlGetSumVariableContents

LlDefineVariable

Syntax:

```
INT LlDefineVariable (HLLJOB hJob, LPCTSTR lpszName, LPCTSTR
   lpszCont);
```

Task:

Defines a variable of the type *LL_TEXT* and its contents.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to a string with the name of the variable

lpszCont: Pointer to a string with the contents of the variable

Return Value:

Error code

Hints:

Please note the general hints in the section "Variables and Fields in List & Label".

This function defines a text variable and can be mixed with the other *LlDefineVariable...()* functions.

LlDefineVariable(...) is identical to *LlDefineVariableExt(..., LL_TEXT, NULL)*.

List & Label predefines the following variables:

Field	Meaning
LL.CountDataThisPage	Numerical, footer field, defined data records per page
LL.CountData	Numerical, footer field, defined data records total
LL.CountPrintedDataThisPage	Numerical, footer field, printed data records per page
LL.CountPrintedData	Numerical, footer field, printed data records total
LL.SortStrategy	String, sort expression
LL.FilterExpression	String, filter expression

The difference between "defined" and "printed" data records is that the user can apply a filter condition to the data records so that with every data record sent from the program the "defined" numbers increase, but not necessarily the "printed" ones (the latter values are only increased if the data record has been printed, that is, matches the filter condition).

Example:

```
HLLJOB hJob;

hJob = LlJobOpen(0);
LlDefineVariableStart(hJob);
LlDefineVariable(hJob, "Name", "Smith");
LlDefineVariable(hJob, "forename", "George");
LlDefineVariableExt(hJob, "residence", "Cambridge, LL_TEXT, NULL);
```

```
LlDefineVariableExt(hJob, "Postal Code", "*CB1*",
         LL_BARCODE_3OF9, NULL);
<... etc ...>
LlJobClose(hJob);
```

See also:

LlDefineVariableStart, LlDefineVariableExt, LlDefineVariableExtHandle, LlGetVariableContents, LlGetVariableType

LlDefineVariableExt

Syntax:

```
INT LlDefineVariableExt (HLLJOB hJob, LPCTSTR lpszName,
   LPCTSTR lpszCont, INT lPara, LPVOID lpPara);
```

Task:

Defines a variable and its contents.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to a string with the name of the variable

lpszCont: Pointer to a string with the contents of the variable

lPara: Variable type (*LL_TEXT*, *LL_NUMERIC*, ...)

lpPara: For future extensions, must be NULL.

Return Value:

Error code

Hints:

Please note the general hints in the section "Variables and Fields in List & Label".

This function can be mixed with the other *LlDefineVariable...()*-functions.

The variables predefined by List & Label are listed within the description of *LlDefineVariable()*.

Example:

```
hJob = LlJobOpen(0);
LlDefineVariableStart(hJob);
LlDefineVariableExt(hJob, "City", "Washington", LL_TEXT, NULL);
LlDefineVariableExt(hJob, "ZIP Code", "*90206*",
         LL_BARCODE_3OF9, NULL);
LlDefineVariableExt(hJob, "Photo", "i.bmp", LL_DRAWING, NULL);
LlJobClose(hJob);
```

See also:

LlDefineVariableStart, LlDefineVariable, LlDefineVariableExtHandle, LlGetVariableContents, LlGetVariableType

LlDefineVariableExtHandle

Syntax:

```
INT LlDefineVariableExtHandle (HLLJOB hJob, LPCTSTR lpszName,
   HANDLE hContents, INT lPara, LPVOID lpPara);
```

Task:

Defines a variable and its contents.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to a string with the name of the variable

hContents: Handle to an object of type:HMETAFILE, HENHMETAFILE,HICON or HBITMAP

lPara: *LL_DRAWING_HMETA*, *LL_DRAWING_HEMETA*, *LL_DRAWING_HICON* or *LL_DRAWING_HBITMAP*

lpPara: For future extensions, must be NULL.

Return Value:

Error code

Hints:

Please note the general hints in the section "Variables and Fields in List & Label".

This function can be mixed with the other *LlDefineVariable...()*-functions.

The handle must be valid as long as it is needed, that is during the entire layout definition or until after *LlPrintFields()* or *LlPrint()* return.

After use the handle can or should be deleted with the normal API function.

Example:

```
HLLJOB hJob;
HMETAFILE hMeta;
HDC    hMetaDC;
INT    i;

hMetaDC = CreateMetaFile(NULL);  // curve
SelectObject(hMetaDC, GetStockObject(NULL_PEN));
Rectangle(hMetaDC, 0, 0, LL_META_MAXX, LL_META_MAXY);
for (i = 0; i < 10;++ i)
{
   MoveTo(hMetaDC, 0, MulDiv(i, LL_META_MAXY, 10));
   LineTo(hMetaDC, MulDiv(i, LL_META_MAXX, 100), MulDiv(,
LL_META_MAXY, 10);
}
MoveTo(hMetaDC, 0, MulDiv(((100*i) & 251) % 100, LL_META_MAXY, 100));
for (i = 0; i < 10;++ i)
   LineTo(hMetaDC, MulDiv(i, LL_META_MAXX, 10),
      MulDiv(((100*i) & 251) % 100, LL_META_MAXY, 100));
```

```
hMeta = CloseMetaFile(hMetaDC);

hJob = LlJobOpen(0);
LlDefineVariableStart(hJob);
LlDefineVariable(hJob, "Name", "Smith");
LlDefineVariable(hJob, "Forename", "George");
LlDefineVariableExtHandle(hJob, "Chart", hMeta,
                          LL_DRAWING_META, NULL);
LlDefineVariableExt(hJob, "Postal code", "*CB5 4RB*",
                          LL_BARCODE_3OF9, NULL);
<... etc ...>
LlJobClose(hJob);
DeleteObject(hMeta);
```

See also:

LlDefineVariableStart, LlDefineVariable, LlDefineVariableExt, LlGetVariableContents, LlGetVariableType

LlDefineVariableStart

Syntax:

```
void LlDefineVariableStart (HLLJOB hJob);
```

Task:

Empties List & Label's internal variable buffer in order to delete old definitions.

Parameter:

hJob: List & Label job handle

Hints:

Does not necessarily have to be called. However, as with every *LlDefineVariable...()* the internal variable list is checked for a variable which is already available with the same name and type, this can be somewhat accelerated with this function. Otherwise you only need to redefine the variables whose contents change as the old contents of the variable are " over-written" ; the contents of the remaining variables remain the same.

If you use *LlPrintIsVariableUsed()*, *LlDefineVariableStart()* may not be called after the invocation of *LlPrint[WithBox]Start()*, otherwise *LlPrintIsVariableUsed()* will always return FALSE.

Important: This function must not be called within the print loop!

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);
LlDefineVariableStart(hJob);
LlDefineVariable(hJob, "Name", "Smith");
LlDefineVariable(hJob, "Forename", "George");
<...etc ...>
LlDefineVariable(hJob, "Forename", "James");
```

```
<... etc ...>
LlJobClose(hJob);
```

See also:

LlDefineVariable, LlDefineVariableExt, LlDefineVariableExtHandle, LlGetVariableContents, LlGetVariableType

LlDesignerAddAction

Syntax:

```
INT LlDesignerAddAction(HLLJOB hJob, UINT nID, UINT nFlags,
   LPCTSTR pszMenuText, LPCTSTR pszMenuHierarchy,
   LPCTSTR pszTooltipText, UINT nIcon, LPVOID pvReserved);
```

Task:

Extends the Designer's menu and optionally the toolbar of the Designer. In contrast to using the callback *LL_CMND_MODIFYMENU* a command button with a selectable icon can be added to the toolbar here. This command must be called before *LlDefineLayout()*.

Parameter:

hJob: List & Label Job-Handle

nID: Menu-ID for the new action to be added. This ID is passed by the callback LL_*CMND_SELECTMENU*, when the user selects the corresponding menu item or toolbar button. User defined IDs should be in the range between 10100 and 10999.

nFlags: Combination (ORed) of the following flags:

Value	Meaning
LLDESADDACTIONFLAG_ADD_TO_TOOLBAR	Add a command button to the toolbar in addition to the menu item.
LLDESADDACTION_MENUITEM_APPEND	The menu item is added behind the entry in pszMenuHierarchy.
LLDESADDACTION_MENUITEM_INSERT	The menu item is added in front of the entry pszMenuHierarchy.

As well as an optional Keycode as a Shortcut and a combination of the following flags as modifiers:

Value	Meaning
LLDESADDACTION_ACCEL_CONTROL	Keyboard shortcut is CTRL+Keycode.
LLDESADDACTION_ACCEL_SHIFT	Keyboard shortcut is SHIFT+Keycode.

Value	Meaning
LLDESADDACTION_ACCEL_ALT	Keyboard shortcut is ALT+Keycode.
LLDESADDACTION_ACCEL_VIRTKEY	Should always be set.

pszMenuText: Menu text without a keyboard shortcut (this will be added automatically). You can however, use the "&" symbol to allocate the shortcuts for menu navigation. Use "." as a hierarchy separator to create submenu items. For example, in order to create a Menu "Draft" with a sub-point "Invoices", use "Draft.Invoices" as a menu text.

pszMenuHierarchy: Menu hierarchy of the new menu item. The description is given in the form of "<Level>.<Level>..." whereby "Level" is always the 0-based index of the menu entry. For example, to insert a new entry in the first place in the "Edit" menu, use "1.0" and *LLDESADDACTION_MENUITEM_INSERT*.

pszTooltipText: Text for the tooltip on the toolbar command button. Will only be evaluated if the flag *LLDESADDACTIONFLAG_ADD_TO_TOOLBAR* is set. May be NULL.

nIcon: Icon-ID for the command button. Will only be evaluated if the flag *LLDESADDACTIONFLAG_ADD_TO_TOOLBAR* is set. Use the program IconSelector.exe (in the Tools directory) to see the list of available icons with their IDs.

pvReserved: For future extensions, must be NULL.

Return Value:

Error code

Hints:

To execute the actual action, the *LL_CMND_SELECTMENU*-Callback has to be processed.

See also:

LlDefineLayout

LlDesignerFileOpen

Syntax:

```
INT LlDesignerFileOpen(HLLJOB hJob, LPCTSTR pszFileName,
   UINT nFlags);
```

Task:

Opens the specified project file when the Designer is open.

Parameter:

hJob: List & Label Job-Handle

pszFileName: Project file name including path and file extension

nFlags: Combination (ORed) of a flag from the following two groups at any one time:

Value	Meaning
LL_DESFILEOPEN_OPEN_EXISTING	File must already exist, otherwise an Error Code will be returned.
LL_DESFILEOPEN_CREATE_ALWAYS	File will always be newly created. If file already exists, then file content will be deleted.
LL_DESFILEOPEN_CREATE_NEW	File will always be newly created if not already existing. If file already exists, an error code will be returned.
LL_DESFILEOPEN_OPEN_ALWAYS	If file exists, it will be opened, otherwise new file will be created.
LL_DESFILEOPEN_OPEN_IMPORT	Imports an existent file into an already opened project.

Value	Meaning
LL_DESFILEOPENFLAG_SUPPRESS_SAVEDIALOG	The currently opened project will be automatically saved without user interaction before loading a new project.
LL_DESFILEOPENFLAG_SUPPRESS_SAVE	The currently opened project will be closed automatically without being saved. All changes after the last save will therefore be lost!
LL_DESFILEOPENFLAG_DEFAULT	The currently opened project will be saved or closed as selected by the user – if necessary before the new project is loaded.

Return Value:

Error code

Hints:

The function can only be used within a designer event. Typical use is in connection with *LIDesignerAddAction()* in order to automate certain application workflows.

See also:

LlDesignerFileSave

LlDesignerFileSave

Syntax:

```
INT LlDesignerFileSave(HLLJOB hJob, UINT nFlags);
```

Task:

Saves the currently opened project file when the Designer is open.

Parameter:

hJob: List & Label Job-Handle

nFlags: For future extension, must be "0" (LL_DESFILESAVE_DEFAULT).

Return Value:

Error code

Hints:

The function can only be used within a designer event. Typical use is in connection with *LlDesignerAddAction()* in order to automate certain application workflows.

See also:

LlDesignerFileOpen

LlDesignerGetOptionString

Syntax:

```
INT LlDesignerGetOptionString(HLLJOB hJob, INT nMode,
   LPTSTR pszBuffer, UINT nBufSize);
```

Task:

Queries various settings when the Designer is open.

Parameter:

hJob: List & Label Job-Handle

nMode: Option index, see *LlDesignerSetOptionString()*

pszBuffer: Buffer for return value, may be NULL

nBufSize: Size of buffer

Return value:

Error code or buffer size needed, if pszBuffer is NULL.

Hints:

Valid values for the mode parameter can be found at the description of *LlDesignerSetOptionString()*.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlDesignerSetOptionString

LlDesignerInvokeAction

Syntax:

```
INT LlDesignerInvokeAction(HLLJOB hJob, INT nMenuIndex);
```

Task:

Activates the action menu item if the Designer is open.

Parameter:

hJob: List & Label Job-Handle

nMenuIndex: Index of function. Available functions can be found in file MENUID.TXT.

Return Value:

Error code

Hints:

If the function is to be utilized, it must be used within a designer event. Typical use is in connection with *LlDesignerAddAction()* in order to automate certain application workflows.

See also:

LlDefineLayout, LlDesignerAddAction

LlDesignerProhibitAction

Syntax:

```
INT LlDesignerProhibitAction (HLLJOB hJob, INT nMenuIndex);
```

Task:

Hiding of menu items in the Designer (and their respective toolbar buttons).

Parameter:

hJob: List & Label job handle

nMenuIndex: Menu function index

The function index can have the following values:

Value	Meaning
0	All function exclusions are deleted, the menu item list is reset (default menu is restored). This is automatically called by *LlJobOpen()* and *LlJobOpenLCID()*. This function needs to be used for several *LlDefineLayout()* calls with different lock entries, otherwise the lock entries will be added.
LL_SYSCOMMAND_-MINIMIZE	The Designer window cannot be minimized (iconized).
LL_SYSCOMMAND_-MAXIMIZE	The Designer window cannot be maximized.
other	The menu IDs of the deleted menus can be given here. The IDs of the menu items in List & Label can be found in the file MENUID.TXT included in your package.

Return Value:

Error code

Hints:

If this function is used, it must be called before the function *LlDefineLayout()*.

This call can be made several times in turn for different function index values as the entries are added to a lock-entry list which is evaluated at the call of *LlDefineLayout()*. They can even be called during the *LL_CMND_MODIFYMENU* callback.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDefineVariableStart(hJob);
LlDefineVariable(hJob, "Name", "Smith");
LlDefineVariable(hJob, "Forename", "George");
LlDefineVariable(hJob, "PIN", "40|08150|77500", LL_BARCODE_EAN13,
NULL);
LlDesignerProhibitAction(hJob, LL_SYSCOMMAND_MAXIMIZE);
LlDesignerProhibitAction(hJob, LL_SYSCOMMAND_MINIMIZE);
LlDefineLayout(hJob, hWndMain, "Test", LL_PROJECT_LABEL, "test.lbl")
LlJobClose(hJob);
```

See also:

LlDefineLayout, LlDesignerProhibitEditingObject, LlDesignerProhibitFunction

LlDesignerProhibitEditingObject

Syntax:

```
INT LlDesignerProhibitEditingObject(HLLJOB Job, LPCTSTR pszObject);
```

Task:

Prohibits the editing of the passed object.

Parameter:

hJob: List & Label job-handle

pszObject: Object name

Return Value:

Error code

Hints:

With NULL or "" the list of prohibited objects will be deleted.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDesignerProhibitEditingObject(hJob, "MyText");
...
LlJobClose(hJob);
```

See also:

LlDefineLayout, LlDesignerProhibitAction, LlDesignerProhibitFunction

LlDesignerProhibitFunction

Syntax:

```
INT LlDesignerProhibitFunction (HLLJOB hJob, LPCTSTR pszFunction);
```

Task:

Hides the given function in the formula wizard. Must be called before any functions are evaluated.

Parameter:

hJob: List & Label job handle

pszFunction: Function name.

Return Value:

Error code

Hints:

If you pass NULL or an empty string, the list of functions to be hidden will be reset.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDesignerProhibitFunction(hJob, "");
LlDesignerProhibitFunction(hJob, "CStr$");
...
LlJobClose(hJob);
```

See also:

LlDefineLayout, LlDesignerProhibitAction, LlDesignerProhibitEditingObject

LlDesignerRefreshWorkspace

Syntax:

```
INT LlDesignerRefreshWorkspace(HLLJOB hJob);
```

Task:

Activates an update of all tool windows, menu items etc. in the Designer. Use this function to ensure that the Designer immediately shows all the changes made to the object model using DOM within the open Designer.

Parameter:

hJob: List & Label Job-Handle

Return code:

Error code

Hints:

This function can only be used within a designer event. It is typically used in connection with *LlDesignerAddAction()*.

See also:

LlDefineLayout, LlDesignerAddAction

LlDesignerSetOptionString

Syntax:

```
INT LlDesignerSetOptionString (HLLJOB hJob, INT nMode,
   LPCTSTR pszValue);
```

Task:

Defines various settings when the Designer is open.

Parameter:

hJob: List & Label Job-Handle

nMode: The following values are possible as function index:

LL_DESIGNEROPTSTR_PROJECTFILENAME

The name of the project currently opened. If you have created a new file through an action, it can be named in this way. Otherwise corresponds to a "Save as…".

LL_DESIGNEROPTSTR_ WORKSPACETITLE

Assigns the window title in the Designer. You can use the format placeholder %s within the text to show the project name.

LL_DESIGNEROPTSTR_ PROJECTDESCRIPTION

Assigns the project description that will also be shown in "Open file" dialog.

pszValue: new value

Return value:

Error code

See also:

LlDesignerGetOptionString

LlDlgEditLineEx

Syntax:

```
INT LlDlgEditLineEx(HLLJOB Job, HWND hWnd, LPTSTR pszBuffer,
   UINT nBufSize, UINT nParaTypes, LPCTSTR pszTitle,
   BOOL bTable, LPVOID pReserved);
```

Task:

This function is only available in the Enterprise edition! Starts the List & Label formula wizard independently of the Designer. This means that List & Label formulas can also be used at points in the application that are independent of printing.

Parameter:

hJob: List & Label job handle

hWnd: Window handle of the calling program

pszBuffer: Buffer for return value

nBufSize: Size of buffer

nParaTypes: Expected return type. One or several ORed data type constants (e.g. LL_TEXT, LL_DATE)

pszTitle: Window title. Note that the title will be preceded by the word "Edit".

bTable: Defines whether only variables (*FALSE*) or fields (*TRUE*) will be available

pReserved: Reserved, must be NULL or empty (").

Hints:

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

Return value:

Error code

LlDomCreateSubobject

Syntax:

```
INT LlDomCreateSubobject(HLLDOMOBJ hDOMObj, INT nPosition,
   LPCTSTR pszType, PHLLDOMOBJ phDOMSubObj);
```

Task:

Creates a new subobject within the specified DOM list. Detailed application examples can be found in chapter "DOM Functions".

Parameter:

hDomObj: DOM handle for the list

nPosition: Index (0-based) of the element to be inserted. All following elements are moved back one position.

pszType: Desired element type, e.g. "Text", for creating a new text object in the object list

phDOMSubObj: Pointer to DOM handle for return

Return value:

Error code

See also:

Chapter "DOM Functions"

LlDomDeleteSubobject

Syntax:

```
INT LlDomDeleteSubobject(HLLDOMOBJ hDOMObj, INT nPosition);
```

Task:

Deletes a subobject from the specified DOM list. Detailed application examples can be found in chapter "DOM Functions".

Parameter:

hDomObj: DOM handle for the list

nPosition: Index (0-based) of the element to be deleted. All following elements are moved forward one position.

Return value:

Error code

See also:

Chapter "DOM Functions"

LlDomGetObject

Syntax:

```
INT LlDomGetObject(HLLDOMOBJ hDOMObj, LPCTSTR pszName,
   PHLLDOMOBJ phDOMSubObj);
```

Task:

Provides a subobject of the specified DOM object, and is used e.g. to request the object list from the project. Detailed application examples can be found in chapter "DOM Functions".

Parameter:

hDomObj: DOM handle for the "parent" object

pszName: Name of the desired subobject, e.g. "Objects"

phDOMSubObj: Pointer to DOM handle for return

Return value:

Error code

See also:

Chapter "DOM Functions"

LlDomGetProject

Syntax:

```
INT LlDomGetProject(HLLJOB hJob, PHLLDOMOBJ phDOMObj);
```

Task:

Provides the currently loaded project object. Can be used e.g. after *LlPrint(WithBox)Start*, to change the project during printout with DOM functions. In order to create new projects or to edit projects *before* printout, use *LlProjectOpen(), LlDomGetProject(), LlProjectSave() and LlProjectClose()*. Detailed application examples can be found in chapter "DOM Functions".

Parameter:

hJob: List & Label job handle

phDOMSubObj: Pointer to DOM handle for return

Return value:

Error code

See also:

LlProjectOpen, LlProjectSave, LlProjectClose , chapter "DOM Functions"

LlDomGetProperty

Syntax:

```
INT LlDomGetProperty(HLLDOMOBJ hDOMObj, LPCTSTR pszName,
   LPTSTR pszBuffer, UINT nBufSize);
```

Task:

Returns the content of the specified property. Detailed application examples can be found in chapter "DOM Functions".

Parameter:

hDomObj: DOM handle for the object to be queried

pszName: Name of the desired property, e.g. "Condition", for requesting the appearance condition of an object.

pszBuffer: Buffer for the return value. Can be NULL (see notes)

nBufSize: Size of buffer

Return value:

Error code or required buffer size

Hints:

If pszBuffer is NULL, the return value is the length of the required buffer (in TCHARS, so BYTEs for SBCS/MBCS and WCHARs for UNICODE) including the string termination.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

Chapter "DOM Functions"

LlDomGetSubobject

Syntax:

```
INT LlDomGetSubobject(HLLDOMOBJ hDOMObj, INT nPosition,
   PHLLDOMOBJ phDOMSubObj);
```

Task:

Returns the specified subelement of the DOM list. Detailed application examples can be found in chapter "DOM Functions".

Parameter:

hDomObj: DOM handle for the list to be queried

nPosition: Index (0-based) of the desired element

phDOMSubObj: Pointer to DOM handle for return

Return value:

Error code

See also:

Chapter "DOM Functions"

LlDomGetSubobjectCount

Syntax:

```
INT LlDomGetSubobjectCount(HLLDOMOBJ hDOMObj, _LPINT pnCount);
```

Task:

Returns the number of elements in the specified DOM list. The number of objects in the object list can be determined, for example. Detailed application examples can be found in chapter "DOM Functions".

Parameter:

hDomObj: DOM handle for the list to be queried

pnCount: Pointer for return

Return value:

Error code

See also:

Chapter "DOM Functions"

LlDomSetProperty

Syntax:

```
INT LlDomSetProperty(HLLDOMOBJ hDOMObj, LPCTSTR pszName,
   LPCTSTR pszValue);
```

Task:

Sets the specified property to the passed value. Detailed application examples can be found in chapter "DOM Functions".

Parameter:

hDomObj: DOM handle for the object to be altered

pszName: Name of the desired property, e.g. "Condition", for setting the appearance condition of an object

pszValue: New value of the property

Return value:

Error code

See also:

Chapter "DOM Functions"

LlEnumGetEntry

Syntax:

```
HLISTPOS LlEnumGetEntry(HLLJOB hJob, HLISTPOS hPos,
   LPSTR pszNameBuf, UINT nNameBufsize, LPSTR pszContBuf,
   UINT nContBufSize, _LPHANDLE pHandle, _LPINT pType);
```

Task:

Returns the name and contents of a variable or (chart) field.

Parameter:

hJob: List & Label job handle

hPos: The handle of the current field iterator

pszNameBuf, nNameBufsize: Buffer where the name should be stored

pszContBuf, nContBufSize: Buffer where the contents should be stored. pszContBuf can be NULL to ignore the contents string.

pHandle: Pointer to a handle where the handle value should be stored. Can be NULL to ignore the handle value. See *LlDefineVariableExtHandle()* and *LlDefineFieldExtHandle()*.

pType: Pointer to an INT, in which the type (*LL_TEXT*, ...) will be stored. May be NULL to ignore the type.

Return Value:

Error code

Hints:

During the *LlEnum...()* functions, a call to *LlDefineVariableStart()* or *LlDefineFieldStart()* is prohibited!

The iterator functions can be used to enumerate variables and/or fields and to get their names, contents and types.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

Example:

The following example traverses the list of variables and prints all of them (*LL_TYPEMASK* is the constant for all possible variable types):

```
HLISTPOS hPos = LlEnumGetFirstVar(hJob, LL_TYPEMASK);
while (hPos != NULL)
{
   TCHAR szName[64+1];
   TCHAR szContents[256+1];
   LlEnumGetEntry(hJob, hPos, szName, sizeof(szName), szContents,
         sizeof(szContents), NULL, NULL);
   printf("%s - %s\n",szName,szContents);
   hPos = LlEnumGetNextEntry(hJob, hPos, LL_TYPEMASK);
}
```

See also:

LlEnumGetFirstVar, LlEnumGetFirstField, LlEnumGetFirstChartField, LlEnumGetNextEntry

LlEnumGetFirstChartField

Syntax:

```
HLISTPOS LlEnumGetFirstChartField (HLLJOB hJob, UINT nFlags);
```

Task:

Returns an iterator for the first chart field. The name does not have to be known, the chart fields are returned in the order in which they are declared to List & Label.

Parameter:

hJob: List & Label job handle

nFlags: Flags for the allowed types of fields (to be 'or'ed):
LL_TEXT, *LL_BOOLEAN*, *LL_BARCODE*, *LL_DRAWING*, *LL_NUMERIC*, *LL_DATE*, *LL_HTML, LL_RTF*, *LL_TYPEMASK* (to iterate all of them)

Return Value:

Iterator of first chart field, or NULL if no field exists.

Hints:

During the iteration, a call to *LlDefineChartFieldStart()* is prohibited!

See also:

LlEnumGetFirstVar, LlEnumGetFirstField, LlEnumGetNextEntry, LlEnumGetEntry

LlEnumGetFirstField

Syntax:

```
HLISTPOS LlEnumGetFirstField (HLLJOB hJob, UINT nFlags);
```

Task:

Returns an iterator for the first field. The name does not have to be known, the fields are returned in the order in which they are declared to List & Label.

Parameter:

hJob: List & Label job handle

nFlags: Flags for the allowed types of fields (to be 'or'ed):
LL_TEXT, *LL_BOOLEAN*, *LL_BARCODE*, *LL_DRAWING*, *LL_NUMERIC*, *LL_DATE*, *LL_HTML*, *LL_RTF*, *LL_TYPEMASK* (to iterate all of them)

Return Value:

Iterator of first field, or NULL if no field exists.

Hints:

During the iteration, a call to *LlDefineFieldStart()* is prohibited!

See also:

LlEnumGetFirstVar, LlEnumGetNextEntry, LlEnumGetEntry

LlEnumGetFirstVar

Syntax:

```
HLISTPOS LlEnumGetFirstVar (HLLJOB hJob, UINT nFlags);
```

Task:

Returns an iterator for the first variable. The name does not have to be known, the variables are returned in the order in which they are declared to List & Label.

Parameter:

hJob: List & Label job handle

nFlags: Flags for the allowed types of fields (to be 'or'ed): *LL_TEXT*, *LL_BOOLEAN*, *LL_BARCODE*, *LL_DRAWING*, *LL_NUMERIC*, *LL_DATE*, *LL_RTF*, *LL_HTML, LL_TYPEMASK* (to iterate all of them)

Return Value:

Iterator of first variable, or NULL if no further variable exists.

Hints:

During the iteration, a call to *LlDefineVariableStart()* is prohibited!

Internal variables are not iterated.

See also:

LlEnumGetFirstField, LlEnumGetNextEntry, LlEnumGetEntry

LlEnumGetNextEntry

Syntax:

```
HLISTPOS LlEnumGetNextEntry (HLLJOB hJob, HLISTPOS hPos, UINT nFlags);
```

Task:

Returns the next field/variable (if any). The iteration starts with *LlEnumGetFirstVar()* or *LlEnumGetFirstField()* and is continued with this function.

Parameter:

hJob: List & Label job handle

hPos: Iterator of the current variable or field

nFlags: Flags for the allowed types of fields (to be 'or'ed): *LL_TEXT*, *LL_BOOLEAN*, *LL_BARCODE*, *LL_DRAWING*, *LL_NUMERIC*, *LL_DATE*, *LL_RTF, LL_HTML*

Return Value:

Iterator for the next variable/field, or NULL if no further variable/field exists.

Hints:

During the *LlEnum...()* functions, a call to *LlDefineVariableStart()* or *LlDefineFieldStart()* is prohibited!

See also:

LlEnumGetFirstVar, LlEnumGetFirstField, LlEnumGetEntry

LlExprError

Syntax:

```
void LlExprError (HLLJOB hJob, LPTSTR lpBuffer, UINT nBuffer Size);
```

Task:

Returns the reason for the error in plain text.

Parameter:

hJob: List & Label job handle

lpBuffer: Address of buffer for error text

nBufferSize: Maximum number of characters to be copied

Return Value:

Error code

Hints:

The function must be called immediately after *LlExprParse()* returns an error.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

Example:

See LlExprParse

See also:

LlExprParse, LlExprEvaluate, LlExprType, LlExprFree

LlExprEvaluate

Syntax:

```
INT LlExprEvaluate (HLLJOB hJOB, HLLEXPR lpExpr, LPTSTR lpBuffer,
   UINT nBufferSize);
```

Task:

Evaluates an expression.

Parameter:

hJob: List & Label job handle

lpExpr: The pointer returned by the corresponding *LlExprParse()*

lpBuffer: Address of buffer for calculated value

nBufferSize: Maximum number of characters to be copied

Return Value:

Error code

Hints:

The buffer is always filled with a zero-terminated string.

Depending on the type of result, the buffer contents are to be interpreted as follows:

Type	Meaning
LL_EXPRTYPE_STRING	The buffer contents are the result string
LL_EXPRTYPE_DOUBLE	The buffer contents are the corresponding representation of the value, for pi e.g. "3.141592". The value is always specified with 6 decimal places.
LL_EXPRTYPE_DATE	The buffer contains the corresponding representation of the Julian value, for example "21548263".
LL_EXPRTYPE_BOOL	The buffer contains either the string "TRUE" or "FALSE".
LL_EXPRTYPE_DRAWING	The buffer contains the name of the drawing variable/drawing field (!), not the contents.
LL_EXPRTYPE_BARCODE	The buffer contains the value which would be interpreted as the barcode.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

Example:

See LlExprParse

See also:

LlExprParse, LlExprType, LlExprError, LlExprFree

LlExprFree

Syntax:

```
void LlExprFree (HLLJOB hJob, HLLEXPR lpExpr);
```

Task:

Releases the parsing tree created by *LlExprParse()*.

Parameter:

hJob: List & Label job handle

lpExpr: The pointer returned from the corresponding *LlExprParse()*

Hints:

To avoid memory leaks, the function must be called when a tree returned by *LlExprParse()* is no longer required.

Example:

See LlExprParse

See also:

LlExprParse, LlExprEvaluate, LlExprType, LlExprError

LlExprGetUsedVars

Syntax:

```
INT LlExprGetUsedVars(HLLJOB hJob, HLLEXPR lpExpr, LPSTR pszBuffer,
UINT nBufSize);
```

Task:

Returns the variables and fields (tab-separated) that were used in a formula with LlExprParse().

Parameter:

hJob: List & Label job handle

lpExpr: The pointer returned by the corresponding *LlExprParse()*

pszBuffer: Buffer for the return value

nBufSize: Size of buffer

Hints:

Corresponds to "LlExprGetUsedVarsEx" with parameter bOrgName = TRUE.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlExprParse, LlExprEvaluate, LlExprType, LlExprError

LlExprGetUsedVarsEx

Syntax:

```
INT LlExprGetUsedVarsEx(HLLJOB hLlJob, HLLEXPR lpExpr, LPSTR
pszBuffer, UINT nBufSize, BOOL bOrgName);
```

Task:

Returns the variables and fields (tab-separated) that were used in a formula with LlExprParse().

Parameter:

hJob: List & Label job handle

lpExpr: The pointer returned by the corresponding *LlExprParse()*

pszBuffer: Buffer for the return value

nBufSize: Size of buffer

bOrgName: TRUE: global field names, FALSE: localized field names

Hints:

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlExprParse, LlExprEvaluate, LlExprType, LlExprError

LlExprParse

Syntax:

```
LPVOID LlExprParse (HLLJOB hJob, LPCTSTR lpExprText,
   BOOL bTableFields);
```

Task:

Tests the expression for correctness and constructs a function tree for this expression.

Parameter:

hJob: List & Label job handle

lpExprText: Expression

bTableFields: TRUE: reference to fields and variables FALSE: reference to variables

Return Value:

Pointer to an internal structure (parsing tree)

Hints:

If an error is signaled (Address = NULL) then you can query the error text with *LlExprError()*.

The variables defined with *LlDefineVariable()* can be integrated into the expression if bTableFields is FALSE, otherwise the fields defined with *LlDefineField()* are included in the expression.

If the expression is used for calculation several times, it is recommended that you translate it once with *LlExprParse()* and then carry out the calculations, releasing the tree at the end.

Example:

```
LPVOID    lpExpr;
char      lpszErrortext[128];
char      lpszBuf[20];
Long      lDateOne;
Long      lDateTwo;

LlDefineVariable(hJob, "Date", "29.2.1964", LL_TEXT);
lpExpr = LlExprParse(hJob, "DateToJulian(DATE(Date))", FALSE);
if (lpExpr)
{
   if (LlExprType(hJob, lpExpr) != LL_EXPRTYPE_DOUBLE)
   {
      // something is wrong, must be numerical!
   }
   LlExprEvaluate(hJob, lpExpr, lpszBuf, sizeof(lpszBuf));
   lDateOne = atol(lpszBuf);
   // lDateOne now has the Julian date
   // 29.2.1964
   LlDefineVariable(hJob, "Date", "28.10.2017", LL_TEXT);
   LlExprEvaluate(hJob, lpExpr, lpszBuf, sizeof(lpszBuf));
   lDateTwo = atol(lpszBuf);
   // lDateTwo now has the Julian date
   LlExprFree(hJob, lpExpr);
}
else
{
   // Error!
   LlExprError(hJob, lpszErrortext, sizeof(lpszErrortext));
}
```

See also:

LlExprEvaluate, LlExprType, LlExprError, LlExprFree

LlExprType

Syntax:

```
INT LlExprType (HLLJOB hJOB, HLLEXPR lpExpr);
```

Task:

Evaluates the result type of the expression.

Parameter:

hJob: List & Label job handle

lpExpr: The pointer returned from the corresponding *LlExprParse()*

Return Value:

Type of result:

Value	Meaning
LL_EXPRTYPE_STRING	String
LL_EXPRTYPE_DOUBLE	Numerical value
LL_EXPRTYPE_DATE	Date
LL_EXPRTYPE_BOOL	Boolean value
LL_EXPRTYPE_DRAWING	Drawing
LL_EXPRTYPE_BARCODE	Barcode

Example:

See LlExprParse

See also:

LlExprParse, LlExprEvaluate, LlExprError, LlExprFree

LlGetChartFieldContents

Syntax:

```
INT LlGetChartFieldContents (HLLJOB hJob, LPCTSTR lpszName,
   LPTSTR lpszBuffer, UINT nBufSize);
```

Task:

Returns the contents of the corresponding chart field.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to a string with the name of the chart field

lpszBuffer: Address of buffer for contents

nBufSize: Maximum number of characters to be copied

Return Value:

Error code (*LL_ERR_UNKNOWN_FIELD* or 0)

Hints:

This function can be used in callback routines to ask for the contents of chart fields.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlDefineChartFieldStart, LlDefineChartFieldExt, LlGetFieldType

LlGetDefaultPrinter

Syntax:

```
INT LlGetDefaultPrinter(LPTSTR pszPrinter, LLPUINT pnBufferSize,
   _PDEVMODE pDevMode, LLPUINT pnDevModeBufSize, UINT nOptions)
```

Task:

Returns the name of the default printer and a DEVMODE struct corresponding to the default settings.

Parameter:

pszPrinter: Address of buffer for the printer name. May be NULL (see hints).

pnBufferSize: Size of the buffer (in TCHARs).

pDevMode: Address of buffer for the DEVMODE struct. May be NULL (see hints)

pnDevModeBufSize: Size of the buffer (in bytes).

nOptions: Reserved, must be 0.

Return Value:

Error code

Hints:

If pszPrinter and pDevMode is NULL, the required buffer sizes are stored in pnPrinterBufferSize and pnDevModeBufferSize.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlSetPrinterToDefault, LlSetPrinterInPrinterFile

LlGetDefaultProjectParameter

Syntax:

```
INT LlGetDefaultProjectParameter(HLLJOB hLlJob,
   LPCTSTR pszParameter, LPTSTR pszBuffer, INT nBufSize,
   _LPUINT pnFlags)
```

Task:

Returns the default value of a project parameter (see chapter "Project Parameters")

Parameter:

hJob: List & Label job handle

pszParameter: Parameter name. May be NULL (see hints)

pszBuffer: Address of buffer for contents. May be NULL (see hints)

nBufSize: Size of the buffer (in TCHARs).

pnFlags: Pointer to an UINT defining the type of the parameter (for valid values see *LlSetDefaultProjectParameter()*). May be NULL if the value is not required.

Return Value:

Error code or required buffer size

Hints:

If pszParameter is NULL, a semicolon separated list of all USER parameters is returned.

If pszBuffer is NULL, the return value equals the size of the required buffer (in TCHARS) including the termination.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlSetDefaultProjectParameter, LlPrintSetProjectParameter, LlPrintGetProject-Parameter

LlGetErrortext

Syntax:

```
INT LlGetErrortext(INT nError, LPTSTR lpszBuffer, UINT nBufSize);
```

Task:

Provides a localized error message for the passed error code.

Parameter:

nError: Error code

lpszBuffer: Pointer to buffer in which the message is to be stored

nBufSize: Size of buffer

Return value:

Error code or required buffer size

Hints

This function can be used to display an error message. More frequent errors are e.g. LL_ERR_EXPRESSION (-23) or LL_ERR_NOPRINTER (-11). If a job has already been opened, the output will occur in the language of the respective job, otherwise the language of the first language kit found will be used.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

LlGetFieldContents

Syntax:

```
INT LlGetFieldContents (HLLJOB hJob, LPCTSTR lpszName,
   LPTSTR lpszBuffer, UINT nBufSize);
```

Task:

Returns the contents of the corresponding (chart) field.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to a string with the name of the field

lpszBuffer: Address of buffer for contents

nBufSize: Maximum number of characters to be copied

Return Value:

Error code (*LL_ERR_UNKNOWN_FIELD* or 0)

Hints:

This function can be used in callback routines to ask for the contents of fields.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlDefineFieldStart, LlDefineFieldExt, LlDefineFieldExtHandle, LlGetFieldType

LlGetFieldType

Syntax:

```
INT LlGetFieldType (HLLJOB hJob, LPCTSTR lpszName);
```

Task:

Returns the type of the corresponding field.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to a string with the name of the field

Return Value:

Field type (positive), or error code (negative)

Hints:

This function can be used in callback routines to ask for the type of fields.

See also:

LlDefineFieldStart, LlDefineFieldExt, LlDefineFieldExtHandle, LlGetFieldContents

LlGetLastErrorText

Syntax:

```
INT   LlGetLastErrorText(HLLJOB hJob, LPWSTR pszBuffer, UINT
nBufSize);
```

Task:

Returns the List & Label error text and the detailed windows error.

Parameter:

hJob: List & Label job handle

pszBuffer: Buffer for return value

nBufSize: Size of buffer

Return value:

Error code or buffer size needed, if pszBuffer is NULL.

Hints:

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

LlGetNotificationMessage

Syntax:

```
UINT LlGetNotificationMessage (HLLJOB hJob);
```

Task:

Returns the message number for callbacks.

Parameter:

hJob: List & Label job handle

Return Value:

Current message number

Hints:

The default message number has the value of the function RegisterWindowMessage("cmbtLLMessage");

The callback function has higher priority; if it is defined, no message is sent.

Should not be used in components that offer events on their own, e.g. .NET, VCL or OCX components.

Example:

```
HLLJOB    hJob;
UINT      wMsg;

LlSetDebug(TRUE);
hJob = LlJobOpen(0);
v = LlGetNotificationMessage(hJob);
...
LlJobClose(hJob);
```

See also:

LlSetNotificationMessage, LlSetNotificationCallback

LlGetOption

Syntax:

```
INT_PTR LlGetOption (HLLJOB hJob, INT nMode);
```

Task:

Requests various switches and settings (see below) from List & Label.

Parameter:

hJob: List & Label job handle

nMode: Option index, see *LlSetOption()*

Return Value:

The value of the corresponding option

Hints:

The option indices are listed in the description of *LlSetOption()*. In addition, there are some new or (with regard to the function *LlSetOption()*) modified options:

LL_OPTION_LANGUAGE

Returns the currently selected language (See *LlJobOpen()* and *LlJobOpenLCID()*).

LL_OPTION_HELPAVAILABLE

LOWORD: See *LlSetOption()*
HIWORD: Checks whether the help file is present: TRUE: usable, FALSE: not usable (not present)

LL_OPTION_DEFPRINTERINSTALLED

Returns whether the operating system has a default printer.

Example:

```
HLLJOB    hJob;
UINT      nLanguage;

LlSetDebug(TRUE);
hJob = LlJobOpen(0);
// ....
nLanguage = LlGetOption(hJob, LL_OPTION_LANGUAGE);
// ....
LlJobClose(hJob);
```

See also:

LlSetOption

LlGetOptionString

Syntax:

```
INT LlGetOptionString (HLLJOB hJob, INT nMode, LPTSTR pszBuffer,
   UINT nBufSize);
```

Task:

Requests various string settings (see below) from List & Label.

Parameter:

hJob: List & Label job handle

nMode: Option index, see *LlSetOptionString()*

pszBuffer: Pointer to a buffer where the requested value will be stored.

nBufSize: Size of the buffer

Return Value:

The value of the corresponding option

Hints:

The option indices are listed in the description of *LlSetOptionString()*.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

Example:

```
HLLJOB    hJob;
TCHAR     szExt[128];

LlSetDebug(TRUE);
hJob = LlJobOpen(0);
// ....
LlGetOptionString(hJob, LL_OPTIONSTR_PRJEXT,
   szExt,sizeof(szExt));
// ....
LlJobClose(hJob);
```

See also:

LlSetOptionString

LlGetPrinterFromPrinterFile

Syntax:

```
INT LlGetPrinterFromPrinterFile (HLLJOB hJob, UINT nObjType,
   LPCTSTR pszObjName, INT nPrinter, LPTSTR pszPrinter,
   LLPUINT pnSizePrn, _PDEVMODE pDM, LLPUINT pnSizeDm);
```

Task:

Queries the printer configuration from the printer configuration file of List & Label.

Parameter:

hJob: List & Label job handle

nObjType: *LL_PROJECT_LABEL*, *LL_PROJECT_CARD* or *LL_PROJECT_LIST*

pszObjName: File name of the project with file extension

nPrinter: Index of the printer to be queried (0=first, 1=second) If you pass values starting from 100 (e.g. in a loop until you receive LL_ERR_PARAMETER as return value) you can query the printer for the various layout regions (corresponding to their order being set in the Designer via 'Project > Page Setup'). If the project contains only one printer, *nPrinter* must be -1.

pszPrinter: Address of buffer for printer name. If this pointer is NULL and pnSizePrn is not NULL, the necessary size of the buffer will be stored in *pnSizePrn.

pnSizePrn: Address of variable with buffer size (Size in characters, therefore the doubled size in Bytes must be reserved for the Unicode API).

pDM: Address of buffer for the DEVMODE structure. If this pointer is NULL and pnSizeDm non-NULL, the necessary size of the buffer will be stored in *pnSizeDm.

pnSizeDm: Address of variable with buffer size.

Return Value:

Error code

Hints:

The DEVMODE structure is defined and described in the Windows API.

Due to the possibility to define layout regions in the Designer the practical benefit of this function has been quite limited. We recommend using the LL object model according to chapter "Using the DOM-API (Professional/Enterprise Edition Only)" to access the layout regions and the associated printers.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlSetPrinterInPrinterFile

LlGetProjectParameter

Syntax:

```
INT LlGetProjectParameter(HLLJOB hJob, LPCTSTR lpszProjectName,
   LPCTSTR lpszParameter, LPTSTR lpszBuffer, UINT nBufSize);
```

Task:

Returns the value of the project parameter for the given project file. If the project parameter contains a formula, it is returned as is without being evaluated.

Parameter:

hJob: List & Label Job-Handle

lpszProjectName: Pointer to a string with the project name

lpszParameter: Pointer to a string with the parameter name

lpszBuffer Address of buffer for contents

nBufSize: Maximum number of characters to be copied

Return Value:

Error code

Example:

```
HLLJOB hJob;
TCHAR Buffer[1024];
hJob = LlJobOpen(0);

LlSetDefaultProjectParameter(hJob, "QueryString",
   "SELECT * FROM PRODUCTS", LL_PARAMETERFLAG_SAVEDEFAULT);
// call up designer
…

// then before print starts
LlGetProjectParameter(hJob, "c:\\repository\\report.lst",
"QueryString", Buffer, 1024);
<... etc ...>
LlJobClose(hJob);
```

Hints:

This API is especially useful if the project parameter is queried before printing to offer report parametrization to the user.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlPrintIsVariableUsed, LlPrintIsChartFieldUsed, LlPrintIsFieldUsed

LlGetSumVariableContents

Syntax:

```
INT LlGetSumVariableContents (HLLJOB hJob, LPCTSTR lpszName,
   LPTSTR lpszBuffer, UINT nBufSize);
```

Task:

Returns the contents of the corresponding sum variable.

Parameter:

hJob: Job handle

lpszName: Pointer to a string with the name of the sum variable.

lpszBuffer: Address of buffer for contents

nBufSize: Maximum number of characters to be copied

Return Value:

Error code (*LL_ERR_UNKNOWN_FIELD* or 0)

Hints:

This function can be used in callback routines to ask for the contents of sum variables.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlDefineSumVariable, LlGetUserVariableContents, LlGetVariableContents

LlGetUsedIdentifiers

Syntax:

```
INT LlGetUsedIdentifiers(HLLJOB hJob, LPCTSTR lpszProjectName,
   LPTSTR lpszBuffer, UINT nBufSize);
```

Task:

Returns a list of variables, fields and chart fields that are actually used within the given project file, in order to increase performance, as only these values need to be provided.

Parameter:

hJob: List & Label Job-Handle

lpszProjectName: Pointer to a string with the project name

lpszBuffer Address of buffer for contents

nBufSize: Maximum number of characters to be copied

Return Value:

Error code

Hints:

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlGetUsedIdentiefersEx

LlGetUsedIdentifiersEx

Syntax:

```
INT LlGetUsedIdentifiersEx(HLLJOB hJob, LPCTSTR lpszProjectName,
   UINT nIdentifierTypes, LPTSTR lpszBuffer, UINT nBufSize);
```

Task:

Returns a list of variables, fields and chart fields that are actually used within the given project file, in order to increase performance, as only these values need to be provided.

Parameter:

hJob: List & Label Job-Handle

lpszProjectName: Pointer to a string with the project name

nIdentifierTypes Identifier types that shall be considered. The values can be combined with OR:

Value	Meaning
LL_USEDIDENTIFIERSFLAG_VARIABLES	Variables
LL_USEDIDENTIFIERSFLAG_FIELDS	Fields
LL_USEDIDENTIFIERSFLAG_CHARTFIELDS	Chart fields
LL_USEDIDENTIFIERSFLAG_TABLES	Tables (see LIDbAddTable)
LL_USEDIDENTIFIERSFLAG_RELATIONS	Relations (see LIDbAddTableRelation)

lpszBuffer Address of buffer for contents

nBufSize: Maximum number of characters to be copied

Return Value:

Error code

Hints:

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlGetUsedIdentiefers

LlGetUserVariableContents

Syntax:

```
INT LlGetUserVariableContents (HLLJOB hJob, LPCTSTR lpszName,
   LPTSTR lpszBuffer, UINT nBufSize);
```

Task:

Returns the contents of the corresponding user variable.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to a string with the name of the sum variable.

lpszBuffer: Address of buffer for contents

nBufSize: Maximum number of characters to be copied

Return Value:

Error code (*LL_ERR_UNKNOWN_FIELD* or 0)

Hints:

This function can be used in callback routines to ask for the contents of user variables.

The variable type can be requested with *LlGetVariableType()* or *LlGetFieldType()*.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlGetSumVariableContents, LlGetVariableContents

LlGetVariableContents

Syntax:

```
INT LlGetVariableContents (HLLJOB hJob, LPCTSTR lpszName,
   LPTSTR lpszBuffer, UINT nBufSize);
```

Task:

Returns the contents of the corresponding variable.

Parameter:

hJob: Job handle

lpszName: Pointer to a string with the name of the variable

lpszBuffer: Address of buffer for contents

nBufSize: Maximum number of characters to be copied

Return Value:

Error code (*LL_ERR_UNKNOWN* or 0)

Hints:

This function can be used in callback routines to ask for the contents of variables.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlDefineVariableStart, LlDefineVariableExt, LlDefineVariableExtHandle, LlGetVariableType

LlGetVariableType

Syntax:

```
INT LlGetVariableType (HLLJOB hJob, LPCTSTR lpszName);
```

Task:

Returns the type of the corresponding variable.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to a string with the name of the variable

Return Value:

Variable type (positive), or error code (negative)

Hints:

This function can be used in callback routines to ask for the type of variables.

See also:

LlDefineVariableStart, LlDefineVariableExt, LlDefineVariableExtHandle, LlGetVariableContents

LlGetVersion

Syntax:

```
INT LlGetVersion(INT nCmd);
```

Task:

Returns the version number of List & Label.

Parameter:

Value	Meaning
LL_VERSION_MAJOR (1)	Returns the main version number, e.g. 26

LL_VERSION_MINOR (2)	Returns the minor version number, for example 1, (1 means 001 as the sub-version has three digits).

Return Value:

See parameter

Example:

```
int   v;
v = LlGetVersion(VERSION_MAJOR);
```

LlJobClose

Syntax:

```
void LlJobClose (HLLJOB hJob);
```

Task:

Releases the internal variables, frees resources etc.

Parameter:

hJob: List & Label job handle

Hints:

This function should be called at the end (coupled with *LlJobOpen()* or *LlJobOpenLCID()*), i.e. after using the List & Label DLL or when ending your program.

Example:

```
HLLJOB   hJob;

hJob = LlJobOpen(1);
LlDefineVariableStart(hJob);
<... etc ...>
LlJobClose(hJob);
```

See also:

LlJobOpen, LlJobOpenLCID

LlJobOpen

Syntax:

```
HLLJOB LlJobOpen (INT nLanguage);
```

Task:

Initializes internal variables and resources of the DLL for a calling program. Almost all DLL commands require the return value of this function as the first parameter.

Parameter:

nLanguage: Chosen language for user interactions (dialogs)

Value	Meaning
CMBTLANG_DEFAULT	Default language (use settings in Windows) For other languages see header file declarations
CMBTLANG_GERMAN	German
CMBTLANG_ENGLISH	English

Further constants can be found in your declaration file.

If this parameter is "or"ed with the constant *LL_JOBOPENFLAG_NOLLXPRE-LOAD,* no List & Label extensions will be preloaded.

Return Value:

A handle which is required as a parameter for most functions in order to have access to application-specific data.

A valid value is greater than 0. If the value is less than 0, it shows an errorcode. Please see chapters "General Notes About the Return Value" and "Error Codes" for further details.

Hints:

For ease of maintenance, we suggest putting global settings in one place, immediately after the *LlJobOpen()* call (dialog design, callback modes, expression mode, ...).

The C?LL26.DLL requires the language-dependent components which are stored in a separate DLL, e.g. C?LL2600.LNG or C?LL2601.LNG. They are loaded depending on the language setting.

If List & Label is no longer required, then the job should be released with the function *LlJobClose()* to give the DLL a chance to release the internal variables for this job.

Many additional language kits are available. Ask our sales department for further information or have a look at our web site at www.combit.com. The language IDs appended to the file name are a hex representation of the *CMBTLANG_xxxx* language codes found in the header (*.H, *.BAS, *.PAS, ...) file.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(CMBTLANG_ENGLISH);
LlDefineVariableStart(hJob);
LlDefineVariable(hJob, "Name", "Smith");
LlDefineVariable(hJob, "forename", "George");
```

```
<... etc ...>
LlJobClose(hJob);
```

See also:

LlJobOpenLCID, LlJobClose, LlSetOption, LlDesignerProhibitAction, LlSetFileExtensions

LlJobOpenLCID

Syntax:

```
HLLJOB LlJobOpenLCID (_LCID nLCID);
```

Task:

See *LlJobOpen()*.

Parameter:

nLCID: Windows locale ID for user interactions (dialogs)

Return Value:

See LlJobOpen().

Hints:

Calls *LlJobOpen()* with the respective *CMBTLANG_...* value.

Example:

```
HLLJOB hJob;
hJob = LlJobOpenLCID(LOCALE_USER_DEFAULT);
LlDefineVariableStart(hJob);
LlDefineVariable(hJob, "Name", "Smith");
LlDefineVariable(hJob, "forename", "George");
<... etc ...>
LlJobClose(hJob);
```

See also:

LlJobOpen, LlJobClose, LlSetOption, LlDesignerProhibitAction, LlSetFileExtensions

LlJobStateRestore

Syntax:

```
INT LlJobStateRestore(HLLJOB hLlJob, _PISTREAM pStream,UINT nFlags);
```

Task:

This API is used e.g. by the .NET DesignerControl to restore the state of a job previously saved on one machine on a different other machine. Depending on the flags passed, the variables, fields and database structure are deserialized from the stream. Thus, a following call to *LlDefineLayout()* offers the structures read from the stream in the Designer.

Parameter:

hJob: List & Label job handle

pStream: Stream that was created by a preceeding call to *LlJobStateSave()*. The stream format is proprietary and may change anytime without notice.

nFlags: Combination of LL_JOBSTATEFLAG_... values. They determine which values are read from the stream (variables, fields, chart fields, database structure, dictionaries, other job settings). To deserialize all available information, use LL_JOBSTATEFLAG_ALL.

Return Value:

See LlJobOpen().

See also:

LlJobStateSave

LlJobStateSave

Syntax:

```
INT LlJobStateRestore(HLLJOB hLlJob, _PISTREAM pStream,UINT nFlags,
 bool bPacked);
```

Task:

This API is used e.g. by the .NET DesignerControl to save the state of a job. Depending on the flags passed, the variables, fields and database structure are serialized to the stream.

Parameter:

hJob: List & Label job handle

pStream: Serialization stream. The stream format is proprietary and may change anytime without notice.

nFlags: Combination of LL_JOBSTATEFLAG_... values. They determine which values are written to the stream (variables, fields, chart fields, database structure, dictionaries, other job settings). To serialize all available information, use LL_JOBSTATEFLAG_ALL.

bPacked: Determines if the stream content should be packed.

Return Value:

See LlJobOpen().

See also:

LlJobStateRestore

LlLocAddDesignLCID

Syntax:

```
INT LlLocAddDesignLCID(HLLJOB hJob, LCID nLCID);
```

Task:

Adds a localization language to the project. For all added languages translations can be provided via *LlLocAddDictionaryEntry()*.

Parameter:

hJob: List & Label job handle

nLCID: Windows locale ID. The locale ID passed with the first call will be considered as the base language. All translations that will be provided via *LlLocAddDictionaryEntry()* need to use this language for the dictionary keys. If nLCID is 0, all languages will be removed from the list of localization locale IDs.

Return Value:

Error code

See also:

LlLocAddDictionaryEntry

LlLocAddDictionaryEntry

Syntax:

```
INT LlLocAddDictionaryEntry(HLLJOB hJob, LCID nLCID, LPCTSTR pszKey,
   LPCTSTR pszValue, UINT nType);
```

Task:

Adds a translation pair to one of the dictionaries. The dictionaries allow the localisation of project resp. Designer items.

Parameter:

hJob: List & Label job handle

nLCID: Windows locale ID specifing the dictionary to which the translation shall be added. This dictionary must already have been declared via *LlLocAddDesignLCID()*.

pszKey: Key for the dictionary (original text in the base language).

pszValue: Translated text for the dictionary.

nType: Dictionary type.

Value	Meaning
LL_DICTIONARY_TYPE_STATIC	Static (fixed) text

Value	Meaning
LL_DICTIONARY_TYPE_IDENTIFIER	Name of field or variable
LL_DICTIONARY_TYPE_TABLE	Table name
LL_DICTIONARY_TYPE_RELATION	Relation name
LL_DICTIONARY_TYPE_SORTORDER	Sortorder name

Return Value:

Error code

Hints:

Use this function to use the same project definition file for various localizations. After having added languages via *LlLocAddDictionaryEntry()* the Designer toolbar will offer a button for choosing the language. LL_DICTIONARY_TYPE_-STATIC allows the localization of static text by using the Translate$ Designer function including Intellisense support. The static text may contain up to three placeholders which are marked as {0}, {1} and {2}.

At print time the used language will be automatically set according to the thread locale ID (which is the system language by default). If you want to set a specific language as the default, use LL_OPTION_LCID. This default setting can be overruled by the end-user via the Designer.

For clean up purposes, set pszKey and pszValue to NULL and nType to 0. This will delete all dictionary entries from all dictionary types.

Example:

```
HLLJOB hJob;

hJob = LlJobOpen(CMBTLANG_DEFAULT);

// Add languages
LlLocAddDesignLCID(hJob, 9); // English as base language
LlLocAddDesignLCID(hJob, 7); // German as translation language

// Add translations
LlLocAddDictionaryEntry(hJob, 7, "ArticleNumber", "Artikelnummer",
         LL_DICTIONARY_TYPE_IDENTIFIER);
LlLocAddDictionaryEntry(hJob, 7, "Price", "Preis",
         LL_DICTIONARY_TYPE_IDENTIFIER);
LlLocAddDictionaryEntry(hJob, 7, "Page {0} of {1}", "Seite {0} von
{1}",
         LL_DICTIONARY_TYPE_STATIC);

LlDefineVariableStart(hJob);
LlDefineVariable(hJob, "ArticleNumber", "12345");
LlDefineVariable(hJob, "Price", "123");

// Invoke Designer etc.
```

```
...
LlJobClose(hJob);
```

See also:

LlLocAddDesignLCID, LL_OPTION_LCID

LlPreviewDeleteFiles

Syntax:

```
INT LlPreviewDeleteFiles (HLLJOB hJob, LPCTSTR lpszObjName,
   LPCTSTR lpszPath);
```

Task:

Deletes the temporary file(s) which have been created by the preview print.

Parameter:

hJob: List & Label job handle

lpszObjName: Valid file name with extension and without path

lpszPath: Valid path of the preview files ending with a backslash "\".

Return Value:

Error code

Hints:

Should always be called after *LlPreviewDisplay()*, as the preview files are generally only valid momentarily.

Of course, if you want to archive, send or print them at a later time, this should NOT be called.

See also:

LlPrintStart, LlPrintWithBoxStart, LlPreviewDisplay, LlPrintEnd, LlPreviewSetTempPath

LlPreviewDisplay

Syntax:

```
INT LlPreviewDisplay (HLLJOB hJob, LPCTSTR lpszObjName,
   LPCTSTR lpszPath, HWND hWnd);
```

Task:

Starts the preview window.

Parameter:

hJob: List & Label job handle

lpszObjName: Valid file name with file extension and without path name

lpszPath: Valid path of the preview files ending with a backslash "\".

hWnd: Window handle of the calling program

Return Value:

Error code

Hints:

The preview is a window that can be started independently of the Designer and shows the data that has been printed by the preview print process.

LlPreviewDisplay() calls *LlPreviewDisplayEx()* with *LL_PRVOPT_PRN_ASK-PRINTERIFNEEDED*.

See also:

LlPrintStart, LlPrintWithBoxStart, LlPreviewDeleteFiles, LlPrintEnd, LlPreviewSetTempPath, LlPreviewDisplayEx

LlPreviewDisplayEx

Syntax:

```
INT LlPreviewDisplayEx (HLLJOB hJob, LPCTSTR lpszObjName,
   LPCTSTR lpszPath, HWND hWnd, UINT nOptions, LPVOID pOptions);
```

Task:

Starts the preview. Additional options can define the behavior.

Parameter:

hJob: List & Label job handle

lpszObjName: Valid file name with file extension and without path name

lpszPath: Valid path of the preview files ending with a backslash "\".

hWnd: Window handle of the calling program

nOptions:

Value	Meaning
LL_PRVOPT_PRN_-USEDEFAULT	Preview uses the system's default printer
LL_PRVOPT_PRN_-ASKPRINTERIFNEEDED	If the printer that is stored in the preview file (i.e. the printer that has been used for the preview print process) is not found in the current computer's printers, a printer dialog is shown so that the user can select the printer.
LL_PRVOPT_PRN_-ASKPRINTERALWAYS	A printer dialog will allow the user to choose his default printer for the preview.

pOptions: Reserved, set to NULL or "".

Return Value:

Error code

Hints:

The preview is a window that can be started independently of the Designer. It shows the data printed by the preview print process.

If lpszPath is empty, the path of the project file is used.

See also:

LlPrintStart, LlPrintWithBoxStart, LlPreviewDeleteFiles, LlPreviewSetTempPath, LlPreviewDisplay

LlPreviewSetTempPath

Syntax:

```
INT LlPreviewSetTempPath (HLLJOB hJob, LPCTSTR lpszPath);
```

Task:

Sets a temporary path for the print preview file(s). Especially useful for applications running in a network environment.

Parameter:

hJob: List & Label job handle

lpszPath: Valid path with a concluding backslash "\"

Return Value:

Error code

Hints:

The preview file(s) will be stored in this path. The file name is the project's name, the file extension is ".LL". The preview file can be archived, sent or viewed whenever needed.

If the path is NULL or "", the path in which the project file is stored is taken.

This command must be called before the first call to *LlPrint()* in the print loop.

See also:

LlPrintStart, LlPrintWithBoxStart

LlPrint

Syntax:

```
INT LlPrint (HLLJOB hJob);
```

Task:

Output of all objects on the printer.

Parameter:

hJob: List & Label job handle

Return Value:

Error code

Hints:

Normal objects and the header of a table object (see option *LL_OPTION_DELAY-TABLEHEADER*) are printed. A table object has to be filled with calls of *LlPrintFields()* afterwards. LlPrint is responsible for a page break.

Label/card projects: As long as *LlPrint()* returns *LL_WRN_REPEAT_DATA, LlPrint()* must be called again, so objects that have caused a page break must be printed again on the next label /page.

This function is described explicitly in the chapter "Further Programming Basics"

See also:

LlPrintFields, LlPrintEnableObject

LlPrintAbort

Syntax:

```
INT LlPrintAbort (HLLJOB hJob);
```

Task:

Aborts the print (an incomplete page will remain incomplete or may not be printed).

Parameter:

hJob: List & Label job handle

Return Value:

Error code

Hints:

Is necessary to abort the print by code if *LlPrintWithBoxStart()* is not used and print abortion is necessary.

The difference to the 'normal' end, i.e. no longer having to call *LlPrint()* or *LlPrintFields()* is that data which is still in the printer driver is discarded, so that the print may be ended halfway through a page.

The *LlPrint...()* calls following this call will return *LL_USER_ABORTED*, so your print loop will be ended automatically.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

if (LlPrintStart(hJob, LL_PROJECT_LABEL, "test.lbl",
   LL_PRINT_NORMAL) == 0)
{
   for all data records
   {
      <... etc...>
      if (bDataError)
         LlPrintAbort(hJob);
   }
   LlPrintEnd(hJob);
}
else
   MessageBox(NULL, "error", "List & Label", MB_OK);
LlJobClose(hJob);
```

See also:

LlPrintStart, LlPrintWithBoxStart, LlPrintEnd

LlPrintCopyPrinterConfiguration

Syntax:

```
INT LlPrintCopyPrinterConfiguration (HLLJOB hJob,
   LPCTSTR lpszFilename, INT nFunction);
```

Task:

Allows saving and restoration of the printer configuration file.

Parameter:

hJob: List & Label job handle

lpszFilename: File name of the printer configuration file with file extension

nFunction: Action

Action	Meaning
LL_PRINTERCONFIG_SAVE	Saves the printer configuration file of the currently opened project in a file with the name lpszFilename.

LL_PRINTERCONFIG_-RESTORE	Copies the previously saved configuration file (created with *LL_PRINTERCONFIG_SAVE*) back to the current project.

Return Value:

Error code (always 0)

Hints:

It is important that *LL_PRINTERCONFIG_RESTORE* is called before(!) *LlPrint()*!

Example:

The following principle should be used for hand-made copies on a temporary printer, that is, a user can choose to temporarily change the printer using the printer dialog box, and choose multiple copies. Usually the second and following passes would print to the default printer, which is not intended.

```
for each copy
{
   LlPrintWithBoxStart(...)
   if (first copy)
   {
      LlPrintOptionsDialog(...);
      LlPrintCopyPrinterConfiguration("curcfg.~~~",
         LL_PRINTERCONFIG_SAVE);
   }
   else
   {
       LlPrintCopyPrinterConfiguration("curcfg.~~~",
         LL_PRINTERCONFIG_RESTORE);
   }
   .. LlPrint(), LlPrintFields(), ...
}
```

See also:

LlPrintStart, LlPrintWithBoxStart, LlSetPrinterToDefault, LlSetPrinterInPrinterFile, LlGetPrinterFromPrinterFile, LlSetPrinterDefaultsDir

LlPrintDbGetRootTableCount

Syntax:

```
INT LlPrintDbGetRootTableCount(HLLJOB hJob);
```

Task:

Returns the number of tables at the root level. Necessary to correctly display a progress bar.

Parameter:

hJob: List & Label job handle

Return Value:

Number of tables

Hints:

See the hints in chapter "Printing Relational Data".

See also:

LIDbAddTable, LIDbAddTableRelation, LIDbAddTableSortOrder, LIPrintDbGetCurrentTable, LIPrintDbGetCurrentTableSortOrder, LIPrintDbGetCurrentTableRelation

LlPrintDbGetCurrentTable

Syntax:

```
INT LlPrintDbGetCurrentTable(HLLJOB hJob, LPTSTR pszTableID,
   UINT nTableIDLength, BOOL bCompletePath);
```

Task:

This function returns the table that is currently printed/filled.

Parameter:

hJob: List & Label job handle

pszTableID: Buffer in which the string is to be stored

nTableIDLength: Size of buffer.

bCompletePath: If true, the complete table hierarchy will be returned, e.g. "Orders > OrderDetails". If false, only the table name (e.g. "OrderDetails") is returned.

Return Value:

Error code

Hints:

See the hints in chapter "Printing Relational Data".

See also:

LIDbAddTable, LIDbAddTableRelation, LIDbAddTableSortOrder, LIPrintDbGetCurrentTableSortOrder, LIPrintDbGetCurrentTableRelation

LlPrintDbGetCurrentTableFilter

Syntax:

```
INT LlPrintDbGetCurrentTableFilter(HLLJOB hJob,
   PVARIANT pvFilter, PVARIANT pvParams);
```

Task:

This function returns the current table filter in data source native syntax. The translation has to be performed in the *LL_QUERY_EXPR2HOSTEXPRESSION* callback. This callback is triggered for each part of the filter expression that is used in the Designer.

Parameter:

hJob: List & Label job handle

pvFilter: This parameter receives the translated filter expression. As usual for VARIANTs, it must be initialized before (VariantInit()) and freed after use (VariantClear()).

pvParams: If the filter expression uses parameters (see callback documentation), this argument receives a VARIANTARRAY with the parameter values. As usual for VARIANTs, it must be initialized before (VariantInit()) and and freed after use (VariantClear()).

Return Value:

Error code

See also:

LIDbAddTable, LL_QUERY_EXPR2HOSTEXPRESSION

LlPrintDbGetCurrentTableRelation

Syntax:

```
INT LlPrintDbGetCurrentTableRelation(HLLJOB hJob,
   LPTSTR pszRelationID, UINT nRelationIDLength);
```

Task:

Queries the ID of the current subrelation to be printed. The ID can also be empty if a subtable has been inserted in the Designer using a filter. In this case your code should ideally pre-filter the table at database level (faster) or alternatively iterate the entire table and leave the filtering to List & Label (much slower).

Parameter:

hJob: List & Label job handle

pszRelationID: Buffer in which the string is to be stored.

nRelationIDLength: Size of buffer.

Return Value:

Error code

Hints:

See the hints in chapter "Printing Relational Data".

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LIDbAddTable, LIDbAddTableRelation, LIDbAddTableSortOrder, LIPrintDbGetCurrentTable, LIPrintDbGetCurrentTableSortOrder

LlPrintDbGetCurrentTableSortOrder

Syntax:

```
INT LlPrintDbGetCurrentTableSortOrder(HLLJOB hJob,
   LPTSTR pszSortOrderID, UINT nSortOrderIDLength);
```

Task:

This function returns the current table sort order to be printed. If multiple (stacked) sortings are supported (see *LIDbAddTableEx()*), a tab separated list is returned.

Parameter:

hJob: List & Label job handle

pszSortOrderID: Buffer in which the string is to be stored.

nSortOrderIDLength: Size of buffer.

Return Value:

Error code

Hints:

See the hints in chapter "Printing Relational Data".

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LIDbAddTable, LIDbAddTableRelation, LIDbAddTableSortOrder, LIPrintDbGetCurrentTable, LIPrintDbGetCurrentTableRelation

LlPrintDeclareChartRow

Syntax:

```
INT LlPrintDeclareChartRow (HLLJOB hJob, UINT nFlags);
```

Task:

This function is used to inform the chart objects contained in the project that data is available.

Parameter:

hJob: List & Label job handle

nFlags: Specifies the chart type for which data is available

Return Value:

Error code

Hints:

The following flags may be used with this function:

LL_DECLARECHARTROW_FOR_OBJECTS: informs chart objects that data is available.

LL_DECLARECHARTROW_FOR_TABLECOLUMNS: informs chart objects contained in table columns that data is available.

Please note the hints in the chart chapter of this manual.

This call does not actually print the objects, but only tells them to store the current data. Only a call to *LlPrint()* (chart objects) or *LlPrintFields()* (charts in table columns) actually prints the charts.

Example:

```
// while data to put into chart object...
   ... LlDefineChartFieldExt(...);
   LlPrintDeclareChartRow(hJob, LL_DECLARECHARTROW_FOR_OBJECTS);
// now print chart object
ret = LlPrint();
```

See also:

LlDefineChartFieldExt, LlDefineChartFieldStart

LlPrintDidMatchFilter

Syntax:

```
INT LlPrintDidMatchFilter (HLLJOB hJob);
```

Task:

Specifies whether the last data record printed matched the filter provided by the user, i.e. if it was really printed.

Parameter:

hJob: List & List & Label job handle

Return Value:

<0: Error code; 0: not printed; 1: printed

Hints:

This function can only be called after *LlPrint()* / *LlPrintFields()*.

Example:

```
ret = LlPrint();
if (ret == 0 && LlPrintDidMatchFilter(hJob))
   ++ nCountOfPrintedRecords;
```

See also:

LlPrintGetFilterExpression, LlPrintWillMatchFilter, LL_NTFY_FAILS_FILTER-Callback

LlPrintEnableObject{XE LlPrintEnableObject"}

Syntax:

```
INT LlPrintEnableObject(HLLJOB hJob, LPCTSTR lpszObject,
   BOOL bEnable);
```

Task:

Enables the object to be printed or disables it in order to tell List & Label to ignore it.

Parameter:

hJob: List & Label job handle

lpszObject: Object name, see below

bEnable: TRUE: Object can be printed; FALSE: Object should be ignored

Return Value:

Error code (important!)

Hints:

The object name can be " (empty) to address all objects, otherwise it must be the object name (entered by the user) with the prefix ':'.

If the user is able to change objects and object names in the Designer, it is important to ask for the return value to test whether the object exists at all!

This function is particularly important for filling several independent tables. Before calling *LlPrint(),* all table objects must be enabled.

Example:

```
LlPrintEnableObject(hJob, "", TRUE);
LlPrintEnableObject(hJob, ":AuthorList", FALSE);
```

See also:

LlPrint, LlPrintFields

LlPrintEnd

Syntax:

```
INT LlPrintEnd (HLLJOB hJob, INT nPages);
```

Task:

Ends the print job.

Parameter:

hJob: List & Label job handle

nPages: Number of empty pages desired after the print

Return Value:

Error code

Hints:

The behavior is described in the programming part of this manual.

Please always use *LlPrintEnd()* if you have used *LlPrintStart()* or *LlPrintWithBoxStart()* and these commands were not aborted with an error, otherwise resource and memory losses may result.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

if (LlPrintStart(hJob, LL_PROJECT_LABEL, "test.lbl", LL_PRINT_NORMAL)
== 0)
{
   <... etc...>
   LlPrintEnd(hJob, 0);
}
else
   MessageBox(NULL, "error", "List & Label", MB_OK);
LlJobClose(hJob);
```

See also:

LlPrintStart, LlPrintWithBoxStart, LlPrintFieldsEnd

LlPrinterSetup

Syntax:

```
INT LlPrinterSetup (HLLJOB hJob, HWND hWnd, UINT nObjType,
   LPCTSTR lpszObjName);
```

Task:

Opens a printer selection window and saves the user's selection in the printer definition file.

Parameter:

hJob: List & Label job handle

hWnd: Window handle of the calling program

nObjType:

Value	Meaning
LL_PROJECT_LABEL	for labels
LL_PROJECT_CARD	for cards
LL_PROJECT_LIST	for lists

lpszObjName: Valid project file name with path and file extension

Return Value:

Error code

Hints:

Must be called before *LlPrint(WithBox)Start()*. Allows printer selection without having to perform a printing process (as you must do when using *LlPrintOptionsDialog()*).

We do not recommend this function, *LlPrintOptionsDialog()* is much more flexible.

See also:

LlPrintStart, LlPrintWithBoxStart, LlPrintOptionsDialog, LlPrintGetPrinterInfo, LlSetPrinterInPrinterFile

LlPrintFields

Syntax:

```
INT LlPrintFields (HLLJOB hJob);
```

Task:

Output of a table line.

Parameter:

hJob: List & Label job handle

Return Value:

Error code or command

Hints:

LlPrintFields() prints a data line in all non-hidden tables if the line matches the filter condition.

With the return value *LL_WRN_REPEAT_DATA,* List & Label informs you that you have to start a new page for the entry. With the corresponding *LlPrint()* on the next page the record pointer should not be moved to the next record.

If more tables are added via *LlDbAddTable()* the return value also can be *LL_WRN_TABLECHANGE.* Please refer to chapter "Printing Relational Data" for further information.

The exact behavior is described in the programming part of this manual.

See also:

LlPrint, LlPrintEnableObject

LlPrintFieldsEnd

Syntax:

```
INT LlPrintFieldsEnd (HLLJOB hJob);
```

Task:

Prints (tries to print) the footer on the last page and appended objects.

Parameter:

hJob: List & Label job handle

Return Value:

Error code

Hints:

Only needed in list projects.

Is necessary to make sure that the footer is printed, even if no other normal field data is available.

If the return value is *LL_WRN_REPEAT_DATA*, the footer could not be printed on the (last) page. *LlPrintFieldsEnd()* might have to be called multiple times to print the footer on another page.

If more tables are added via *LlDbAddTable()* the return value also can be *LL_WRN_TABLECHANGE.* Please refer to chapter "Printing Relational Data" for further information.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

if (LlPrintStart(hJob, LL_PROJECT_LIST, "test.lst",
   LL_PRINT_NORMAL) == 0)
{
   <... etc...>
```

```
    <data finished>
    while (LlPrintFieldsEnd(hJob) == LL_WRN_REPEAT_DATA)
    {
        <define variables for next page>
        // allow user to abort
        LlPrintUpdateBox(hJob)
    }

    LlPrintEnd(hJob, 0);
}
else
    MessageBox(NULL, "Error", "List & Label", MB_OK);
LlJobClose(hJob);
```

See also:

LlPrintEnd

LlPrintGetChartObjectCount

Syntax:

```
INT LlPrintGetChartObjectCount(HLLJOB hJob, UINT nType);
```

Task:

Returns the number of chart objects in the current project.

Parameter:

hJob: List & Label job handle

nType: Location of chart

Return Value:

Error code or number of charts

Hints:

nType must be one of the following:

LL_GETCHARTOBJECTCOUNT_CHARTOBJECTS: Returns the number of chart objects, excluding those placed in table columns.

LL_GETCHARTOBJECTCOUNT_CHARTOBJECTS_BEFORE_TABLE: Returns the number of chart objects before the table in the print order.

LL_GETCHARTOBJECTCOUNT_CHARTCOLUMNS: Returns the number of chart columns.

This function can be used to optimize the printing loop. More hints can be found in the chart chapter of this manual.

See also:

LlPrint

LlPrintGetCurrentPage

Syntax:

```
INT LlPrintGetCurrentPage (HLLJOB hJob);
```

Task:

Returns the page number of the page currently printing.

Parameter:

hJob: List & Label job handle

Return Value:

<0: Error code
>=0: page number

Hints:

This function can only be used if a print job is open, i.e. after *LlPrint[WithBox]-Start()*.

It is the same page number that is returned by the *Page()* function in the Designer, or by *LlPrintGetOption(hJob, LL_PRNOPT_PAGE)*;

See also:

LlPrint

LlPrintGetFilterExpression

Syntax:

```
INT LlPrintGetFilterExpression (HLLJOB hJob, LPTSTR lpszBuffer,
   INT nBufSize);
```

Task:

Gets the chosen filter condition (if the project has assigned one).

Parameter:

hJob: List & Label job handle

lpszBuffer: Buffer in which the string is to be stored

nBufSize: Size of the buffer

Return Value:

Error code

Hints:

This function can only be used if a print job is open, i.e. after *LlPrint[WithBox]-Start()*.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlPrintWillMatchFilter, LlPrintDidMatchFilter

LlPrintGetItemsPerPage

Syntax:

```
INT LlPrintGetItemsPerPage (HLLJOB hJob);
```

Task:

Returns the number of labels on a page (no. of columns * no. of lines).

Parameter:

hJob: List & Label job handle

Return Value:

<0: error code
>=0: number of labels

Hints:

1 is always returned for *LL_PROJECT_LIST*.

Can be used to calculate the total number of output pages. See hints in the programming part of this manual.

See also:

LlPrintGetItemsPerTable

LlPrintGetOption

Syntax:

```
INT LlPrintGetOption (HLLJOB hJob, INT nIndex);
```

Task:

Returns the various print options which are set by the user in the *LlPrintOptionsDialog()*.

Parameter:

hJob: List & Label job handle

nIndex: One of the values listed below

Return Value:

Setting chosen by the user

Hints:

In addition to the constants listed for *LlPrintSetOption()*, there are several modified or additional (read-only) settings:

LL_PRNOPT_COPIES_SUPPORTED

Returns a flag indicating whether the currently selected copies count is supported by the printer (this is usually important for list projects only).

Important: This function will - if successful - set the printer copies in the driver!

If it is not successful, the *LL_PRNOPT_COPIES* must be set to 1 and the copies must be made manually, if needed.

LL_PRNOPT_UNIT

This option returns the measurement units set in the system settings. Returned values are one of the following constants:

Value	Meaning
LL_UNITS_MM_DIV_10	1/10 mm
LL_UNITS_MM_DIV_100	1/100 mm (default on metric systems)
LL_UNITS_MM_DIV_1000	1/1000 mm
LL_UNITS_INCH_DIV_100	1/100 inch
LL_UNITS_INCH_DIV_1000	1/1000 inch (default on imperial systems)
LL_UNITS_SYSDEFAULT_LORES	Default low resolution on the system
LL_UNITS_SYSDEFAULT_HIRES	Default high resolution on the system
LL_UNITS_SYSDEFAULT	Default resolution on the system

LL_PRNOPT_USE2PASS

Returns if the printing process uses the two pass method, because the *TotalPages$()* function has been used.

LL_PRNOPT_PRINTORDER

Returns the print order of the (labels/file cards) in the project. Default: *LL_PRINTORDER_HORZ_LTRB* (horizontal, from top left to bottom right)

LL_PRNOPT_DEFPRINTERINSTALLED

Returns a flag indicating whether the operating system has a default printer

LL_PRNOPT_JOBID

Use this option after *LlPrint()* to determine the job number of the print job from the spooler.

This ID can be used with the Windows-API functions to control the execution of the print job.

See also:

LlPrintSetOption, LlPrintOptionsDialog, LlPrintGetOptionString

LlPrintGetOptionString

Syntax:

```
INT LlPrintGetOptionString (HLLJOB hJob, INT nIndex, LPTSTR pszBuffer,
   UINT nBufSize);
```

Task:

Returns various print option string settings.

Parameter:

hJob: List & Label job handle

nIndex: See *LlPrintSetOptionString()*

pszBuffer: Address of buffer for the string

nBufSize: Maximum number of characters to be copied

Return Value:

Error code

Hints:

See LlPrintSetOptionString

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlPrintSetOption

LlPrintGetPrinterInfo

Syntax:

```
INT LlPrintGetPrinterInfo (HLLJOB hJob, LPTSTR lpszPrn,
   UINT nPrnBufSize, LPTSTR lpszPort, UINT nPortBufSize);
```

Task:

Returns information about the target printer.

Parameter:

hJob: List & Label job handle

lpszPrn: Address of buffer for the printer name

nPrnBufSize: Length of the buffer lpszPrn

lpszPort: Address of buffer for the printer port

nPortBufSize: Length of the buffer lpszPort

Return Value:

Error code

Hints:

Examples for printer names are 'HP DeskJet 500' or 'NEC P6', for printer port 'LPT2:' or '\\server\printer1'

In case of an export, the printer contains the description of the exporter and the port is empty.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlPrintStart, LlPrintWithBoxStart

LlPrintGetProjectParameter

Syntax:

```
INT LlPrintGetProjectParameter(HLLJOB hLlJob, LPCTSTR pszParameter,
   BOOL bEvaluated, LPTSTR pszBuffer, INT nBufSize, _LPUINT pnFlags)
```

Task:

Returns the value of a project parameter

Parameter:

hJob: List & Label job handle

pszParameter: Parameter name. May be NULL (see hints)

pszBuffer: Address of buffer for contents. May be NULL (see hints)

bEvaluated: If the parameter is of the type LL_PARAMETERFLAG_FORMULA, this flag decides whether the parameter should be evaluated first.

nBufSize: Size of the buffer (in TCHARs)

Return Value:

Error code or required buffer size

Hints:

This function cannot be called before *LlPrint[WithBox]Start()*!

If pszParameter is NULL, a semicolon-separated list of all USER parameters is returned.

If pszBuffer is NULL, the return value equals the size of the required buffer (in TCHARS) including the termination.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlSetDefaultProjectParameter, LlGetDefaultProjectParameter, LlPrintGetProjectParameter

LlPrintIsChartFieldUsed

Syntax:

```
INT LlPrintIsChartFieldUsed (HLLJOB hJob, LPCTSTR lpszFieldName);
```

Task:

Specifies whether the given chart field is used in one of the expressions or conditions of the project.

Parameter:

hJob: List & Label job handle

lpszFieldName: Chart field name

Return Value:

Value	Meaning
1	Chart field is used
0	Chart field is not used
LL_ERR_UNKNOWN	Chart field is not defined

A valid value is greater than 0. If the value is less than 0, it shows an errorcode. Please see chapters "General Notes About the Return Value" and "Error Codes" for further details.

Hints:

This function can only be called after *LlPrintStart()* or *LlPrintWithBoxStart()*.

This function needs *LL_OPTION_NEWEXPRESSIONS* to be set to true (default).

As calling *LlDefineChartFieldStart()* clears the "used" flags, this function will return *LL_ERR_UNKNOWN* or 0 afterwards, regardless of whether the field is actually used or not. Therefore do not use *LlDefineChartFieldStart()* after *LlPrint[WithBox]Start()*.

Instead of using a specific field name, wildcards can be used. For further information see *LlPrintIsFieldUsed()*.

Example:

```
if (LlPrintIsChartFieldUsed(hJob, "Month") == 1)
   LlDefineChartFieldExt(hJob, "Month", <...>);
```

See also:

LlPrintStart, LlPrintWithBoxStart, LlPrintIsVariableUsed, LlPrintIsFieldUsed

LlPrintIsFieldUsed

Syntax:

```
INT LlPrintIsFieldUsed (HLLJOB hJob, LPCTSTR lpszFieldName);
```

Task:

Specifies whether the given field from the loaded project is used in one of the expressions or conditions of the project. To query the used fields even before starting a print job, the usage of *LlGetUsedIdentifiers* is preferable.

Parameter:

hJob: List & Label job handle

lpszFieldName: Field name

Return Value:

Value	Meaning
1	Field is used
0	Field is not used
LL_ERR_UNKNOWN	Field is not defined

A valid value is greater than 0. If the value is less than 0, it shows an errorcode. Please see chapters "General Notes About the Return Value" and "Error Codes" for further details.

Hints:

This function can only be called after *LlPrintStart()* or *LlPrintWithBoxStart()*.

This function needs *LL_OPTION_NEWEXPRESSIONS* to be set to true (default).

As calling *LlDefineFieldStart()* clears the "used" flags, this function will return *LL_ERR_UNKNOWN* or 0 afterwards, regardless of whether the field is actually used or not. Therefore do not use *LlDefineFieldStart()* after *LlPrint[WithBox]Start()*.

Instead of using a specific field name, wildcards can be used. This is especially useful if you pass your fields ordered hierarchically, e.g. all fields from the "Article" table use "Article." as prefix. Simply do a search for "Article.*" to find out whether the table has been used at all by the user.

Example:

```
if (LlPrintIsFieldUsed(hJob, "Name") == 1)
   LlDefineFieldExt(hJob, "Name", <...>);
```

See also:

LlPrintStart, LlPrintWithBoxStart, LlPrintIsVariableUsed

LlPrintIsVariableUsed

Syntax:

```
INT LlPrintIsVariableUsed (HLLJOB hJob, LPCTSTR lpszFieldName);
```

Task:

Specifies whether the given variable is used in one of the expressions or conditions of the project. Note the hints for *LlPrintIsFieldUsed*.

Parameter:

hJob: List & Label job handle

lpszFieldName: Field name

Return Value:

Value	Meaning
1	Variable is used
0	Variable is not used
LL_ERR_UNKNOWN	Variable is not defined

A valid value is greater than 0. If the value is less than 0, it shows an errorcode. Please see chapters "General Notes About the Return Value" and "Error Codes" for further details.

Hints:

This function can only be called after *LlPrintStart()* or *LlPrintWithBoxStart()*.

This function needs *LL_OPTION_NEWEXPRESSIONS* to be set to true (default).

As calling *LlDefineVariableStart()* clears the "used" flags, this function will return *LL_ERR_UNKNOWN* or 0 afterwards, regardless of whether the field is actually used or not. Therefore do not use *LlDefineVariableStart()* after *LlPrint[WithBox]Start()*.

Instead of using a specific variable name, wildcards can be used. For further information see *LlPrintIsFieldUsed()*.

Example:

```
if (LlPrintIsVariableUsed(hJob, "Name") == 1)
   LlDefineVariableExt(hJob, "Name", <...>);
```

See also:

LlPrintStart, LlPrintWithBoxStart, LlPrintIsFieldUsed

LlPrintOptionsDialog

Syntax:

```
INT LlPrintOptionsDialog (HLLJOB hJob, HWND hWnd, LPCTSTR lpszText);
```

Task:

Calls a print option selection window and enables the user to select print-specific settings.

Parameter:

hJob: List & Label job handle

hWnd: Window handle of the calling program

lpszText: Text to be passed in the dialog, e.g. 'Only 55 labels will be printed'

Return Value:

Error code

Hints:

This function is equivalent to *LlPrintOptionsDialogTitle()* with NULL as dialog title. See this section for further hints.

See also:

LlPrinterSetup, LlPrintSetOption, LlPrintGetOption, LlPrintOptionsDialogTitle

LlPrintOptionsDialogTitle

Syntax:

```
INT LlPrintOptionsDialogTitle (HLLJOB hJob, HWND hWnd,
   LPCTSTR lpszTitle, LPCTSTR lpszText);
```

Task:

Calls a print option selection window and enables the user to select print-specific settings.

Parameter:

hJob: List & Label job handle

hWnd: Window handle of the calling program

lpszTitle: Dialog title

lpszText: Text to be passed in the dialog, e.g. 'Only 55 labels will be printed'

Return Value:

Error code

Hints:

The following settings can be made:

Printer (or reference printer for export)
Export destination
Page number of the first page (if not hidden)
Number of copies required (if this has not been removed by *LlPrintSetOption()*)
Starting position with *LL_PROJECT_LABEL*, *LL_PROJECT_CARD*, if more than one label/file card per page exists
Print destination
Page range (print from ... to ...)

Default values can be defined with *LlPrintSetOption()*. This function must be called after *LlPrintStart()* / *LlPrintWithBoxStart()* but before calling *LlPrint()* for the first time.

The number of copies might have to be evaluated by the programmer as some printer drivers do not have the relevant function implemented. See programmer's hints in this manual.

The function *LlPrinterSetup(...)* allows you to call a print selection dialog without further settings.

See also:

LlPrinterSetup, LlPrintSetOption, LlPrintGetOption, LlPrintOptionsDialog

LlPrintResetProjectState

Syntax:

```
INT LlPrintResetProjectState (HLLJOB hJob);
```

Task:

Resets the print state of the whole project, so that printing starts as if *LlPrint(WithBox)Start()* has just been called.

Parameter:

hJob: List & Label Job handle

Return Value:

Error code

Hints:

This API resets the print state of the whole project (objects, page numbers, user and sum variables etc).

This function can be used for mail merge tasks.

Example:

```
<start print job>
<while letters have to be printed>
{
   <get record>
   <print one letter>
   <if no error>
      LlPrintResetProjectState(hJob)
   <get next record of the database>
   <advance to next record>
}
<end print job>
```

LlPrintSelectOffsetEx

Syntax:

```
INT LlPrintSelectOffsetEx (HLLJOB hJob, HWND hWnd);
```

Task:

Opens a dialog in which the user can choose the first label's position in the label array.

Parameter:

hJob: List & Label job handle

hWnd: Window handle of the calling application

Return Value:

Error code

Hints:

Not applicable for list projects!

Default values can be defined with *LlPrintSetOption()*. This function must be called after *LlPrintStart()* / *LlPrintWithBoxStart()* but before calling *LlPrint()* for the first time.

The offset can be set and read via *LL_PRNOPT_OFFSET*.

The dialog is the same as the one offered by the *LlPrintOptionsDialog[Title]()*.

The return value is in the range of 0 to (*MAX_X*MAX_Y*-1).

See also:

LlPrintOptionsDialog

LlPrintSetBoxText

Syntax:

```
INT LlPrintSetBoxText (HLLJOB hJob, LPCTSTR lpszText,
   INT nPercentage);
```

Task:

Sets text and meter percentage value in the abort dialog box.

Parameter:

hJob: List & Label job handle

lpszText: Text which should appear in the box

nPercentage: Progress percentage

Return Value:

Error code

Hints:

To make the text multi-line, line feeds ('\x0a') can be inserted.

Unchanged texts or NULL pointers are not re-drawn to avoid flickering, unchanged percentage values or '-1' are also ignored.

Example:

```
HLLJOB hJob;

hJob = LlJobOpen(0);

if (LlPrintWithBoxStart(hJob, LL_PROJECT_LABEL, "test.lbl",
LL_PRINT_NORMAL,
      LL_BOXTYPE_NORMALMETER, hWnd, "print") == 0)
{
   LlPrintSetBoxText(hJob, "starting...", 0);
   <... etc...>
   LlPrintEnd(hJob);
   LlPrintSetBoxText(hJob, "done", 100);
}
```

```
else
   MessageBox(NULL, "error", "List & Label", MB_OK);
LlJobClose(hJob);
```

See also:

LlPrintWithBoxStart, LlPrintUpdateBox, LlPrint

LlPrintSetOption

Syntax:

```
INT LlPrintSetOption (HLLJOB hJob, INT nIndex, INT nValue);
```

Task:

Sets various print options for the print job or the print options dialog, for example to preset the number of copies required.

Parameter:

hJob: List & Label job handle

nIndex:

LL_PRNOPT_COPIES

Number of copies to be preset in the print dialog box. A value of *LL_COPIES_HIDE* will hide the "copies" option. The task of supporting copies is described in the programming hints section.

Default: 1

LL_PRNOPT_FIRSTPAGE

First page of the page range that shall be printed. If "All" has been chosen, this is identical to *LL_PRNOPT_PAGE*.

Default: INT_MIN

LL_PRNOPT_JOBPAGES

Number of pages a print job should contain if you choose *LL_PRINT_MULTIJOB* in *LlPrint[WithBox]Start()*.

Default: INT_MAX

LL_PRNOPT_LASTPAGE

Page number of the last page to be printed.

Default: INT_MAX

LL_PRNOPT_OFFSET

Position of the first label in the label array. The position the number refers to is defined by the print order.

Default: 0

LL_PRNOPT_PAGE

Page number of the first page printed by List & Label. If this should not be selectable, *LL_PAGE_HIDE* is the value to be passed.

Default: 1

LL_PRNOPT_PRINTDLG_ALLOW_NUMBER_OF_FIRST_PAGE

In the print dialog, this option determines the page number that starts on the first printed page, e.g. if you already have a cover page or other pages with page numbers.

Default: 0

LL_PRNOPT_PRINTDLG_ONLYPRINTERCOPIES

The print options dialog will only allow a copies value to be entered if the printer supports copies.

Caution: the printer copies may not be useful for labels, as these may need programmer's copies support instead of the printer's. See chapter "Copies".

Default: FALSE

LL_PRNOPT_UNITS

Returns the same value as *LlGetOption(..., LL_OPTION_UNITS).*

nValue: Sets the option corresponding to the nIndex

Return Value:

Error code

See also:

LlPrintStart, LlPrintWithBoxStart, LlPrintGetOption, LlPrintOptionsDialog

LlPrintSetOptionString

Syntax:

```
INT LlPrintSetOptionString (HLLJOB hJob, INT nIndex, LPCTSTR
pszValue);
```

Task:

Sets various print options for List & Label.

Parameter:

hJob: List & Label job handle

nIndex: See below

pszValue: The new value

Return Value:

Error code

Hints:

Values for nIndex:

LL_PRNOPTSTR_EXPORT

Sets the default export destination (for example "RTF", "HTML", "PDF", etc.) to be used (or shown in the print dialog)

LL_PRNOPTSTR_ ISSUERANGES

A string containing default settings for the issue range, for example "1,3-4,10-".

LL_PRNOPTSTR_ PAGERANGES

A string containing default settings for the range(s) like shown in the printer options dialog, for example "1,3-4,10-". Further variations are possible, e.g. "1,3,…" for uneven pages or "2,4,…" for every second page. When using "…" the pattern will be automatically continued accordingly.

LL_PRNOPTSTR_PRINTDST_FILENAME

The default file name that the print should be saved to if "print to file" has been chosen.

LL_PRNOPTSTR_PRINTJOBNAME

You can set the job name to be used for the print spooler with this option.

You need to set it before the print job starts (that is, before the first call to *LlPrint()*).

Example:

```
HLLJOB hJob;

hJob = LlJobOpen(0);
// LlPrintStart(...);
LlPrintSetOptionString(hJob, LL_PRNOPTSTR_PRINTDST_FILENAME,
"c:\temp\ll.prn");
// ....
// LlPrintEnd();
LlJobClose(hJob);
```

See also:

`LlPrintGetOptionString`

LlPrintSetProjectParameter

Syntax:

```
INT LlPrintSetProjectParameter(HLLJOB hLlJob, LPCTSTR pszParameter,
   LPCTSTR pszValue, UINT nFlags)
```

Task:

Changes the value of a project parameter (see chapter "Project Parameters")

Parameter:

hJob: List & Label job handle

pszParameter: Parameter name

pszValue: Parameter value

nFlags: Parameter type (see *LlSetDefaultProjectParameter()*). Will only be used for new parameters.

Return value:

Error code

Hints:

This function cannot be called before *LlPrint[WithBox]Start()*!

See also:

LlSetDefaultProjectParameter, LlGetDefaultProjectParameter, LlPrintGetProjectParameter

LlPrintStart

Syntax:

```
INT LlPrintStart (HLLJOB hJob, UINT nObjType, LPCTSTR lpszObjName,
   INT nPrintOptions, INT nReserved);
```

Task:

Starts the print job, loads the project definition.

Parameter:

hJob: List & Label job handle

nObjType: *LL_PROJECT_LABEL*, *LL_PROJECT_LIST* or *LL_PROJECT_CARD*

lpszObjName: The file name of the project with file extension

nPrintOptions: Print options

Value	Meaning
LL_PRINT_NORMAL	output to printer
LL_PRINT_PREVIEW	output to preview
LL_PRINT_FILE	output to file
LL_PRINT_EXPORT	output to an export module that can be defined with *LlPrintSetOptionString(LL_PRNOPTSTR_EXPORT)*

Optionally combined with *LL_PRINT_MULTIPLE_JOBS*: output in several smaller print jobs (see below) with network spooler print.

nReserved: for future extensions

Return Value:

Error code

Hints:

Please check the return value!

nPrintOptions for *LL_PRINT_NORMAL* can be combined with 'or' using *LL_PRINT_MULTIPLE_JOBS* so that the print job can be split into several smaller individual jobs. The number of the page after which the job should split can be set with *LlPrintSetOption()*.

No abort dialog box is displayed, see *LlPrintWithBoxStart()*.

Example:

```
HLLJOB hJob;

hJob = LlJobOpen(0);

if (LlPrintStart(hJob, LL_PROJECT_LABEL, "test.lbl",
   LL_PRINT_NORMAL) == 0)
{
   <... etc ...>
   LlPrintEnd(hJob);
}
else
   MessageBox(NULL, "Error", "List & Label", MB_OK);
LlJobClose(hJob);
```

See also:

LlPrintWithBoxStart, LlPrintEnd, LlPrintSetOption

LlPrintUpdateBox

Syntax:

```
INT LlPrintUpdateBox (HLLJOB hJob);
```

Task:

Allows redrawing of the abort box used if you print with *LlPrintWithBoxStart()*.

Parameter:

hJob: List & Label job handle

Return Value:

Error code

Hints:

This is basically a message loop.

This function should be called if you run lengthy operations to get your data, as it allows the dialog box to react to any necessary window repaint or an abort button press.

List & Label implicitly calls this function on *LlPrint()*, *LlPrintFields()* or *LlPrintSetBoxText()* calls, so it is only needed if your own processing lasts some time.

See also:

LlPrintWithBoxStart, LlPrintSetBoxText

LlPrintWillMatchFilter

Syntax:

```
INT LlPrintWillMatchFilter (HLLJOB hJob);
```

Task:

Specifies whether the present data record matches the filter chosen by the user, i.e. whether it will be printed with the next *LlPrint()* or *LlPrintFields()* function.

Parameter:

hJob: List & Label job handle

Return Value:

<0: Error code
0: Not printed
1: Printed

Hints:

This function can only be called after *LlPrintStart()* or *LlPrintWithBoxStart()*.

The function calculates the filter value using the currently defined data (variables or fields).

Example:

```
if (LlPrintWillMatchFilter(hJob))
    ....
```

See also:

LlPrintGetFilterExpression, LlPrintDidMatchFilter

LlPrintWithBoxStart

Syntax:

```
INT LlPrintWithBoxStart (HLLJOB hJob, UINT nObjType,
   LPCTSTR lpszObjName, INT nPrintOptions, INT nBoxType,
   HWND hWnd, LPCTSTR lpszTitle);
```

Task:

Starts the print job and opens the project file. Supports an abort window.

Parameter:

hJob: Job handle

nObjType: *LL_PROJECT_LABEL*, *LL_PROJECT_LIST* or *LL_PROJECT_CARD*

lpszObjName: The file name of the project with file extension. If you use a data provider as data source, several project files can be passed here semicolon-separated. Then a combination print is performed and the outputs of the individual projects are summarized as total output.

nPrintOptions:

Value	Meaning
LL_PRINT_NORMAL	output to printer
LL_PRINT_PREVIEW	output to preview
LL_PRINT_FILE	output to file
LL_PRINT_EXPORT	output to an export module that can be defined with *LlPrintSetOptionString(LL_PRN-OPTSTR_EXPORT)*

Optionally combined with one of the following flags:

Value	Meaning
LL_PRINT_MULTIPLE_JOBS	output in several smaller print jobs (see below) with network spooler print.
LL_PRINT_REMOVE_-UNUSED_VARS	Fields and variables not required by the project are removed from the internal buffer after printing starts. This can speed up following declarations considerably. The recommended practice however is to query the required data using *LlGetUsedIdentifiers()* which is the better alternative.

These options influence *LL_OPTIONSTR_EXPORTS_ALLOWED*.

nBoxType:

Value	Meaning
LL_BOXTYPE_-NORMALMETER	Abort box with bar meter and text
LL_BOXTYPE_-BRIDGEMETER	Abort box with bridge meter and text
LL_BOXTYPE_EMPTYABORT	Abort box with text
LL_BOXTYPE_-NORMALWAIT	Box with bar meter and text, no abort button
LL_BOXTYPE_BRIDGEWAIT	Box with bridge meter and text, no abort button
LL_BOXTYPE_EMPTYWAIT	Box with text, no abort button
LL_BOXTYPE_STDABORT	Abort box with system bar meter
LL_BOXTYPE_STDWAIT	Box with bar meter, no abort button
LL_BOXTYPE_NONE	No box.

Note that the Boxtype parameter is only listed here for compatibility with older operating systems. By default, the standard progress bar of the operating system is used.

hWnd: Window handle of the calling program (used as parent of the dialog box)

lpszTitle: Title of the abort dialog box, also appears as text in the print manager

Return Value:

Error code

Hints:

Please check the return value!

An application modal abort dialog box is shown as soon as the print starts. Its title is defined by the passed parameter. In the dialog box there is a percentage-meter-control and a two-line static text, both of which can be set using *LlPrintSetBoxText()* to show the user the print progress, and also an abort button if required (see below).

Example:

```
HLLJOB    hJob;
hJob = LlJobOpen(0);

if (LlPrintWithBoxStart(hJob, LL_PROJECT_LABEL, "test.lbl",
      LL_PRINT_NORMAL, LL_BOXTYPE_NORMALMETER, hWnd, "print") == 0)
{
   LlPrintSetBoxText(hJob, "There we go", 0);
   <... etc...>
   LlPrintEnd(hJob, 0);
```

```
}
else
   MessageBox(NULL, "error", "List & Label", MB_OK);
LlJobClose(hJob);
```

See also:

LlPrintStart, LlPrintEnd

LlProjectClose

Syntax:

```
HLLDOMOBJ LlProjectClose(HLLJOB hJob);
```

Task:

This function is only available starting with the Professional Edition! Closes an open project and releases the relevant project file again. The file is not saved! Detailed application examples can be found in chapter "DOM Functions".

Parameter:

hJob: List & Label job handle

Return value:

Error code

Example:

See chapter "DOM Functions".

See also:

LlProjectSave, LlProjectOpen

LlProjectOpen

Syntax:

```
INT LlProjectOpen(HLLJOB hJob, UINT nObjType,
   LPCTSTR pszObjName, UINT nOpenMode);
```

Task:

This function is only available starting with the Professional Edition! Opens the specified project file. Call LlDomGetProject() to retrieve the DOM handle for the project object afterwards. This object is the basis for all further DOM functions. Detailed application examples can be found in chapter "DOM Functions".

Parameter:

hJob: List & Label job handle

nObjType:

Value	Meaning
LL_PROJECT_LABEL	for labels
LL_PROJECT_CARD	for index cards
LL_PROJECT_LIST	for lists

pszObjName: Project file name with path and file extension

nOpenMode: Combination (ORing) of a flag from each of the following three groups:

Value	Meaning
LL_PRJOPEN_CD_OPEN_EXISTING	File must already exist, otherwise error code will be returned.
LL_PRJOPEN_CD_CREATE_ALWAYS	File is always newly created. If it already exists, the content is deleted.
LL_PRJOPEN_CD_CREATE_NEW	File is newly created if it does not exist. If file already exists, error code is returned.
LL_PRJOPEN_CD_OPEN_ALWAYS	If file exists the content is used, otherwise file is newly created.

Value	Meaning
LL_PRJOPEN_AM_READWRITE	File is opened for read/write access.
LL_PRJOPEN_AM_READONLY	File is only opened for read access.

Value	Meaning
LL_PRJOPEN_EM_IGNORE_FORMULAERRORS	Syntax errors are ignored. See notes.

Return value:

Error code

Hints

If the flag LL_PRJOPEN_EM_IGNORE_FORMULAERRORS is used, syntax errors in the project are ignored. This has the advantage that projects can be successfully opened and edited even if the data structure is unknown or undefined. As the formulas in the project are then treated as placeholders, the section with the used variables (see *LlGetUsedIdentifiers()*) cannot be correctly written, if you e.g. add further columns to a table. The content of this section is left unchanged when saving. The same applies for the case where a new table, which has not previously been used, is inserted in a report container. Therefore, LL_PRJOPEN_EM_IGNORE_FORMULAERRORS must not be set for such cases. If the flag is not set, LL_NTFY_EXPRERROR can be used to collect the error messages for display.

Example:

See chapter "DOM Functions".

See also:

LlProjectSave, LlProjectClose, LlDomGetProject

LlProjectSave

Syntax:

```
HLLDOMOBJ LlProjectSave(HLLJOB hJob, LPCTSTR pszObjName);
```

Task:

This function is only available starting with the Professional Edition! Saves an open project. Detailed application examples can be found in chapter "DOM Functions".

Parameter:

hJob: List & Label job handle

pszObjName: Project file name with path and file extension. May be NULL (see notes)

Return value:

Error code

Hints

If pszObjName is NULL, the file is saved under the same name as when it was opened.

Example:

See chapter "DOM Functions".

See also:

LlProjectOpen, LlProjectClose

LlRTFCopyToClipboard

Syntax:

```
INT LlRTFCopyToClipboard(HLLJOB hJob, HLLRTFOBJ hRTF);
```

Task:

Copies the contents of the RTF object to the clipboard. Several clipboard formats are available: CF_TEXT, CF_TEXTW and CF_RTF.

Parameter:

hJob: List & Label job handle

hRTF: RTF object handle

Return Value:

Error code

See also:

LlRTFCreateObject

LlRTFCreateObject

Syntax:

```
HLLRTFOBJ LlRTFCreateObject(HLLJOB hJob);
```

Task:

Creates an instance of a List & Label RTF object to be used in stand-alone mode.

Parameter:

hJob: List & Label job handle

Return Value:

RTF object handle, or NULL in case of an error.

See also:

LlRTFGetText, LlRTFDeleteObject

LlRTFDeleteObject

Syntax:

```
INT LlRTFDeleteObject(HLLJOB hJob, HLLRTFOBJ hRTF);
```

Task:

Destroys the instance of the stand-alone RTF object.

Parameter:

hJob: List & Label job handle

hRTF: RTF object handle

Return Value:

Error code

See also:

LIRTFCreateObject

LlRTFDisplay

Syntax:

```
INT LlRTFDisplay(HLLJOB hJob, HLLRTFOBJ hRTF, HDC hDC,
   _PRECT pRC, BOOL bRestart, LLPUINT pnState);
```

Task:

Paints the contents of the RTF object in a device context (DC). Can be used to display the RTF contents in a window or print them to the printer.

Parameter:

hJob: List & Label job handle

hRTF: RTF object handle

hDC: DC for the device. If NULL, the standard printer will be used.

pRC: Pointer to the rect with logical coordinates (mm/10, inch/100 etc.) in which the contents will be printed. May be NULL for a printer DC, in which case the whole printable area of the page will be used.

bRestart: If TRUE, the output will start at the beginning of the text. Otherwise it will be continued after the point where the previous print ended, thus enabling a multi-page print.

pnState: State value used by the next *LlRTFDisplay()* call

Return Value:

Error code

Example:

```
// Create Printer-DC
HDC   hDC = CreateDC(NULL,"\\\\prnsrv\\default",NULL,NULL);
RECT  rc = {0,0,1000,1000};
BOOL bFinished = FALSE;
```

```
INT nPage = 0;
// Init document
StartDoc(hDC,NULL);
while (!bFinished)
{
   nPage++;
   UINT     nState = 0;
   // Init page
   StartPage(hDC);
   // Prepare DC (set coordinate system)
   SetMapMode(hDC,MM_ISOTROPIC);
   SetWindowOrgEx(hDC,rc.left,rc.top,NULL);
   SetWindowExtEx(hDC,rc.right-rc.left,rc.bottom-rc.top,NULL);
   SetViewportOrgEx(hDC,0,0,NULL);
   SetViewportExtEx(hDC,GetDeviceCaps(hDC,HORZRES),
   GetDeviceCaps(hDC,VERTRES),NULL);
   // print RTF-Text
   BOOL bFinished = (LlRTFDisplay(hJob, hRTF, hDC, &rc, nPage ==
                     1, &nState) == LL_WRN_PRINTFINISHED);
   // done page
   EndPage(hDC);
}
EndDoc(hDC);
```

See also:

LlRTFCreateObject

LlRTFEditObject

Syntax:

```
INT LlRTFEditObject(HLLJOB hJob, HLLRTFOBJ hRTF, HWND hWnd,
   HDC hPrnDC, INT nProjectType, BOOL bModal);
```

Task:

Displays the RTF editor to the user. All variables and – in case of LL_PROJECT_LIST – all fields are available for use in expressions.

Parameter:

hJob: List & Label job handle

hRTF: RTF object handle

hWnd: Handle of parent window or host control

hPrnDC: Reference DC of the destination (usually a printer DC). Important for the choice of available fonts. Can be NULL, in which case the default printer is used.

nProjectType: *Project type (LL_PROJECT_LABEL, LL_PROJECT_CARD or LL_PROJECT_LIST*).

bModal: if TRUE, the dialog will be displayed modally. If FALSE, the control passed as hWnd will be replaced by the RTF control. Please note, that the window created by Visual C++ MFC is not suitable for the non modal mode. We suggest using the RTF OCX control (cmll26r.ocx) instead.

Return Value:

Error code

See also:

LlRTFCreateObject

LlRTFEditorInvokeAction

Syntax:

```
INT LlRTFEditorInvokeAction(HLLJOB hJob, HLLRTFOBJ hRTF,
   INT nControlID);
```

Task:

Allows activation of an action in the RTF control by code. This is important for in-place RTF controls (see *LlRTFEditObject()*) if the hosting application provides a separate menu.

Parameter

hJob: List & Label job handle

hRTF: RTF object handle

nControlID: Control ID of the button to be activated. The IDs of the items in List & Label can be found in the file MENUID.TXT of your List & Label installation.

Return Value:

Error code

See also:

LlRTFCreateObject , LlRTFEditorProhibitAction, LlRTFEditObject

LlRTFEditorProhibitAction

Syntax:

```
INT LlRTFEditorProhibitAction(HLLJOB hJob, HLLRTFOBJ hRTF,
   INT nControlID);
```

Task:

Disables buttons in the RTF control.

Parameter

hJob: List & Label job handle

hRTF: RTF object handle

nControlID: Control ID of the button to be disabled. The IDs of the items in List & Label can be found in the file MENUID.TXT of your List & Label installation.

Return Value:

Error code

See also:

LlRTFCreateObject , LlRTFEditorInvokeAction, LlRTFEditObject

LlRTFGetText

Syntax:

```
INT LlRTFGetText(HLLJOB hJob, HLLRTFOBJ hRTF, INT nFlags,
   LPTSTR lpszBuffer, UINT nBufferSize);
```

Task:

Returns the text of an RTF object

Parameter:

hJob: List & Label job handle

hRTF: RTF object handle

nFlags: Options (see *LlRTFGetTextLength()*)

lpszBuffer: Address of buffer for the text

nBufferSize: Maximum number of characters to be copied

Return Value:

Error code

Hints:

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

Example:

```
HLLRTFOBJ hRTF = LlRTFCreateObject(hJob);
if (LlRTFEditObject(hJob, hRTF, NULL, NULL, LL_PROJECT_LABEL) >= 0)
{
   INT nFlags = LL_RTFTEXTMODE_RTF|LL_RTFTEXTMODE_EVALUATED);
   INT nLen = LlRTFGetTextLength(hJob,hRTF,nFlags);
   TCHAR* pszText = new TCHAR[nLen+1];
   LlRTFGetText(hJob, hRTF, nFlags, pszText, nLen+1);
   printf("'%s'\n\n", pszText);
   delete[] pszText;
}
```

See also:

LlRTFCreateObject, LlRTFGetTextLength

LlRTFGetTextLength

Syntax:

```
INT LlRTFGetTextLength(HLLJOB hJob, HLLRTFOBJ hRTF, INT nFlags);
```

Task:

Returns the size of the text contained in the object. Necessary to determine the required buffer size for the text.

Parameter:

hJob: List & Label job handle

hRTF: RTF object handle

nFlags: One option from each of the two groups mentioned below, combined using a bitwise 'or' (or by addition):

Value	Description
Options for the format of the text to be retrieved:	
LL_RTFTEXTMODE_RTF	RTF-formatted text (incl. RTF control words etc.)
LL_RTFTEXTMODE_PLAIN	Text in plain text format

Options for the evaluation state:	
LL_RTFTEXTMODE_RAW	Text in plain format, with unevaluated formulas if applicable
LL_RTFTEXTMODE_EVALUATED	Text in evaluated format (all formulas replaced by their computed results)

Return Value:

Length of the buffer (negative in case of an error)

See also:

LlRTFCreateObject, LlRTFGetText

LlRTFSetText

Syntax:

```
INT LlRTFSetText(HLLJOB hJob, HLLRTFOBJ hRTF, LPCTSTR lpszText);
```

Task:

Sets the text in the RTF control. The format of the text (plain or RTF) is auto-detected.

Parameter:

hJob: List & Label job handle

hRTF: RTF object handle

lpszText: New contents

Return Value:

Error code

See also:

LlRTFCreateObject

LlSelectFileDlgTitleEx

Syntax:

```
INT LlSelectFileDlgTitleEx (HLLJOB hJob, HWND hWnd, LPCTSTR pszTitle,
   UINT nObjType, LPTSTR pszBuffer, UINT nBufLen, LPVOID pReserved);
```

Task:

Opens a file selection dialog with an optionally integrated preview window.

Parameter:

hJob: List & Label job handle

hWnd: Window handle of the calling program

pszTitle: Title for the dialog

nObjType:

Value	Meaning
LL_PROJECT_LABEL	for labels
LL_PROJECT_CARD	for cards
LL_PROJECT_LIST	for lists

Combined with *LL_FILE_ALSONEW* if a file name for a new (not yet existing) project can be entered.

pszBuffer, nBufSize: Buffer for the file name with file extension. Must be initialized with a file name or an empty string.

pReserved: Reserved, set to NULL or empty ("").

Return Value:

Error code

Hints:

Important for Visual Basic (and some other languages as well), if the OCX control is not used: the buffer must be allocated and initialized by an 0-terminated string.

Advantages compared to a normal CommonDialog: display of the project description, a preview sketch, the language consistency within List & Label and the adaptation of the dialog design.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

Example:

```
char szFilename[260 + 1];
INT nRet;

nRet = LlSelectFileDlgTitleEx(hJob, hWnd, "Report" , LL_PROJECT_LIST,
          szFilename, sizeof(szFilename));
if (nRet == OK)
{
   <then do what you have to do>
}
```

See also:

LL_OPTION_OFNDIALOG_NOPLACESBAR, LL_OPTIONSTR_..._PRJDESCR

LlSetDebug

Syntax:

```
void LlSetDebug (INT nOnOff);
```

Task:

Switches the debug mode on or off.

Parameter:

nOnOff: 0 if debug mode is to be switched off, otherwise the following values can be additionally passed:

Value	Meaning
LL_DEBUG_CMBTLL	to switch on normal debugging-info
LL_DEBUG_CMBTDWG	to switch on debugging-info for graphic functions
LL_DEBUG_CMBTLL_-NOCALLBACKS	switch off debugging-info for notifications/callbacks
LL_DEBUG_CMBTLL_NOSTORAGE	switch off debugging-info for storage- (*LlStgSys...()*-) functions
LL_DEBUG_CMBTLL_NOSYSINFO	do not issue system information dump on *LlSetDebug()*
LL_DEBUG_CMBTLL_LOGTOFILE	debug output will also be directed to a log file (COMBIT.LOG in your %APPDATA% directory).

Hints:

Use the program Debwin included in your package to show the debug output in a separate window.

If debug mode is switched on in List & Label with *LlSetDebug(LL_DEBUG_-CMBTLL),* the DLL prints every function call with the corresponding parameters and results. An '@' is added to the function names, so that the function calls can be easily differentiated from other internal List & Label debugging output.

The output is indented in case a DLL in debugging mode calls other functions of a DLL (even itself) which is also in debugging mode.

Further information can be found in chapter "Debug Tool Debwin".

Example:

```
HLLJOB    hJob;
int       v;

LlSetDebug(LL_DEBUG_CMBTLL | ...);
hJob = LlJobOpen(0);
v = LlGetVersion(VERSION_MAJOR);
LlJobClose(hJob);
```

prints approx. the following in the debugging output:

```
@LlJobOpen(0) = 1
@LlGetVersion(1) = 6
@LlJobClose(1)
```

LlSetDefaultProjectParameter

Syntax:

```
INT LlSetDefaultProjectParameter(HLLJOB hLlJob,
   LPCTSTR pszParameter, LPCTSTR pszValue, UINT nFlags)
```

Task:

Sets the default value of a project parameter (see chapter "Project Parameters")

Parameter:

hJob: List & Label job handle

pszParameter: Parameter name. If this parameter is NULL, all USER parameters will be deleted from the internal list.

pszValue: Parameter value

nFlags: Parameter type. See chapter "Project Parameters" for valid values.

Return Value:

Error code

Hints:

This function should be called before *LlDefineLayout()* and *LlPrint[WithBox]Start()*!

See also:

LlGetDefaultProjectParameter, LlPrintSetProjectParameter, LlPrintGetProject-Parameter

LlSetFileExtensions

Syntax:

```
INT LlSetFileExtensions (HLLJOB hJob, INT nObjType,
   LPCTSTR lpszProjectExt, LPCTSTR lpszPrintExt,
   LPCTSTR lpszSketchExt);
```

Task:

Setting of user-defined file extensions.

Parameter:

hJob: List & Label job handle

nObjType: Project type

Value	Meaning
LL_PROJECT_LABEL	for labels
LL_PROJECT_CARD	for cards
LL_PROJECT_LIST	for lists

lpszProjectExt: Extension

Type	Default
LL_PROJECT_LABEL	"lbl"
LL_PROJECT_CARD	"crd"
LL_PROJECT_LIST	"lst"

lpszPrintExt: Extension for printer definitions file

Type	Default
LL_PROJECT_LABEL	"lbp"
LL_PROJECT_CARD	"crp"
LL_PROJECT_LIST	"lsp"

lpszSketchExt: Extension for file dialog sketch

Type	Default
LL_PROJECT_LABEL	"lbv"
LL_PROJECT_CARD	"crv"
LL_PROJECT_LIST	"lsv"

Return Value:

Error code

Hints:

It is important that all 9 file extensions are different!

Please call this function before *LlDefineLayout()* and before the functions *LlPrint...Start()*, preferably directly after *LlJobOpen()* or *LlJobOpenLCID()*.

You can also get and set these extensions with *LlSetOptionString()*.

Example:

```
HLLJOB hJob;
int    v;

hJob = LlJobOpen(0);
v = LlSetFileExtensions(hJob, LL_PROJECT_LIST, "rpt", "rptp", "rptv");
// ....
LlJobClose(hJob);
```

LlSetNotificationCallback

Syntax:

```
FARPROC LlSetNotificationCallback (HLLJOB hJob, FARPROC lpfnNotify);
```

Task:

Definition of a procedure which will be called for notifications.

Parameter:

hJob: List & Label job handle

lpfnNotify: The address of a function (see below)

Return Value:

Address of the procedure if successful, NULL otherwise

Hints:

The callback function has higher priority than the message; if it is defined no message is sent, but the callback function is called.

This function cannot be used if the .NET component, OCX or VCL controls are used.

The callback function has the following definition:

```
LPARAM STDCALL MyCallback(UINT nFunction, LPARAM lParam)
```

and must be an exported function

The definition of the parameter nFunction and lParam can be found in chapter "Callbacks and Notifications".

Example:

```
LPARAM STDCALL MyCallback(UINT nFunction, LPARAM lParam)
{  //....}

HLLJOB hJob;
unsigned int wMsg;

LlSetDebug(TRUE);
hJob = LlJobOpen(0);
v = LlSetNotificationCallback(hJob, MyCallback);
// ....
LlJobClose(hJob);
```

See also:

LlGetNotificationMessage, LlSetNotificationMessage

LlSetNotificationCallbackExt

Syntax:

```
FARPROC LlSetNotificationCallbackExt (HLLJOB hJob, INT nEvent,
         FARPROC lpfnNotify);
```

Task:

Definition of a procedure which will be called for notifications of the given event.

Parameter:

hJob: List & Label job handle

nEvent: Event-ID (LL_CMND_xxx or LL_NTFY_xxxx)

lpfnNotify: The address of a function (see below)

Return Value:

Address of the procedure if successful, NULL otherwise

Hints:

The "specialized" callback function has a higher priority than the "general" callback function or a message.

List & Label first of all searches for a specialized callback function for the event to be raised. If one is defined, it will be called. If not, List & Label checks whether a general callback handler has been installed with *LlSetNotificationCallback()*. If so, it will be called. If not, List & Label checks whether a message for the current event has been defined using *LlSetNotificationMessage().* If so, the message will be sent. This function may not be used with the .NET component as this component already uses the API for its own functionality.

The callback function has the following definition:

```
LPARAM STDCALL MyCallback(UINT nFunction, LPARAM lParam)
```

and must be an exported function

The definition of the parameter nFunction and lParam can be found in chapter "Callbacks and Notifications".

Example:

```
LPARAM STDCALL MyCallback(UINT nFunction, LPARAM lParam)
//....

HLLJOB hJob;
unsigned int wMsg;
hJob = LlJobOpen(0);
v = LlSetNotificationCallbackExt(hJob,
LL_CMND_CHANGE_DCPROPERTIES_DOC, MyCB);
// ....
LlJobClose(hJob);
```

See also:

LlSetNotificationCallback

LlSetNotificationMessage

Syntax:

```
UINT LlSetNotificationMessage (HLLJOB hJob, UINT nMessage);
```

Task:

Definition of a message number which differs from the presetting for callback (USER) objects.

Parameter:

hJob: List & Label job handle

nMessage: The new message number

Return Value:

Error code

Hints:

The default message number has the value of the function RegisterWindowMessage("cmbtLLMessage").

The callback function has higher priority; if this is defined, no message is sent.

The definition of the parameter nFunction and lParam can be found in chapter "Callbacks and Notifications".

Example:

```
HLLJOB hJob;
unsigned int    wMsg;

LlSetDebug(TRUE);
hJob = LlJobOpen(0);
v = LlSetNotificationMessage(hJob, WM_USER + 1);
// ....
LlJobClose(hJob);
```

See also:

LlGetNotificationMessage, LlSetNotificationCallback

LlSetOption

Syntax:

```
INT LlSetOption (HLLJOB hJob, INT nMode, INT_PTR nValue);
```

Task:

Sets diverse options in List & Label.

Parameter:

hJob: List & Label job handle

nMode: Mode index, see below

nValue: New value

Return Value:

Error code

Hints:

Please call this function before *LlDefineLayout()* and before the functions *LlPrint...Start()*, preferably directly after *LlJobOpen()/LlJobOpenLCID()*.

LL_OPTION_ADDVARSTOFIELDS

TRUE: in list projects, the formula wizard offers variables in addition to fields in a table column formula.

FALSE: in table objects, only fields will be offered (default).

This option only affects list projects.

LL_OPTION_ALLOW_COMBINED_COLLECTING_OF_DATA_FOR_-COLLECTIONCONTROLS

TRUE: If multiple report container elements use the same data source (ex. multiple charts or crosstabs etc.), the data is passed only once to the report and is re-used by all these elements. Depending on the project, this might yield a noticeable performance boost. The main drawback is a larger memory footprint. Additionally, it is no longer possible to change the value of variables that influence the properties or contents of the elements during printing (default).

FALSE: Each element gets its own data.

LL_OPTION_ALLOW_LLX_EXPORTERS

TRUE: List & Label will accept export modules that are loaded during *LL_OPTIONSTR_LLXPATHLIST*

FALSE: List & Label will not use export module functionality.

This option must be set before the LLXPATHLIST call.

Default: TRUE

LL_OPTION_CALCSUMVARSONINVISIBLELINES

This sets the default value for the Designer option specifying whether or not sum variables should also be calculated if data lines are suppressed. The value selected in the Designer will then be saved in and loaded from the project file.

Default: FALSE

LL_OPTION_CALC_SUMVARS_ON_PARTIAL_LINES

TRUE: The sum variables are updated as soon as one data line for the record has been printed.

FALSE: The sum variables are updated as soon as all data lines have been completely printed.

Default: FALSE

LL_OPTION_CALLBACKMASK

The value can be any combination of the following values:

LL_CB_PAGE, *LL_CB_PROJECT*, *LL_CB_OBJECT*, *LL_CB_HELP*, *LL_CB_TABLELINE*, *LL_CB_TABLEFIELD*

For the definition of parameters, please read the chapter on callbacks.

LL_OPTION_CALLBACKPARAMETER

Sets a parameter that is passed in the scCallback structure to any of the callbacks. Please refer to the chapter on callbacks for further details.

LL_OPTION_CODEPAGE

This option sets or reads the code page which is used for all SBCS/DBCS and Unicode translations in List & Label.

This applies to the "A" API of the DLL as well as reading/writing project files.

This setting is used globally, so it is valid for all List & Label jobs in one task, and the hJob parameter is ignored.

The code page has to be installed on the system (NLS[1] for this code page must be installed).

Default: *CP_ACP*.

LL_OPTION_COMPAT_PROHIBITFILTERRELATIONS

This prevents the link from being created using a filter when a table sub-element is added. The dialog for selecting the link type is not displayed and the link is always created using relations.

Default: FALSE

[1] NLS = National Language Support

LL_OPTION_COMPRESSRTF

The text of an RTF control is stored in the project file. When this option is set to TRUE, the text will be compressed.

Set this option to *FALSE* if you want to see the text in the project file (for example for debugging).

Default: *TRUE*

LL_OPTION_COMPRESSSTORAGE

TRUE: the preview data will be compressed. This is a bit slower, but saves a lot of disk space.

FALSE: no compression (default).

LL_OPTION_CONVERTCRLF

TRUE: List & Label translates CR-LF combinations in variable and field contents to LF (and prevents duplicate line breaks) (default).

FALSE: contents remain unchanged.

LL_OPTION_DEFAULTDECSFORSTR

This option sets the number of decimal places that the Designer function Str$() uses if the number is not defined by the user in the Designer.

Default: 5

LL_OPTION_DEFDEFFONT

Allows you to set the handle of the font used as default for the project's default font. The handle need not be valid after the API call, an own copy of the font will be used.

This font can be set by *LL_OPTIONSTR_DEFDEFFONT*.

Default: *GetStockObject(ANSI_VAR_FONT)*

LL_OPTION_DELAYTABLEHEADER

This option defines whether List & Label prints the table header when calling *LlPrint()* or when first printing a table line (*LlPrintFields()*):

TRUE: at *LlPrintFields()*, thus triggered by the first table line (Default)

FALSE: at *LlPrint()*. Of course, if fields are used in the header line, they must be defined at the *LlPrint()* call.

LL_OPTION_ DESIGNEREXPORTPARAMETER

See chapter "Direct Print and Export From the Designer".

LL_OPTION_ DESIGNERPREVIEWPARAMETER

See chapter "Direct Print and Export From the Designer".

LL_OPTION_ DESIGNERPRINT_SINGLETHREADED

See chapter "Direct Print and Export From the Designer".

LL_OPTION_ERR_ON_FILENOTFOUND

TRUE: if a graphic file is not found during print time LL_ERR_DRAWINGNOTFOUND will be returned

FALSE: the error will be ignored without any feedback. (Default)

LL_OPTION_ESC_CLOSES_PREVIEW

This option defines whether the "Escape" key closes the preview window. Default: FALSE.

LL_OPTION_EXPRSEPREPRESENTATIONCODE

Character code of the character that is used to divide an expression into multiple lines in the expression wizard.

This value might have to be changed for code pages other than standard Western code page, as the default might be used for a printable character.

LL_OPTION_FAVORITE_SETTINGS

This option defines the behavior of the property favorites. The following values can be comined.

Value	Meaning
LL_FAVORITES_ENABLE_FAVORITES_BY_DEFAULT	Favorites are editable, button is activated by default, registry settings will be ignored.
LL_FAVORITES_HIDE_FAVORITES_BUTTON	Favorites are not editable, button is invisible.

Default: Favorites are editable, the button shows the last state from the registry.

LL_OPTION_FONTQUALITY

(See also LL_OPTION_FONTPRECISION below)

Can be used to influence the Windows font mapper, for example to use a device font. The value set by this option will be used for the LOGFONT.lfQuality field when a font instance is being created.

The permitted values are referenced in the MSDN documentation.

Default: DEFAULT_QUALITY.

LL_OPTION_FONTPRECISION

(See also LL_OPTION_FONTQUALITY above)

Can be used to influence the font mapper of Windows, for example to use a device font. The value set by this option will be used for the LOGFONT.lfOutPrecision field when a font instance is being created.

The permitted values are referenced in the MSDN documentation.

Default: OUT_STRING_PRECIS.

LL_OPTION_FORCE_DEFAULT_PRINTER_IN_PREVIEW

TRUE: printer name setting will not passed on to the preview, so that the preview always uses the default printer in its print routines

FALSE: printer name setting will be passed on to the preview (default)

LL_OPTION_FORCEFONTCHARSET

Selects whether all fonts in the system are offered in font selection combo boxes or whether they must support the charset of the default LCID (or the font set with *LL_OPTION_LCID*). See also *LL_OPTION_SCALABLEFONTSONLY*.

Default: FALSE

LL_OPTION_FORCEFIRSTGROUPHEADER

Set to TRUE to force the first group header to be printed even if the evaluated result of the "group by" property is empty.

Default: FALSE

LL_OPTION_HELPAVAILABLE

TRUE: display help buttons (default)

FALSE: do not display help buttons

LL_OPTION_IDLEITERATIONCHECK_MAX_ITERATIONS

This option is used to set the maximum number of attempts to print an object. Useful to prevent endless loops if non-wrappable content is exceeding the available space.

Default: 0 (no limit)

LL_OPTION_IMMEDIATELASTPAGE

FALSE: the *LastPage()* flag will not be set before all objects have been printed (up to a table in case of a report project).

TRUE: a non-finished object will immediately force *LastPage()* to be FALSE, and will reset all its appended objects.

Default: *TRUE*

LL_OPTION_INCREMENTAL_PREVIEW

TRUE: The preview is displayed as soon as the first page has been created and further pages are added to the display incrementally. If the user closes the preview window during printing, you will receive LL_ERR_USER_ABORTED from the print functions. This error code must therefore be processed in any case. If *LlPreviewDisplay()* is called at the end of printing, this API will only return when the user closes the preview window.

FALSE: The preview is not displayed immediately, the application must explicitly call *LlPreviewDisplay()* for display.

Default: TRUE

LL_OPTION_INTERCHARSPACING

TRUE: the space between the characters for block-justified text will vary.

FALSE: only the width of spaces between words will be varied (default)

LL_OPTION_INCLUDEFONTDESCENT

TRUE: the logfont member LOGFONT.lfDescent is considered when calculating the line distances. This leads to a wider line space but prevents extreme font descents from being cut off (Default from version 13 on).

FALSE: compatible mode

LL_OPTION_LCID

When you set this option, the default values for locale-dependent parameters are set accordingly (inch/metric unit, decimal point, thousands separator, currency symbol and fonts (see *LL_OPTION_FORCEFONTCHARSET*)).

It also defines the default locale for the *Loc...$()* and *Date$()* functions.

Default: *LOCALE_USER_DEFAULT*.

LL_OPTION_LOCKNEXTCHARREPRESENTATIONCODE

Character code of the character that represents a 'line break lock' in the Designer.

This value might have to be changed for code pages other than standard western code page, as the default might be used for a printable character. In most cases you can also use the Code 160 (NO BREAK SPACE).

LL_OPTION_MAXRTFVERSION

Windows or Microsoft applications are supplied with many different RTF controls that support different features and show a different behaviour.

Using this option, you can set the maximum version number of the RTF control to use in the first step. For example, setting the option to 0x100 causes List & Label to load RTF control version 1 (if it exists). Setting the option to 0x401 causes List & Label to try loading RTF control version 4.1. If no control with a version smaller or equal the selected version can be loaded a control with a higher version will be used instead to avoid a loss of data.

To not load any RTF control you should set this option to 0. Advantage is a faster start up and using less resources.

This option must be called with job handle –1 before the fist List & Label job has been opened.

LL_OPTION_METRIC

TRUE: List & Label Designer is set to metric system

FALSE: List & Label Designer is set to inches

Default value depends on the system's setting.

See *LL_OPTION_UNITS*

LL_OPTION_NOAUTOPROPERTYCORRECTION

FALSE: Setting interdependent properties mutually influences each other. (Default)

TRUE: Interdependent properties can be set independently.

This option is sometimes required when working with the DOM object model to prevent automatic property switching. If e.g. the font of a paragraph is set to a Chinese font, the property "Charset" would be automatically switched accordingly. If this is not desired, use this option to switch the behavior.

LL_OPTION_NOCONTRASTOPTIMIZATION

FALSE: Contrast optimization for table fields, charts and crosstab cells. This automatically changes from black to white and vice versa, based on the contrast of the font color when compared to the background. (Default)

TRUE: No contrast optimization.

LL_OPTION_NOFAXVARS

FALSE: The variables for fax are visible in the Designer (default).

TRUE: The variables for fax are not visible in the Designer.

LL_OPTION_NOFILEVERSIONUPGRADEWARNING

This option defines the behavior of the Designer when opening a project file from an older version of List & Label.

TRUE: Conversion takes place without user interaction.

FALSE: A warning box is displayed to the user, indicating that the project file will not be editable by an older version of List & Label once it has been saved with the current version (default).

LL_OPTION_NOMAILVARS

FALSE: The variables for email are visible in the Designer (default).

TRUE: The variables for email are not visible in the Designer.

LL_OPTION_NONOTABLECHECK

TRUE: For a list project, List & Label does not check whether at least one table object is present (default).

FALSE: List & Label performs the check and returns *LL_ERR_NO_TABLEOBJECT* if the project contains no table.

LL_OPTION_NOPARAMETERCHECK

TRUE: List & Label does not check the parameters passed to its DLL functions, which results in a higher processing speed.

FALSE: The parameters will be checked (default).

LL_OPTION_NOPRINTERPATHCHECK

TRUE: List & Label does not check if the printers that are relevant for the project exists. If for example network printers are used in the project, that are currently not available, waiting time will occur.

FALSE: List & Label checks if the printers that are relavant for the project exist (default).

LL_OPTION_NOPRINTJOBSUPERVISION

With this option monitoring of the print jobs can be switched on (see *LL_INFO_PRINTJOBSUPERVISION)*. Default: *TRUE*.

LL_OPTION_NOTIFICATIONMESSAGEHWND

Sets the window that is to receive notification messages (callbacks) when no callback procedure is explicitly set.

Usually events are sent to the first non-child window, starting with the window handle given in *LIDefineLayout()* or *LIPrintWithBoxStart()*. You can modify the destination with this call.

Default: NULL (default behavior)

LL_OPTION_NULL_IS_NONDESTRUCTIVE

List & Label handles NULL-values according to the SQL-92 specification where possible. An important effect of that is, that functions and operators, which get NULL-values as parameter or operator generally also return NULL as the result. An example is the following Designer formula:

```
Title+" "+Firstname+" "+Lastname
```

If Title is filled with NULL, the result of the formula is also NULL according to the standard. As it often can be desired to get

```
Firstname+" "+Lastname
```

instead, the option *LL_OPTION_NULL_IS_NONDESTRUCTIVE* defines that NULL-values do not result in the complete expression to become NULL (against the specification) but according to the data type will become "0", an empty string or an invalid date. The better alternative is however to work with *NULLSafe()* Designer functions, where the replacement value can be defined exactly in case of NULL.

TRUE: NULL-values will be displayed by replacement values.

FALSE: NULL-values as operators or parameter result in NULL as function value (default).

LL_OPTION_PARTSHARINGFLAGS

This option allows you to use the variables passed to List & Label within the report sections Table of Contents, Index and Reverse Side. The following flags can be or'ed:

Value	Meaning
LL_PARTSHARINGFLAG_VARIABLES_TOC (0x01)	Table of contents
LL_PARTSHARINGFLAG_VARIABLES_IDX (0x02)	Index
LL_PARTSHARINGFLAG_VARIABLES_GTC (0x04)	Reverse side

LL_OPTION_PHANTOMSPACEREPRESENTATIONCODE

Character code of the character that represents a 'phantom space' in the Designer.

This value might have to be changed for code pages other than standard Western code page, as the default might be used for a printable character.

LL_OPTION_POSTPAINT_TABLESEPARATORS

TRUE: In a table object, the cell borders will be painted only after painting the complete background so that rounding errors during painting (background of the following line can paint over the lower frame line) are avoided (Default from version 23 on).

FALSE: Compatible mode, the cell borders will be painted directly after each cell.

LL_OPTION_PREVIEW_SCALES_RELATIVE_TO_PHYSICAL_SIZE

This option allows by using flags (0x1: Designer, 0x2: Preview Window) to decide where the preview should match the true physical size on screen when setting the zoom to 100%. Requires Win 8.1 or newer.

LL_OPTION_PRINTERDCCACHE_TIMEOUT_SECONDS

Determines how long (in seconds) printer device contexts are cached. Please note that some printers do not start a print job before the corresponding device context is closed. For these, you might want to change this setting to 0. The default value is 60.

LL_OPTION_ PRINTERDEVICEOPTIMIZATION

TRUE: Printers that are effectively equal concerning their DEVMODE structs are optimized away (default). This also means that print jobs may be collated even if there are pages with different printer settings in between. To prevent this from happening, you may switch this option to FALSE.

FALSE: All printers are shown, print jobs are not collated.

LL_OPTION_PROHIBIT_USERINTERACTION

TRUE: No message boxes and dialogs will be displayed. Message boxes will automatically return the default value. This option is usually set automatically in webserver environments. If the webserver detection should fail for some reason, you can manually force the non-UI mode via this option.

FALSE: Message boxes and dialogs are displayed (default).

LL_OPTION_PROJECTBACKUP

TRUE: A backup file is generated during editing in the Designer (default).

FALSE: No backup file is generated.

LL_OPTION_PRVZOOM_LEFT, LL_OPTION_PRVZOOM_TOP, LL_OPTION_PRVZOOM_WIDTH, LL_OPTION_PRVZOOM_HEIGHT

Preview: the rectangle coordinates of the preview window in percentage of the screen. If this is not set (set to -1), the positions of the window when it was last closed will be used.

LL_OPTION_PRVRECT_LEFT, LL_OPTION_PRVRECT_TOP, LL_OPTION_PRVRECT_WIDTH, LL_OPTION_PRVRECT_HEIGHT

The same in screen pixels.

LL_OPTION_PRVZOOM_PERC

Preview: initial zoom factor in percentage (default: 100). To zoom to page width, set the value to -100.

LL_OPTION_REALTIME

TRUE: Time() and *Now()* will always use the current time

FALSE: The time will be fixed once when the project is loaded (default)

LL_OPTION_RESETPROJECTSTATE_FORCES_NEW_DC

TRUE: The output device context will be created new after *LlPrintResetProjectState()* (default).

FALSE: The device context is preserved after *LlPrintResetProjectState()*. Is not supported by all printers, but results in increased performance with merge print.

LL_OPTION_RESETPROJECTSTATE_FORCES_NEW_PRINTJOB

TRUE: A new print job is forced after *LlPrintResetProjectState()*. This option is especially of use if the same project is printed consecutively multiple times and when it is important that every one of the prints creates its own job in the spooler.

FALSE: Multiple reports can be merged into one print job. A new print job is only created if it is necessary due to duplex prints (default).

LL_OPTION_RETREPRESENTATIONCODE

Character code of the character that represents a 'new line' in the Designer.

This value might have to be changed for code pages other than standard Western code page, as the default might be used for a printable character.

LL_OPTION_RIBBON_DEFAULT_ENABLEDSTATE

TRUE: The Designer and the print preview use the Windows Ribbon framework. The Ribbon may be switched off in the project's options dialog (default).

FALSE: Usage of the Ribbon has to be enabled explicitly in the project's options dialog.

LL_OPTION_RTFHEIGHTSCALINGPERCENTAGE

Percentage value to wrap RTF text a bit earlier so that MS Word does show RTF text completely (default: 100).

LL_OPTION_SCALABLEFONTSONLY

Here you can choose which fonts can be selected in font selection dialogs: only scalable fonts (TrueType and Vector fonts, *TRUE*) or all (*FALSE*).

Raster fonts have the disadvantage of not being scalable, so the preview may not appear as expected.

Default: *TRUE*

LL_OPTION_SETCREATIONINFO

List & Label can store some information about the user (user and computer name, date and time of creation as well as last modification) in the project file and the preview file. This can be important for tracing modifications.

TRUE: Save info (default)

FALSE: Suppress info

LL_OPTION_SHOWPREDEFVARS

TRUE: The internal variables of List & Label will be listed in the variable selection dialog of the formula wizard (default).

FALSE: These will be suppressed.

LL_OPTION_SKETCH_COLORDEPTH

This option sets the color depth of the sketch files for the file selection dialogs. Default is 8, i.e. 256 colors. 32 would be true color.

LL_OPTION_SKIPRETURNATENDOFRTF

RTF texts may contain blank lines at the end.

TRUE: These are removed

FALSE: The blank lines are printed (default).

LL_OPTION_SORTVARIABLES

TRUE: The variables and fields in the selection dialog are sorted alphabetically.

FALSE: The variables and fields in the selection dialog are not sorted (default).

LL_OPTION_SPACEOPTIMIZATION

TRUE: List & Label will default the "space optimization" feature for new paragraphs in text objects and new fields (default).

FALSE: The default state will be unchecked.

This does not apply to existing objects!

LL_OPTION_SUPERVISOR

TRUE: All menu options are allowed, and even locked objects are not locked. This mode enables sections that are not accessible to the user to be used without much additional programming.

FALSE: Limitations are valid (default)

LL_OPTION_SUPPORTS_PRNOPTSTR_EXPORT

TRUE: The user can choose a default exporter for the project, which will be preset in the print options dialog

FALSE: Usually means that the application sets the default exporter

The default print medium is stored in the Project file.

Default: FALSE

LL_OPTION_TABLE_COLORING

LL_COLORING_DESIGNER: the coloring of list objects may only be carried out by List & Label (default)

LL_COLORING_PROGRAM: the coloring of list objects is only carried out by notifications or callback (see chapter "Notifications and Callbacks"); color setting in the Designer is not possible

LL_COLORING_DONTCARE: the coloring is first of all carried out by the program by notification or callback and then additionally by List & Label.

LL_OPTION_TABREPRESENTATIONCODE

Character code of the character that represents a 'tab' in the Designer.

This value might have to be changed for code pages other than standard Western code page, as the default might be used for a printable character.

LL_OPTION_UNITS

Description and values see LL_PRNOPT_UNIT.

LL_OPTION_USEBARCODESIZES

Some barcodes have size limitations (minimum and/or maximum sizes). If this option is set to TRUE, the user is allowed to switch on the size limitation feature so that when resizing the barcode object, only sizes in the standard size range will be allowed.

LL_OPTION_USECHARTFIELDS

TRUE: Chart objects will get their data through the chart API.

FALSE: Compatible mode, charts get their data through *LlPrintFields()*. (Default)

Please read the hints in the Chart chapter of this manual.

LL_OPTION_USEHOSTPRINTER

TRUE: List & Label passes all printer device operations to the host application, which then has more freedom but also has to work harder. See *LL_CMND_HOSTPRINTER*.

FALSE: List & Label manages the printer device

LL_OPTION_USE_JPEG_OPTIMIZATION

TRUE: List & Label embeds JPEG files as JPEG stream into the preview (meta) files. This leads to a significantly decreased file size but the meta files will only be readable from List & Label and not from any third-party picture editors anymore. (Default).

FALSE: JPEG files will be embedded as bitmap records into the preview (meta) files, which will result in significantly larger file sizes.

LL_OPTION_VARLISTDISPLAY

Determines the order of the variables/fields and folders in the associated tool window.

The following flag groups can be oreded:

Value	Meaning
LL_OPTION_VARLISTDISPLAY_VARSORT_DECLARATIONORDER (0x00)	Display of variables in declaration order
LL_OPTION_VARLISTDISPLAY_VARSORT_ALPHA (0x01)	Display of variables in alphabetical order

Value	Meaning
LL_OPTION_VARLISTDISPLAY_FOLDERPOS_DECLARATIONORDER (0x00)	Display folders in declaration order

Value	Meaning
LL_OPTION_VARLISTDISPLAY_FOLDERPOS_ALPHA (0x10)	Display folders in alphabetical order
LL_OPTION_VARLISTDISPLAY_FOLDERPOS_TOP (0x20)	Display folders top
LL_OPTION_VARLISTDISPLAY_FOLDERPOS_BOTTOM (0x30)	Display folders below

LL_OPTION_VARSCASESENSITIVE

TRUE: Variable and field names are case-sensitive

FALSE: Variable and field names are not case-sensitive ("Name" defines the same variable as "NAME"). This option results in a slightly lower speed. (Default)

LL_OPTION_XLATVARNAMES

TRUE: Special characters in variable and field names will be converted to '_'. (Default)

FALSE: Variable and field names will not be modified. This has a speed advantage, but you must make sure that the field and variable names do not contain these characters. Should be switched to FALSE when using MBCS.

Example:

```
HLLJOB hJob;

LlSetDebug(TRUE);
hJob = LlJobOpen(0);
v = LlSetOption(hJob, LL_OPTION_NEWEXPRESSIONS, TRUE);
// ....
LlJobClose(hJob);
```

See also:

LlGetOption, LlGetOptionString, LlSetOptionString

LlSetOptionString

Syntax:

```
INT LlSetOptionString (HLLJOB hJob, INT nMode, LPCTSTR pszValue);
```

Task:

Sets string options in List & Label.

Parameter:

hJob: List & Label job handle

nMode: Mode index, see below

pszValue: New value

Return Value:

Error code

Hints:

Most of the options need to be set before *LlDefineLayout()* and before the functions *LlPrint...Start()*, preferably directly after *LlJobOpen()/LlJobOpenLCID()*. If an option needs to be set at a different time, this will be stated in that option's description.

LL_OPTIONSTR_CARD_PRJEXT

The file extension for a file card project. Default "crd".

LL_OPTIONSTR_CARD_PRVEXT

The file extension for the bitmap of a file card project that will be shown in the File Open dialog. Default "crv".

LL_OPTIONSTR_CARD_PRNEXT

The file extension for the printer definition file of a file card project. Default "crp".

LL_OPTIONSTR_CURRENCY

This represents the string that is used as currency symbol in the *fstr$()* function.

The default is the value of the user settings in the system, but will be set to the respective locale value on *LL_OPTION_LCID*.

LL_OPTIONSTR_DECIMAL

This represents the string that is used as decimal char in the *fstr$()* function.

The default is the value of the user settings in the system, but will be set to the respective locale value on *LL_OPTION_LCID*.

LL_OPTIONSTR_DEFDEFFONT

Sets the font to be used as default for the project font.

The parameter must have the following format:

"{(R,G,B),H,L}"
R = Red intensity, G = Green intensity, B = Blue intensity
H = Height in points, L = Comma-separated fields of the LOGFONT structure (See SDK)

This DEFDEFFONT can be set using *LL_OPTION_DEFDEFFONT* as handle.

LL_OPTIONSTR_EMBEDDED_EXPORTS

With this option you can embed different export formats in the preview in a multi-pass procedure so that they are also available in the viewer. Pass a semicolon-separated list of the desired formats, e.g. "DOCX;XLS". Note that your application must support the drilldown event for this feature to be available. The data binding of the .NET component provides this support automatically.

LL_OPTIONSTR_EXPORTS_ALLOWED

This property can be used to restrict the output list presented to the user in the *LlPrintOptionsDialog[Title]()* dialog. Also, only the allowed export formats can be configured in the designer.

Pass a semicolon-separated list of allowed export IDs, see *LL_OPTIONSTR_EXPORTS_AVAILABLE*

Example:

```
LlPrintStart(hJob,...,LL_PRINT_EXPORT,...);

// allow only printer and preview (EXPORT sets all bits)
LlSetOptionString(hJob, LL_OPTIONSTR_EXPORTS_ALLOWED, "PRN;PRV");
// Default should be preview!
LlPrintSetOptionString(hJob, LL_PRNOPTSTR_EXPORT, "PRV");

// printer dialog allows user to change
LlPrintOptionsDialog(hJob,....);
// get the final medium:
LlPrintGetOption(hJob, LL_PRNOPTSTR_EXPORT, sMedium,
sizeof(sMedium));
// ...print job....
// finished
LlPrintEnd(hJob,0);

if (strcmp(sMedium,"PRV") == 0) ...
```

LL_OPTIONSTR_EXPORTS_ALLOWED_IN_PREVIEW

This property can be used to restrict the list of possible output formats in the preview dialog.

Pass a semicolon-separated list of allowed export IDs, see *LL_OPTIONSTR_EXPORTS_AVAILABLE*

LL_OPTIONSTR_EXPORTS_AVAILABLE

This is a read-only property.

This function returns a semicolon-separated list of all output media (usually "PRN;PRV;FILE" and the list of the export modules, if any are loaded by *LL_OPTIONSTR_LLXPATHLIST*), for example "PRN;PRV;FILE;HTML;RTF"

The following IDs are predefined if the corresponding modules are installed:

Value	Meaning
PRN	Printer
PRV	Preview
PRES	Presentation
PDF	Adobe PDF Format
XHTML	XHTML/CSS Format
MHTML	Multi-Mime HTML Format
XLS	Microsoft Excel Format
DOCX	Microsoft Word Format
RTF	Rich Text Format (RTF)
XPS	Microsoft XPS Format
PICTURE_MULTITIFF	Multi-TIFF Picture
PICTURE_TIFF	TIFF Picture
PICTURE_PNG	PNG Picture
PICTURE_JPEG	JPEG Picture
PICTURE_BMP	Bitmap Picture
PICTURE_EMF	Metafile Picture (EMF)
FILE	Printing to printer file
PPTX	Microsoft PowerPoint Format
TTY	Pinwriter (TTY)
SVG	SVG Format
TXT	Text (CSV) Format
TXT_LAYOUT	Text (Layout) Format
XML	XML Format

The following formats are not supported anymore and are only available for compatibility reasons. If you still want to use the format you have to enable it explicitly via LlSetOption(hJob, LL_OPTIONSTR_LEGACY_EXPORTERS_ALLOWED,...) or via LL.Core.LlSetOptionString(...).

Value	Meaning
HTML	HTML Format
JQM	HTML jQuery Mobile Format

LL_OPTIONSTR_EXPORTFILELIST

This is a read-only property.

After *LlPrintEnd()*, you can use this function to get a list of files that have been created by the export process.

The return value is a semicolon-separated list of the path names of the files.

This list can be very large, so please allocate sufficient buffer and check the return value of *LISetOption()* on the error value (*LL_ERR_BUFFERTOOSMALL*).

LL_OPTIONSTR_HELPFILENAME

You can use this function to force the help file name, e.g. if you want to display your own help file.

LL_OPTIONSTR_FAX_RECIPNAME, LL_OPTIONSTR_FAX_RECIPNUMBER, LL_OPTIONSTR_FAX_SENDERNAME, LL_OPTIONSTR_FAX_SENDERCOMPANY , LL_OPTIONSTR_FAX_SENDERDEPT, LL_OPTIONSTR_FAX_SENDERBILLINGCODE

These options set a default value for the variables in the fax dialog (Project > Fax Variables). Any changes made by the user in the Designer will override these values.

If the project is sent by fax, these expressions will be evaluated and directly used as parameters for the MS Fax module.

As an alternative, these expressions are also available as variables (LL.Fax.xxxx), so that they can be placed in the project in a special format to be used by other fax drivers. For details, please see the fax software manual.

If these options are not set and the user has not entered any expressions in the fax dialog, the "MS FAX" export will not be available.

LL_OPTIONSTR_LABEL_PRJDESCR, LL_OPTIONSTR_CARD_PRJDESCR, LL_OPTIONSTR_LIST_PRJDESCR, LL_OPTIONSTR_TOC_PRJDESCR, LL_OPTIONSTR_IDX_PRJDESCR, LL_OPTIONSTR_GTC_PRJDESCR

Use this parameter to set the description of the corresponding project types. This description is displayed in the file type combobox of the load and save dialogs. It is recommended to set the corresponding _SINGULAR options as well.

LL_OPTIONSTR_LABEL_PRJDESCR_SINGULAR,
LL_OPTIONSTR_CARD_PRJDESCR_SINGULAR,
LL_OPTIONSTR_LIST_PRJDESCR_SINGULAR,
LL_OPTIONSTR_TOC_PRJDESCR_SINGULAR,
LL_OPTIONSTR_IDX_PRJDESCR_SINGULAR,
LL_OPTIONSTR_GTC_PRJDESCR_SINGULAR

Use this parameter to set the description of the corresponding project types in the singular. This description is displayed in the file type combobox of the load and save dialogs. These options are used in the repository-mode and might be used otherwise in the future as well.

LL_OPTIONSTR_LABEL_PRJEXT

The file extension for a label project. Default "lbl".

LL_OPTIONSTR_LABEL_PRVEXT

The file extension for the bitmap of a label project that will be shown in the File Open dialog. Default "lbv".

LL_OPTIONSTR_LABEL_PRNEXT

The file extension for the printer definition file of a label project. Default "lbp".

LL_OPTIONSTR_LICENSINGINFO

This option defines the licensing. You need to set your own personal license key here.

This option must be set before you distribute your project! Further information is available in the files "PersonalLicense.txt" and "Redist.txt" in the List & Label installation directory. For the List & Label trial version you should not set this option.

LL_OPTIONSTR_LIST_PRJEXT

The file extension for a list project. Default "lst".

LL_OPTIONSTR_LIST_PRVEXT

The file extension for the bitmap of a list project that will be shown in the File Open dialog. Default "lsv".

LL_OPTIONSTR_LIST_PRNEXT

The file extension for the printer definition file of a list project. Default "lsp".

LL_OPTIONSTR_LLFILEDESCR

Sets the description for List & Label preview files for the "save as" dialog in the preview.

LL_OPTIONSTR_LLXPATHLIST

This option defines the extension modules (LLX files) to be loaded. You must pass a list of file paths, separated by semicolons, for the extension modules that you want to use in your application.

The following extension modules are loaded automatically by default, i.e. whenever opening a job or setting this option:

CMLL26PW.LLX
CMLL26HT.LLX
CMLL26EX.LLX
CMLL26OC.LLX

Additionally, for the Professional and Enterprise Edition:

CMLL26BC.LLX

These files are loaded from the DLL's path.

You can use Wildcards ("?", "*") to load multiple modules simultaneously.

To suppress loading of a default extension, pass its file name preceded by a "^", e.g. "^CMLL26PW.LLX". To suppress all default extensions, pass "^*" as first "filename".

When this parameter is used for *LlGetOptionString()* you will get a list of available extension modules (for example "CMLL26PW.LLX;CMLL26HT.LLX").

If debug mode is switched on, List & Label will issue the rules and tell you why which module has been loaded or unloaded.

LL_OPTIONSTR_MAILTO

Can be used to preset the address of the recipient when sending the preview file from the preview. Multiple recipients can be separated by ";".

LL_OPTIONSTR_MAILTO_CC

Can be used to preset the address of a CC recipient when sending the preview file from the preview. Multiple recipients can be separated by ";".

LL_OPTIONSTR_MAILTO_BCC

Can be used to preset the address of a BCC recipient when sending the preview file from the preview. Multiple recipients can be separated by ";".

LL_OPTIONSTR_MAITO_SUBJECT

Can be used to preset the subject when sending the preview file from the preview.

LL_OPTIONSTR_NULLVALUE

Can be used to preset the representation of a NULL value at print time. Default value: empty("").

LL_OPTIONSTR_PREVIEWFILENAME

Can be used to preset the name of the preview file. By default, the files are created in the project file's directory or an alternative directory (see LlPreviewSetTempPath()). The default file name is <Project file name>.LL. This option can be used to preset another name that replaces the <Project file name> part.

LL_OPTIONSTR_PRINTERALIASLIST

Allows you to define printer alias tables, i.e. tables that define which printers are to be used if any one of the "default project" printers is not available.

To delete the table, pass NULL or an empty string ("") to this option.

For each printer, you can provide a translation table with the old printer and one or more replacement printers. You can do this by calling this function more than once, or by issuing multiple definitions for the individual printers, separated by a line break "\n".

A table is defined by the line format:

"old printer=new printer 1[;new printer 2[;...]]"

so for example

"\\server\eti=\\server\eti1;\\server_eti2"

"\\server\a4fast=\\server\standard"

This list will cause List & Label to try the alias list "\\server\eti1;\\server_eti2" (in that order) if the printer "\\server\eti" is not available, until a printer is available or the list is finished. The original paper format will be used for the replacement printers. The parameters are not case-sensitive.

LL_OPTIONSTR_PROJECTPASSWORD

Encrypts the project files to protect them against unauthorized use. The password given here is used for the encryption. The password itself is not stored in the project, so do not forget it!

You can store encrypted projects in unencrypted format in the Designer by pressing a shift key when you save it. A password dialog will pop up and allow you to enter the original password. This is useful for debugging or support cases.

The maximum password length is 5 characters in the range of 1 to 255 (ASCII code), resulting in 40-bit encryption. The password is not (!) absolutely secure, as it is passed by an API. However, the barrier to stealing project files is quite high.

LL_OPTIONSTR_SAVEAS_PATH

The passed parameter will be used as default path for "save as" from the preview. The path may contain path and file name.

LL_OPTIONSTR_SHORTDATEFORMAT

The string used to convert a date in a string in:

```
date$(<Date>, "%x")
```

and for automatic type conversion (*LlExprEval()*, *Concat$()*)

Format and Default: See Windows API *GetLocaleInfo(LOCALE_USER_DEFAULT, LOCALE_SSHORTDATE,...)*

LL_OPTIONSTR_THOUSAND

This represents the string that is used as thousands separator in the *fstr$()* function.

The default is the value of the user settings in the system, but this will be set to the respective locale value on *LL_OPTION_LCID*.

LL_OPTIONSTR_VARALIAS

This option enables you to localize the field and variable names for the Designer. The provided alias will be displayed within the Designer instead of the field/variable name. Only the original names will be stored when saving the file. The project can thus be localized by supplying suitable alias names. The option string needs to be set for each name that should be localized in the form "<alias>=<original name>", e.g.

```
LlSetOptionString(hJob, LL_OPTIONSTR_VARALIAS, "Vorname=FirstName");
LlDefineVariable(hJob, "FirstName", "John");
```

in order to pass a variable "FirstName" that should be displayed as "Vorname" in the Designer.

To delete all passed alias names, just use

```
LlSetOptionString(hJob, LL_OPTIONSTR_VARALIAS, "");
```

The .NET, OCX and VCL components offer a custom dictionary API that makes using this option even easier. See the components' help file for more information.

Example:

```
HLLJOB    hJob;

LlSetDebug(TRUE);
hJob = LlJobOpen(0);
```

```
// all label projects will be called <somewhat>.label
v = LlSetOptionString(hJob, LL_OPTIONSTR_LABEL_PRJEXT, "label");
// ...
LlJobClose(hJob);
```

See also:

LlGetOption, LlGetOptionString, LlSetOptionString

LlSetPrinterDefaultsDir

Syntax:

```
INT LlSetPrinterDefaultsDir (HLLJOB hJob, LPCTSTR pszDir);
```

Task:

Sets the path of the printer definition file, e.g. to use user-specific printer settings in a network environment.

Parameter:

hJob: List & Label job handle

pszDir: Path name of the directory

Return Value:

Error code

Example:

```
HLLJOB    hJob;
hJob = LlJobOpen(0);
LlSetPrinterDefaultsDir(hJob, "c:\\temp\\user");
if (LlPrintStart(hJob, LL_PROJECT_LIST, "c:\\test.lst",
LL_PRINT_NORMAL) == 0)
{
   <... etc ...>
   LlPrintEnd(hJob);
}
else
   MessageBox(NULL, "Error", "List & Label", MB_OK);
LlJobClose(hJob);
```

See also:

LlSetPrinterToDefault, LlPrintStart, LlPrintWithBoxStart, LlPrintCopyPrinterConfiguration, LlPrintSetPrinterInPrinterFile

LlSetPrinterInPrinterFile

Syntax:

```
INT LlSetPrinterInPrinterFile (HLLJOB hJob, UINT nObjType,
   LPCTSTR pszObjName, INT nPrinter, LPCTSTR pszPrinter,
   _PCDEVMODE pDM);
```

Task:

Replaces a printer in a printer configuration file by a new one or allows you to set special printer parameters

Parameter:

hJob: List & Label job handle

nObjType: *LL_PROJECT_LABEL*, *LL_PROJECT_CARD* or *LL_PROJECT_LIST*

pszObjName: Project name with file extension

nPrinter: Printer index (0: range with "Page() == 1" [will be created automatically if necessary], 1: default range, -1: creates only the default range and deletes other ranges that may exist).

If the project contains multiple layout regions you can use indices starting from 99, where 99 will set the printer for all regions, 100 for the first, 101 for the second and so on.

pszPrinter: Printer name

pDM: Address of new DEVMODE structure. If NULL, the default settings of the printer will be used.

Return Value:

Error value

Hints:

This function allows you to define the printer that will be used for printing. If the printer configuration file does not exist, it will be created. By "oring" the project type with LL_PRJTYPE_OPTION_-FORCEDEFAULTSETTINGS you can force the printer's default settings for the print job.

As the printer configuration file will be used by *LlPrint[WithBox]Start()*, the function must be called before this function.

The DEVMODE structure is defined in the Windows API help file.

Due to the possibility to define layout regions in the Designer the practical benefit of this function has been quite limited. We recommend to use the LL object model according to chapter "Using the DOM-API (Professional/Enterprise Edition Only)" to access the layout regions and the associated printers.

The default printer in the preview can be set with LL_OPTION_FORCE_DEFAULT_PRINTER_IN_PREVIEW.

Example:

```
HLLJOB    hJob;
hJob = LlJobOpen(0);
LlSetPrinterInPrinterFile(hJob,  LL_PROJECT_LABEL, "test.lbl", -1,
   "Label Printer", NULL);
```

```
<... etc ...>
LlJobClose(hJob);
```

See also:

LlSetPrinterToDefault, LlPrintStart, LlPrintWithBoxStart, LlPrintCopyPrinterConfiguration, LlPrintSetPrinterDefaultsDir, GetPrinterFromPrinterFile

LlSetPrinterToDefault

Syntax:

```
INT LlSetPrinterToDefault (HLLJOB hJob, UINT nObjType,
   LPCTSTR lpszObjName);
```

Task:

Deletes the printer definition file, so that List & Label uses the default printer set in the project the next time the project is used.

Parameter:

hJob: List & Label job handle

nObjType: *LL_PROJECT_LABEL*, *LL_PROJECT_CARD* or *LL_PROJECT_LIST*

lpszObjName: The file name of the project with file extension

Return Value:

Error code

Hints:

The default printer in the preview can be set with LL_OPTION_FORCE_DEFAULT_PRINTER_IN_PREVIEW.

Example:

```
HLLJOB   hJob;

hJob = LlJobOpen(0);

LlSetPrinterToDefault(hJob, LL_PROJECT_LIST, "test.lst");
if (LlPrintStart(hJob, LL_PROJECT_LIST, "test.lst",
   LL_PRINT_NORMAL) == 0)
{
   <... etc ...>
   LlPrintEnd(hJob);
}
else
   MessageBox(NULL, "Error", "List & Label", MB_OK);
LlJobClose(hJob);
```

See also:

LlSetPrinterDefaultsDir, LlPrintStart, LlPrintWithBoxStart

LlViewerProhibitAction

Syntax:

```
INT LlViewerProhibitAction (HLLJOB hJob, INT nMenuID);
```

Task:

Removes buttons from the preview.

Parameter:

hJob: List & Label job handle

nMenuID: ID of the button you wish to remove. The IDs of the menu items in List & Label can be found in the file MENUID.TXT in your List & Label installation.

Return value:

Error code

Hints:

A 0 as menu ID clears the list of menu items to be suppressed.

See also:

LlPreviewDisplay, LlPreviewDisplayEx

LlXGetParameter

Syntax:

```
INT LlXGetParameter (HLLJOB hJob, INT nExtensionType,
   LPCTSTR pszExtensionName, LPCTSTR pszKey, LPTSTR pszBuffer,
   UINT nBufSize);
```

Task:

Gets parameters from a specific extension module.

Parameter:

hJob: List & Label job handle

nExtensionType: Type of extension

Value	Meaning
LL_LLX_EXTENSIONTYPE_EXPORT	Export module
LL_LLX_EXTENSIONTYPE_BARCODE	2D barcode module

pszExtensionName: Name of the extension ("HTML", "RTF", "PDF417", ...)

pszKey: Name of the parameter

pszBuffer: Pointer to a buffer

nBufSize: Size of the buffer

Return value:

Error code

Hints:

The keys known by the extension modules are specific to them. Please refer to the documentation for the respective module.

See chapter "Important Remarks on the Function Parameters of DLLs" concerning the buffer return value.

See also:

LlXSetParameter

LlXSetParameter

Syntax:

```
INT LlXSetParameter (HLLJOB hJob, INT nExtensionType,
   LPCTSTR pszExtensionName, LPCTSTR pszKey, LPCTSTR pszValue);
```

Task:

Sets parameters in a specific extension module.

Parameter:

hJob: List & Label job handle

nExtensionType: Type of extension

Value	Meaning
LL_LLX_EXTENSIONTYPE_EXPORT	Export module
LL_LLX_EXTENSIONTYPE_BARCODE	2D barcode module

pszExtensionName: Name of the extension ("HTML", "RTF", "PDF417", ...)

pszKey: Name of the parameter

pszValue: Value of the parameter

Return value:

Error code

Hints:

The keys known by the extension modules are specific to them. Please refer to the documentation for the respective module.

Using this function, you can preset certain options of a module, for example the path of the output file.

See also:

LIXGetParameter

6.2 Callback Reference

LL_CMND_DRAW_USEROBJ

Task:

Tells the program to draw the object defined by the user.

Activation:

```
LlDefineVariableExt(hJob, <Name>, <Content>, LL_DRAWING_USEROBJ,
<Parameter>);
LlDefineFieldExt(hJob, <Name>, <Content>, LL_DRAWING_USEROBJ,
<Parameter>);
```

or

```
LlDefineVariableExt(hJob, <Name>, <Content>,
LL_DRAWING_USEROBJ_DLG,<Parameter>);
```

Parameters:

Pointer to an *scLIDrawUserObj* structure:

_nSize: Size of the structure, *sizeof(scLIDrawUserObj)*

_lpszName: Name of the variable assigned to the object

_lpszContents: Text contents of the variable which is assigned to the object. This value is only valid if the variable has been defined by *LIDefineVariableExt()*, otherwise the *_hPara* value is valid.

_lPara: lPara of the variable which is assigned to the object (*LL_DRAWING_USEROBJ* or *LL_DRAWING_USEROBJ_DLG*). Refers to the 4th parameter of the call *LIDefineVariableExt()*.

_lpPtr: *lpPtr* of the variable which is assigned to the object. Refers to the 5th parameter of the call *LIDefineVariableExt()*.

_hPara: Handle contents of the variable which is assigned to the object. This value is valid if the variable has been defined by *LIDefineVariableExtHandle()*, otherwise the value *_lpszContents* is valid.

_bIsotropic: TRUE: the object should be drawn undistorted FALSE: the drawing should be fitted into the rectangle

_lpszParameters: 1) for user-defined objects as table field: NULL 2) for *LL_DRAWING_USEROBJ*: Pointer to an empty string 3) for *LL_DRAWING_USEROBJ_DLG*: Pointer to the string the programmer has returned at *LL_CMND_EDIT_USEROBJ*.

_hPaintDC: Device Context for the printout

_hRefDC: Device Context for references

_rcPaint: Rectangle in which the object should be drawn. The mapping mode is in the normal drawing units, mm/10, inch/100 or inch/1000.

_nPaintMode: 1: on Designer-preview 0: on Printer/Multi-page-preview

Return Value (_lResult):

0

Hints:

In this callback no List & Label function may be called which will produce output (*LlPrint()*, etc.))! Functions like *LlPrintGetCurrentPage()*, *LlPrintGetOption()* or *LlPrintEnableObject()* are allowed.

See: Hints for the use of GDI-objects

Example:

```
case LL_CMND_DRAW_USEROBJ:
   pSCD = (PSCLLUSEROBJ)pSC->_lParam;
   FillRect(pSCD->_hPaintDC, pSCD->_rcPaint,
GetStockObject(atoi(lpszContents)));
   break;
```

LL_CMND_EDIT_USEROBJ

Task:

Requests the program to start an object-specific dialog through which the user can enter and change the corresponding presentation parameters.

Activation:

```
LlDefineVariableExt(hJob,<Name>,<Contents>, LL_DRAWING_USEROBJ_DLG,
<Parameter>);
```

Parameters:

Meaning of the parameter lParam:

Pointer to an *scLlEditUserObj* structure:

_nSize: Size of the structure, *sizeof(scLlEditUserObj)*

_lpszName: Name of the variable which is assigned to the object

_lPara: *lPara* of the variable assigned to the object (*LL_DRAWING_USEROBJ* or *LL_DRAWING_USEROBJ_DLG*). Is identical to the 4th parameter of the call *LlDefineVariableExt()*.

_lpPtr: *lpPtr* of the variable assigned to the object. This refers to the 5th parameter of the call *LlDefineVariableExt()*.

_hPara: Handle-contents of the variable assigned to the object. This value is only valid if the variable has been defined by *LIDefineVariableExtHandle()*, otherwise the value *_lpszContents* is valid.

_bIsotropic: TRUE: the object should be drawn undistorted. FALSE: the drawing should be fitted optimally into the rectangle

_hWnd: Window-handle of the dialog. This should be taken as parent-handle of your dialogs.

_lpszParameters: Pointer to a buffer with the maximum size *_nParaBufSize*.

_nParaBufSize: Size of the buffer allocated by List & Label.

Return Value:

0

Hints:

This callback is sent if objects that are set to a variable of type *LL_DRAWING_USEROBJ_DLG* need to be edited.

In this callback no List & Label function may be called which produces output (*LIPrint()*, etc.) Functions like *LIPrintGetCurrentPage()*, *LIPrintGetOption()* or *LIPrintEnableObject()* are allowed.

See: Hints on the use of GDI-objects.

The editing of the *_bIsotropic* flag is optional, as this can be set by the user in the calling dialog. If you change this flag, the change will be adopted by List & Label.

_lpszParameter points to string in which the values entered in the last dialog call are stored. You can copy your parameter string into the buffer when it is smaller or the same size as the buffer. Otherwise, you can change the pointer value to a pointer that points to your data. The problem of a longer parameter string is that it cannot be released by List & Label if it is an allocated storage area. (Basic principle: you can pass up to 1024 characters. The string cannot be extended, superfluous characters are cut off).

The characters permitted in the parameter string are all printable characters, i.e. characters with codes > = 32 (' ').

Example:

```
case LL_CMND_EDIT_USEROBJ:
   pSCE = (PSCLLEDITUSEROBJ)pSC->_lParam;

   lpszNewParas = MyDialog(pSCE->_hWnd, ..., );

   if (strlen(lpszNewParams) < pSCE->_lpszParameters)
      strcpy(pSCE->_lpszParameters, lpszNewParas);
   else
```

```
      pSCE->_lpszParameters = lpszNewParas;
   break;
```

LL_CMND_ENABLEMENU

Task:

Allows the host application to disable menu items

Activation:

Always activated

Parameters:

Meaning of the parameter lParam:

lParam: menu handle

Hints:

This callback is called when *List* & Label changes or updates the menu. The application can then enable or disable menu items previously inserted by *LL_CMND_MODIFYMENU*.

Example:

```
case LL_CMND_ENABLEMENU:
   if (<whatever>)
      EnableMenuItem(hMenu, IDM_MYMENU, MF_ENABLED|MF_BYCOMMAND);

    else
      EnableMenuItem(hMenu, IDM_MYMENU,
MF_DISABLED|MF_GRAYED|MF_BYCOMMAND);
   break;
```

LL_CMND_EVALUATE

Task:

Asks the program for the interpretation of the contents of the function *External$()* in an expression.

Activation:

While printing, when using an *External$()* function.

Parameters:

lParam is a pointer to an *scLIExtFct* structure:

_nSize: Size of the structure, *sizeof(scLIExtFct)*

_lpszContents: Parameter of the function *External$()*

_bEvaluate: TRUE if the contents are to be evaluated FALSE if only a syntax-test is to be carried out.

_szNewValue: Array where the result is stored as a zero-terminated string. Default: empty

_bError: TRUE: error occurred. FALSE: no error occurred. Default: FALSE

_szError: Array where a possible error definition can be stored, which can be requested later with *LlExprError()*. This text is also displayed to the user in the Designer during the automatic syntax check in case of an error.

Return Value (_lResult):

0 (always)

Hints:

If, for example, the expression in a formula is

```
Name + ", " + External$(Name + ", " + forename)
```

then the parameter is the evaluated result of the formula 'Name + ", " + forename', in this case for example 'Smith, George'.

Important: the return fields must be zero-terminated and may not exceed the maximum length (16385 characters incl. termination for the return value, 128 characters incl. zero-termination for the error string).

LL_CMND_GETVIEWERBUTTONSTATE

Task:

Using this callback, List & Label asks the application about button states of the preview's toolbar buttons.

Activation:

Always activated

Parameters:

HIWORD(lParam) = Tool button ID

LOWORD(lParam) = State defined by List & Label

Return Value (_lResult):

New State	Meaning
0	no change
1	enabled
2	disabled
-1	hidden

Hints:

This function will be called by the preview by List & Label. The IDs of the menu items in List & Label can be found in the file MENUID.TXT of your List & Label installation.

Example:

```
case LL_CMND_GETVIEWERBUTTONSTATE:
   switch (HIWORD(lParam))
   {
      case 112:
         // don't allow one-page print:
         return(-1);
   }
   break;
```

LL_CMND_HELP

Task:

Enables the programmer to use an external help system instead of List & Label's own help system.

Activation:

```
LlSetOption(hJob, LL_OPTION_CALLBACKMASK, <other Flags> | LL_CB_HELP);
```

Parameters:

HIWORD(lParam):

Value	Meaning
HELP_CONTEXT	*LOWORD(lParam)* is then the context number of the help topic
HELP_INDEX	the user wants to see the index of the help file
HELP_HELPONHELP	the user queries the help summary

Return Value (_lResult):

0: Return to List & Label to display its help
1: List & Label should do nothing

Example:

```
case LL_CMND_HELP:
   WinHelp(hWnd, "my.hlp", HIWORD(lPara), LOWORD(lPara));
   pSC._lResult = 1;
   break;
```

LL_CMND_MODIFYMENU

Task:

Allows the application to modify List & Label's menu. This callback is supported for reasons of compatability, to extend the Designer the use of *LlDesignerAddAction()* is recommended.

Activation:

Always activated

Parameters:

Meaning of the parameter lParam:

lParam: menu handle

Return Value:

Ignored, always 0

Hints:

This function is called when List & Label created its menu. The application can add or delete menu items.

The IDs of the menu items in List & Label can be found in the file MENUID.TXT of your List & Label installation. User-defined menus should use IDs above 10100.

This callback is only included for compatibility reasons, to expand the Designer preferably use *LlDesignerAddAction()*.

Example:

```
case LL_CMND_MODIFYMENU:
   DeleteMenu(_hMenu, IDM_HELP_CONTENTS, MF_BYCOMMAND);
   DeleteMenu(_hMenu, IDM_HELP_INDEX, MF_BYCOMMAND);
   DeleteMenu(_hMenu, IDM_HELP_HELPONHELP, MF_BYCOMMAND);
   break;
```

LL_CMND_OBJECT

Task:

Enables the programmer to draw something before or after List & Label into or near the object rectangle or to hide the object during printing.

This function allows many modifications to objects and is the so-called "do-it-all" for object representations.

Activation:

```
LlSetOption(hJob, LL_OPTION_CALLBACKMASK, <other Flags> |
LL_CB_OBJECT);
```

Parameters:

lParam points to an scLIObject structure:

_nSize: Size of the structure, sizeof(scLIObject)

_nType: Type of object:

Object	Meaning
LL_OBJ_TEXT	Text
LL_OBJ_RECT	Rectangle
LL_OBJ_LINE	Line object
LL_OBJ_BARCODE	Barcode object
LL_OBJ_DRAWING	Drawing object
LL_OBJ_TABLE	Table
LL_OBJ_RTF	RTF object
LL_OBJ_TEMPLATE	Template bitmap
LL_OBJ_ELLIPSE	Ellipse/Circle

_lpszName: Name of the object. Either the name given in the Designer or a text like "TABLE (<Rectangle measures>)" - the text which is printed in the status line of the Designer for this object if it is selected.

_bPreDraw: TRUE for a call before List & Label draws the object. FALSE for a call after List & Label has drawn the object.

_hPaintDC: Device Context for the print

_hRefDC: Device Context for references

_rcPaint: Rectangle in which the object is drawn. The mapping mode is in the normal drawing units, mm/10, inch/100 or inch/1000.

Return Value (_lResult):

Value of *_bPreDraw*	*_lResult*
TRUE	0: okay 1: object is not to be drawn (in this case hidden)
FALSE	always 0

Hints:

In this callback no List & Label function may be called which will produce output (*LlPrint()*, etc.)! Functions like *LlPrintGetCurrentPage()* or *LlPrintGetOption()* are allowed. See: Hints on the use of GDI objects.

This function is called twice per object, once with *_bPreDraw* = TRUE, then with *_bPreDraw* = FALSE.

_rcPaint may vary between these calls if the object size becomes smaller (text, table object) or the appearance condition does not match!

_bPreDraw = TRUE:

Use: you can draw an individual background or hide the object.

If you change *_rcPaint,* these modifications will have consequences for the size of the object, as the object is drawn by List & Label in the given rectangle.

_bPreDraw = FALSE:

Use: you can draw an individual background and/or shade, as only then is the true size of the object known.

The rectangle *_rcPaint* is the correct object rectangle. If you change *_rcPaint* then this affects the linked objects, as the data from *_rcPaint* is used as object rectangle, which might influence the coordinates of spatially linked objects!

Example:

```
case LL_CMND_OBJECT:
   pSCO = (PSCLLOBJECT)pSC->_lParam;
   if (pSCO->_nType == LL_OBJ_RECT &&
      pSCO->_bPreDraw == FALSE)
   {
      FillRect(pSCO->_hPaintDC, pSCF->_rcPaint,
         GetStockObject(LTGRAY_BRUSH);
   }
   break;
```

LL_CMND_PAGE

Task:

Allows the programmer to place additional output on the page. This is useful for printing labels, for example, as in this way you can "paint" additional information onto a page (page number, printout time, "demo" text, individual watermarks etc...)

Activation:

```
LlSetOption(hJob, LL_OPTION_CALLBACKMASK, <other flags> | LL_CB_PAGE);
```

Parameters:

lParam points to an scLIPage structure:

_nSize: Size of structure, sizeof(scLIPage)

_bDesignerPreview: TRUE if the call takes place from the Designer preview FALSE if the call takes place during the WYSIWYG-preview or print.

_bPreDraw: TRUE for a call, before List & Label draws the page. FALSE for a call after List & Label has finished the page.

_bDesignerPreview: TRUE if the call takes place from the Designer preview FALSE if the call takes place during the WYSIWYG-preview or print.

_hPaintDC: Device Context for the print

_hRefDC: Device Context for references

Return Value:

0

Hints:

In this callback no List & Label function may be called which will produce output as a result (*LlPrint()*, etc.)! Functions like *LlPrintGetCurrentPage()*, *LlPrintGetOption()* or *LlPrintEnableObject()* are allowed.

See: Hints on the use of GDI-objects.

This function is called twice per page, once with *_bPreDraw* = TRUE, then with *_bPreDraw* = FALSE.

The page size can be determined by the function *GetWindowExt()*. Don't forget: use the hRefDC!

If you change the window origin of the hRefDC for *_bPreDraw* = TRUE with *SetWindowOrg(),* this affects the whole page. In this way you can define a different margin for even/odd pages. This relocation only affects the WYSIWYG viewer and printout, not the Designer preview.

Example:

```
case LL_CMND_PAGE:
   pSCP = (PSCLLPAGE)pSC->_lParam;
   if (pSCP->_bPreDraw && (LlPrintGetCurrentPage(hJob) % 2) == 1)
      SetWindowOrg(pSCP->_hPaintDC, -100, 0);
   break;
```

LL_CMND_PROJECT

Task:

Enables the programmer to place additional drawings in a label or file card project (an individual label, for example).

This callback only occurs with label and file card projects. With list objects, it would be identical to *LL_CMND_PAGE*.

Activation:

```
LlSetOption(hJob, LL_OPTION_CALLBACKMASK,
      <other Flags> | LL_CB_PROJECT);
```

Parameters:

lParam points to an scLIProject structure:

_nSize: Size of the structure, sizeof(scLIProject)

_bPreDraw: TRUE for a call before List & Label draws the page. FALSE for a call after List & Label has drawn the page.

_bDesignerPreview: TRUE if the call takes place from the Designer preview. FALSE if the call takes place during the WYSIWYG preview or print.

_hPaintDC: Device Context for the print

_hRefDC: Device Context for references

_rcPaint: Rectangle in which the object should be drawn. The mapping mode is in the normal drawing units, mm/10, inch/100 or inch/1000.

Return Value:

0

Hints:

In this callback no List & Label function may be called which will produce output (*LlPrint()*, etc.)! Functions like *LlPrintGetCurrentPage()*, *LlPrintGetOption()* or *LlPrintEnableObject()* are allowed.

See: Hints on the use of GDI-objects.

This function is called twice per page, once with *_bPreDraw* = TRUE, then with *_bPreDraw* = FALSE.

Example:

```
case LL_CMND_PROJECT:
   pSCP = (PSCLLPROJECT)pSC->_lParam;
   if (pSCP->_bPreDraw)
   {
      FillRect(pSCL->_hPaintDC, pSCL->_rcPaint,
         GetStockObject(LTGRAY_BRUSH);
   }
   break;
```

LL_CMND_SAVEFILENAME

Task:

Notification that the user has saved the project in the Designer. The file name is passed.

Parameters:

lParam points to the zero-terminated file name.

Return Value (_lResult):

0

Example:

```
case LL_CMND_SAVEFILENAME:
   pszLastFilename = (LPCTSTR)lParam;
```

LL_CMND_SELECTMENU

Task:

Notification that a menu has been selected.

Activation:

Always activated.

Parameters:

lParam: Menu ID of the menu item (negative ID if called from a toolbar button!).

Return Value (_lResult):

TRUE, if List & Label shall not try to execute the command associated with the menu ID (usually if the menu item has been inserted by the application) FALSE otherwise

Example:

```
case LL_CMND_SELECTMENU:
   if (lParam == IDM_MYMENU)
   {
      // execute custom code
      return (TRUE);
   }
   break;
```

LL_CMND_TABLEFIELD

Task:

Enables the programmer to modify the coloring of individual table fields.

Activation:

```
LlSetOption(hJob, LL_OPTION_TABLE_COLORING, LL_COLORING_PROGRAM)
```

In this way, the control of the coloring in tables is left to your program (the corresponding settings in the table characteristic dialog of the Designer won't appear).

or

```
LlSetOption(hJob, LL_OPTION_TABLE_COLORING,LL_COLORING_DONTCARE)
```

With this command, List & Label lets your program draw the background first of all, then it draws the background again (if allowed) with the field pattern defined in the Designer. This allows a kind of cooperation between the programmer and the user.

Parameters:

lParam points to an sclITableField structure:

_nSize: Size of structure, sizeof(sclITableField)

_nType: Type of field:

Value	Meaning
LL_TABLE_FIELD_HEADER	Field is in the header line
LL_TABLE_FIELD_BODY	Field is in the data line
LL_TABLE_FIELD_GROUP	Field is in group header line
LL_TABLE_-FIELD_GROUPFOOTER	Field is in group footer line
LL_TABLE_FIELD_FILL	Field is the filling area when the table has a fixed size and there is some free space below the last data line
LL_TABLE_FIELD_FOOTER	Field is in the footer line

_hPaintDC: Device Context for the print and in the following callback definitions

_hRefDC: Device Context for the references

_rcPaint: Rectangle in which the field is to be drawn. The mapping mode is in the normal drawing units, mm/10, inch/100 or inch/1000.

_nLineDef: Number of line definition to be drawn.

_nIndex: Field index, 0-based (the first column has the index 0, the second 1, etc.)

_rcSpacing: Cell distances

_pszContents: This parameter provides the (evaluated) content of the cell currently being rendered, e.g. "Smith" for a column that contains a field Person.Lastname. You can use this parameter to handle certain values specifically.

Return Value:

0

Hints:

In this callback no List & Label functions may be called which will produce output (*LlPrint()*, etc.)!

If you select a GDI object in these DCs or make other changes, e.g. change the mapping mode, you should reverse the changes before ending the routine. Hint: the API functions *SaveDC()*, *RestoreDC()* can help considerably for complex changes (the used functions are Windows API function).

Example:

```
case LL_CMND_TABLEFIELD:
   pSCF = (PSCLLTABLEFIELD)pSC->_lParam;
   if (pSCF->_nIndex == 1)
   {
      FillRect(pSCF->_hPaintDC, pSCF->_rcPaint,
         GetStockObject(LTGRAY_BRUSH);
   }
   pSC._lResult = 0;
   break;
```

LL_CMND_TABLELINE

Task:

Enables the programmer to modify the coloring of individual table lines, e.g. to produce your own zebra mode (every other line).

Activation:

```
LlSetOption(hJob, LL_OPTION_TABLE_COLORING, LL_COLORING_PROGRAM)
```

In this way the control of the coloring in tables is left to your program (the corresponding setting possibilities won't appear).

```
LlSetOption(hJob, LL_OPTION_TABLE_COLORING,LL_COLORING_DONTCARE)
```

With this command, List & Label lets your program draw the background first of all, then it draws the background with the field background defined in the Designer, when required again. This allows a kind of cooperation between the programmer and the user.

Make sure to set the LL_CB_TABLELINE flag via LL_OPTION_CALLBACKMASK in order to receive this notification.

Parameters:

lParam points to an scLITableLine structure:

_nSize: Size of the structure, sizeof(scLITableLine)

_nType: Type of field:

Value	Meaning
LL_TABLE_LINE_HEADER	Header line
LL_TABLE_LINE_BODY	Data line
LL_TABLE_LINE_GROUP	Group header
LL_TABLE_- LINE_GROUPFOOTER	Group footer

LL_TABLE_LINE_FILL	Filling area when the table has a fixed size and there is some free space below the last data line
LL_TABLE_LINE_FOOTER	Footer line

_hPaintDC: Device Context for the printout

_hRefDC: Device Context for references

_rcPaint: Rectangle in which the line is to be drawn. The mapping mode is in the normal drawing units, mm/10, inch/100 or inch/1000.

_nPageLine: Line index. Marks the 0-based line number on this page.

_nLine: Line index. Marks the 0-based line number of the line in the whole print.

_nLineDef: Number of line definition to be drawn.

_bZebra: TRUE, when the user chooses zebra mode in the Designer.

_rcSpacing: Cell distances

Return Value:

0

Hints:

In this callback no List & Label function may be called which will produce output (*LIPrint()*, etc.)!

See: Hints on the use of GDI-objects

Example:

```
case LL_CMND_TABLELINE:
   pSCL = (PSCLLTABLELINE)pSC->_lParam;
   if ((pSCL->_nPageLine % 2) == 1)
   {
      FillRect(pSCL->_hPaintDC, pSCL->_rcPaint,
         GetStockObject(LTGRAY_BRUSH);
   }
   pscCallback->_lReply = 0;
   break;
```

LL_CMND_VARHELPTEXT

Task:

Assigns a context help string for a variable or field. This string is displayed if the variable/field is selected in the expression wizard.

Activation:

Always activated

Parameters:

lParam: points to a string containing the variable or fieldname

Return Value:

_lReply must point to the description string. Caution: this must remain valid after return of this function, so do not use an automatic stack variable.

Example:

```
case LL_CMND_VARHELPTEXT:
   sVariableDescr = (LPCSTR) pscCallback->_lParam;
   // Check routines for variable
   strcpy(szHelpText, "Variable x for task y");
   pscCallback->_lReply = (LPARAM)szHelpText;
   break;
```

LL_INFO_METER

Task:

Notification that a (possibly) lengthy operation is taking place.

Activation:

Always activated

Parameters:

lParam points to a structure of type scLIMeterInfo:

_nSize: Size of the structure

_hWnd: Handle of the List & Label main window

_nTotal: Total count of objects

_nCurrent: Index of object currently being processed

_nJob: Job ID, tells you what LL is doing:

Value	Meaning
LL_METERJOB_SAVE	saving the objects
LL_METERJOB_LOAD	loading the objects
LL_METERJOB_-CONSISTENCYCHECK	internal consistency check

Hints:

By using this callback, the host application can implement a wait dialog box. We suggest using this callback if the object count exceeds 200 objects to reduce unnecessary screen flickering. To get a percentage value, use *MulDiv(100, _nCurrent, _nTotal)*.

Example:

```
// functions used here for a meter dialog must be replaced by own
functions
case LL_INFO_METER:
{
   scLlMeterInfo* pMI = (scLlMeterInfo*)lParam;
   static HLLJOB   hMeterJob = 0;
   // is actual version?
   if (pMI->_nSize == sizeof(scLlMeterInfo))
   {
      // do I have to do something?
      if (pMI->_nTotal > 0)
      {
         // get parent window handle for Dialog
         HWND  hWndParent = pMI->_hWnd ? pMI->_hWnd : hwndMyFrame;
         // start:
         if (pMI->_nCurrent == 0)
         {
            // open meter bar with 0%!
            hMeterJob =  WaitDlgStart(hWndParent, "wait a moment", 0);
         }
         else
         {
            // end:
            if (pMI->_nCurrent == pMI->_nTotal)
            {
               // end meter bar!
               WaitDlgEnd(hMeterJob);
            }
            else
               // somewhere in between 0 and 100
            {
               // set meter value to MulDiv(100, _nCurrent, _nTotal)
               WaitDlgSetText(hMeterJob, "still working...",
                     MulDiv(100, pMI->_nCurrent, pMI->_nTotal));
            }
         }
      }
   }
}
break;
```

LL_INFO_PRINTJOBSUPERVISION

Task:

Can be used to monitor a print job after it is passed to the spooler.

Parameter:

lParam contains an address of a struct of type *scLlPrintJobInfo*:

_nSize: Size of the structure

_hLlJob: Job handle of the job that started the print

_szDevice: Printer name

_dwJobID: Job ID (not the job ID of the printer but a global one created by List & Label)

_dwState: Combination of state flags (JOB_STATUS_-constants in WINSPOOL.H)

Hints:

Please make sure to set *LL_OPTION_NOPRINTJOBSUPERVISION* to *FALSE* to enable this callback.

The detail depth depends on the print spooler.

The dwState flags are defined as follows:

```
#define JOB_STATUS_PAUSED               0x00000001
#define JOB_STATUS_ERROR                0x00000002
#define JOB_STATUS_DELETING             0x00000004
#define JOB_STATUS_SPOOLING             0x00000008
#define JOB_STATUS_PRINTING             0x00000010
#define JOB_STATUS_OFFLINE              0x00000020
#define JOB_STATUS_PAPEROUT             0x00000040
#define JOB_STATUS_PRINTED              0x00000080
#define JOB_STATUS_DELETED              0x00000100
#define JOB_STATUS_BLOCKED_DEVQ         0x00000200
#define JOB_STATUS_USER_INTERVENTION    0x00000400
#define JOB_STATUS_RESTART              0x00000800
```

LL_NTFY_DESIGNERPRINTJOB

Task:

Via callback *LL_NTFY_DESIGNERPRINTJOB* List & Label informs you about the task that has to be performed. This callback will always be called up in the context of the designer thread (this is the thread, from which *LIDefineLayout()* was called).

Parameters:

The callback parameter is a pointer to a *scLIDesignerPrintJob structure*:

_nUserParam: value you set *LL_OPTION_DESIGNERPREVIEWPARAMETER* to or assigned *LL_OPTION_DESIGNEREXPORTPARAMETER* to.

_pszProjectName: Name of the project to print. This parameter is only valid with the command "START", otherwise NULL.

_pszOriginalProjectFileName: Name of the original project. This parameter is only valid with the command "START", otherwise NULL. It is necessary, so that relative paths and the function ProjectPath$() are correctly evaluated by List & Label.

_nPages: Maximum number of pages to be output. This will have to be passed after print start via

```
LlPrintSetOption(hJob,LL_PRNOPT_LASTPAGE,_nPages)
```

to the print job. If _nPages has the value "0", this means, that the print should not be limited.

_nFunction: Operations to be performed. There are four different operations: Start, Break, Finalize and Status query.

As there are two groups of operation (EXPORT and PREVIEW), this gives 8 constants:

LL_DESIGNERPRINTCALLBACK_PREVIEW_START
LL_DESIGNERPRINTCALLBACK_PREVIEW_ABORT
LL_DESIGNERPRINTCALLBACK_PREVIEW_FINALIZE
LL_DESIGNERPRINTCALLBACK_PREVIEW_QUEST_JOBSTATE
LL_DESIGNERPRINTCALLBACK_EXPORT_START
LL_DESIGNERPRINTCALLBACK_EXPORT_ABORT
LL_DESIGNERPRINTCALLBACK_EXPORT_FINALIZE
LL_DESIGNERPRINTCALLBACK_EXPORT_QUEST_JOBSTATE

_hWnd: Window handle. The meaning of this structure member is described below.

_hEvent: Event handle, needed for communication and synchronization of your application with List & Label.

_pszExportFormat: Preselected export format (required in Ribbon mode only), see chapter "Direct Print and Export From the Designer".

_bWithoutDialog: Print/export without dialog (required in Ribbon mode only), see chapter "Direct Print and Export From the Designer".

Return Value:

Return *LL_DESIGNERPRINTTHREAD_STATE_RUNNING*, if your thread is working otherwise *LL_DESIGNERPRINTTHREAD_STATE_STOPPED*.

Hints:

See chapter "Direct Print and Export From the Designer".

LL_NTFY_EXPRERROR

Task:

Notifies the application of an expression error found by the parser.

Activation:

Always active

Parameters:

lParam: Pointer to an error text

Return Value:

0

Hints:

As *LlExprError()* does not reliably return an incorrect formula after a call to *LlPrintStart()*, this event can be used to collect errors and present them to the user when *LlPrintStart()* fails because of formula errors.

LL_NTFY_EXPRERROR_EX

Task:

Notifies the application of an expression error found by the parser.

Activation:

Always active

Parameters:

The callback parameter is a pointer to a *scLlNtfyExprErrorEx structure*:

_nSize: Size of the structure.

_pszExpr: Erroneous expression.

_pszError: Error text.

_pvHierarchy: Pointer to a VARIANT array, that contains the hierarchy of the error location.

Return Value:

0

Hints:

Since *LlExprError()* only leads to limited success when loading a project (also for printing) (since the internal formula parser may have already parsed a completely different formula when calling the function and thus may no longer have an error "active"), you can collect error messages and their error location via this callback and then report them to the user after *LlPrintStart()*.

LL_NTFY_FAILSFILTER

Task:

Notification that the data record which has just been passed to List & Label was not printed, as it did not comply with the filter requirement.

Activation:

Always active

Parameters:

Meaning of the parameter lParam: none

Return Value:

0

Hints:

In this callback, no List & Label function may be called which will produce output (*LlPrint()*, etc.)!

Serves to set a global variable; but can be made superfluous with *LlPrintDidMatchFilter()*.

Example:

```
case LL_NTFY_FAILSFILTER:
   bFails = TRUE;
   break;
```

LL_NTFY_VIEWERBTNCLICKED

Task:

Notification that a button has been pressed in the preview window.

Activation:

Always activated

Parameters:

lParam: Tool button ID

Return Value (_lResult):

_lResult	Meaning
1	ignore button (action usually done by host application)
0	execute default button function

Hints:

For the IDs please refer to the callback *LL_CMND_GETVIEWERBUTTONSTATE*.

Example:

```
case LL_NTFY_VIEWERBTNCLICKED:
   switch (lParam)
   {
      case 112:
         // beep on one-page print
         // and don't execute it!
         MessageBeep(-1);
         return(1);
   }
   break;
```

LL_NTFY_VIEWERDRILLDOWN

Task:

Notification that a drilldown action should be processed.

Activation:

```
LlSetOption(hJob, LL_OPTION_DRILLDOWNPARAMETER,
    (LPARAM)&oMyDrillDownParameters);
```

Parameters:

lParam points to a structure *scLIDrillDownJob*:

_nSize: Size of the structure

_nFunction: Sets the task:

Task	Meaning
LL_DRILLDOWN_START	Start
LL_DRILLDOWN_FINALIZE	Finalize

_nUserParameter: Value passed with *LL_OPTION_DRILLDOWNPARAMETER*

_pszTableID: Points to a string containing the name of the child table

_pszRelationID: Points to a string containing the name of the relation between child and parent table

_pszSubreportTableID: Points to a string containing the name of the child table.

_pszKeyField: Points to a string containing the name of the key field of the parent table. If the relation contains different key fields, the result is tab delimited. Please note the description of the function *LlDbAddTableRelationEx()*.

_pszSubreportKeyField: Points to a string containing the name of the key field of the child table. If the relation contains different key fields, the result is tab delimited. Please note the description of the function *LlDbAddTableRelationEx()*.

_pszKeyValue: Points to a string containing the contents of the key field of the parent table. If the relation contains different key fields, the result is tab delimited. Please note the description of the function *LlDbAddTableRelationEx()*.

_pszProjectFileName: Name of the project file to be processed.

_pszPreviewFileName: Name of the preview file that has to be created.

_pszTooltipText: Points to a string containing the tool tip text when hovering over a table entry, that can trigger a drilldown report.

_pszTabText: Points to a string containing the tab text, if the user wants a drilldown report shown in a separate tab.

_hWnd: Window handle to show own dialogs (window handle of the preview control).

_nID: Unique drilldown job ID, should not be mistaken with the List & Label print job. To make a unique assignement, in the *FINALIZE* task this ID contains the value that has been assigned in the *START* task.

_hAttachInfo: This parameter is needed for *LlAssociatePreviewControl()* to attach the viewer. Additionally the flags LL_ASSOCIATEPREVIEW-CONTROLFLAG_DELETE_ON_CLOSE and LL_ASSOCIATEPREVIEW-CONTROLFLAG_HANDLE_IS_ATTACHINFO must be used. Further information can be found in chapter "Printing Relational Data".

Return Value:

Return a value that is unique for the entire runtime of the application.

Hints:

This callback will always be called in the context of the preview thread, regardless if initiated from designer or preview print.

Example:

See chapter "Printing Relational Data".

LL_QUERY_DESIGNERACTIONSTATE

Task:

Via this callback List & Label checks the state of the user defined Actions (see *LlDesignerAddAction()*). You can then, depending on requirements, enable or disable the actions.

Activate:

Always active

Parameter:

HIWORD(lParam): the (user defined) ID for the action

LOWORD(lParam): Status setting as set by Designer

Return value (_lResult):

Value	Meaning
1	Action is active
2	Action is not active

Example:

```
case LL_QUERY_DESIGNERACTIONSTATE:
   _lResult = (bEnabled? 1 : 2);
   break;
```

LL_QUERY_EXPR2HOSTEXPRESSION

Task:

Via this callback List & Label asks the host to translate a filter expression (as configured in the Designer) to data source native syntax. The callback is triggered multiple times for each part of the filter expression as soon as the application calls to *LlPrintDbGetCurrentTableFilter()*. This callback can be used e.g. to translate a List & Label filter to an SQL query.

Activate:

Always active, triggered by a call to *LlPrintDbGetCurrentTableFilter()*.

Parameter:

lParam points to a structure ***scLLEXPR2HOSTEXPR***. For readability, the prefix "_p" means that it's a pointer to the argument, but in the description, we call it the argument.

_nSize: Size of the structure

_pszTableID: Table this expression belongs to.

_nType: Type of element that needs to be translated. For most operations, simply set _pvRes to the resulting statement for the operation. If – for example – _nType is LLEXPR2HOSTEXPR_ARG_BINARY_OPERATOR_ADD, the typical return value would be `_pvRes = _pv1 + _T("+") + _pv2`.

Value	Meaning
LLEXPR2HOSTEXPR_ARG_BOOLEAN	a boolean value
LLEXPR2HOSTEXPR_ARG_TEXT	a text value. If you need to parametrize your query to avoid SQL injection attacks, set the _pvName member to a parameter name (on entry, _pvName contains a unique, consecutive integer index that you can use for the name if needed) and set _pvRes to the resulting text. The parameter values will be returned in the corresponding variant passed to *LlPrintDbGetCurrentTableFilter().*
LLEXPR2HOSTEXPR_ARG_NUMBER	a numeric value. _pvArg1->vt is either VT_I4 for an integer, oder VT_R8 for a floating-point value.
LLEXPR2HOSTEXPR_ARG_DATE	a date value
LLEXPR2HOSTEXPR_ARG_UNARY_OPERATOR_SIGN	the sign operator
LLEXPR2HOSTEXPR_ARG_UNARY_OPERATOR_NEGATION	the negation operator
LLEXPR2HOSTEXPR_ARG_BINARY_OPERATOR_ADD	the "+" operator
LLEXPR2HOSTEXPR_ARG_BINARY_OPERATOR_SUBTRACT	the "-" operator
LLEXPR2HOSTEXPR_ARG_BINARY_OPERATOR_MULTIPLY	the "*" operator
LLEXPR2HOSTEXPR_ARG_BINARY_OPERATOR_DIVIDE	the "/" operator
LLEXPR2HOSTEXPR_ARG_BINARY_OPERATOR_MODULO	the "%" operator
LLEXPR2HOSTEXPR_ARG_LOGICAL_OPERATOR_XOR	the logical xor operator

LLEXPR2HOSTEXPR_ARG_LOGICAL_OPERATOR _OR	the logical or operator
LLEXPR2HOSTEXPR_ARG_LOGICAL_OPERATOR_AND	the logical and operator
LLEXPR2HOSTEXPR_ARG_RELATION_EQUAL	the "=" operator
LLEXPR2HOSTEXPR_ARG_RELATION_NOTEQUAL	the "<>" operator
LLEXPR2HOSTEXPR_ARG_RELATION_LARGERTHAN	the ">" operator
LLEXPR2HOSTEXPR_ARG_RELATION_LARGEREQUAL	the ">=" operator
LLEXPR2HOSTEXPR_ARG_RELATION_SMALLERTHAN	the "<" operator
LLEXPR2HOSTEXPR_ARG_RELATION_SMALLEREQUAL	the "<=" operator
LLEXPR2HOSTEXPR_ARG_FUNCTION	a designer function. _pvName contains the function name, _pv1…_pv4 contain the function's arguments.
LLEXPR2HOSTEXPR_ARG_FIELD	a database field. Depending on the target syntax, it might be necessary to escape or frame an identifier name.

_pvRes: a VARIANT to receive the resulting expression. Set the pointer to NULL or the VARIANT type to VT_EMPTY, if no suitable translation is available. The whole (or in case of an AND operator, the current branch of the) expression is not translated.

_nArgs: Number of arguments. This member is important for functions with optional arguments.

_pvName: Name of function to translate. This member can also be set to handle query parameters (see above).

_pv1: Depending on _nType (see above), the first argument of a function or the left-hand side of an operator.

_pv2: Depending on _nType (see above), the second argument of a function or the right-hand side of an operator.

_pv3: The third argument of a function.

_pv4: The fourth argument of a function.

Return value:

Value	Meaning
0	Translation was not handled or cannot be handled. The whole (or in case of an AND operator, the current branch of the) expression is not translated.
1	Translation was handled exactly.
2	Translation was handled inexactly; the result will contain more records than appropriate. In this case, List & Label will run its own filtering in addition in order to filter the exceeding records.

6.3 Managing Preview Files

6.3.1 Overview

The preview print contained in List & Label writes the preview data into a file. This file can be archived for later use, sent to another user who can look at it or print it without any loss of quality.

All data is stored in one file. Using the optional compression that you can switch on using

```
LlSetOption(hJob, LL_OPTION_COMPRESSSTORAGE, 1)
```

the file size can be reduced by up to 2/3. Compression slows down the print process but is convenient, for example, if you wish to present data on the Internet for download or preview using our OCX.

The file has the extension ".LL". We do not provide any information about its inner structure, and we recommend that you do no rely on any details you may find out! This is intentional, as we have our own API to access the data contained in it, so that there is no advantage for you in seeing inside the file. We wish to be free to change the format whenever necessary, without having conflicts with existing software.

6.3.2 The Preview API

You do not need to worry about the details of the preview files - the API functions *LlStgsysxxx()* do that for you.

All of these functions are exported by the C?LS26.DLL. This DLL, which you will usually distribute with external viewers, is as small as possible. If you wish to use this DLL via an import library, you need to link to the C?LS26.LIB file. In some programming languages, it is sufficient to include the respective declaration file.

LlStgsysAppend

Syntax:

```
INT LlStgsysAppend (HLLSTG hStg, HLLSTG hStgToAppend);
```

Task:

Append another preview job to the current storage file.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

hStgToAppend: Handle of the preview file to append.

Return value:

<0: Error code
= 0: okay

Hints:

This function needs both (!) preview files to be in the *LL_STG_STORAGE* format.

If the file to append contains a backside page it will only be used if the original file doesn't already contain such a page itself.

Of course, the current storage format may not be opened with bReadOnly = TRUE!

Example:

```
HLLSTG hStgOrg;
HLLSTG hStgAppend;

hStgOrg = LlStgsysStorageOpen("c:\\test\\label1.ll", FALSE, FALSE);
hStgAppend = LlStgsysStorageOpen("c:\\test\\label2.ll", FALSE, TRUE);
LlStgsysAppend(hStgOrg, hStgAppend);
LlStgsysClose(hStgsysOrg);
LlStgsysClose(hStgsysAppend);
```

LlStgsysConvert

Syntax:

```
INT LlStgsysConvert (HLLSTG hStg, LPCTSTR pszDstFilename,
   LPCTSTR pszFormat);
```

Task:

Converts a preview file to another format.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

pszDstFilename: Name of the target file. It is also possible to use the placeholder %d (e.g. "page %d"). This is important e.g. for converting JPEGs, because without the placeholder only one page will result.

pszFormat: Target format. Valid values are:

"TIFF" (also "PICTURE_MULTITIFF")
"JPEG" (also "PICTURE_JPEG")
"PNG" (also "PICTURE_PNG")
"EMF"
"TTY"
"PDF"
"XPS"
"PRN"
"TXT"
"LL"

This parameter allows you to declare a semicolon-separated list with further export options. You will find the accepted values in the chapter "The Export Modules". Please note that not all options can be supported with this API. An example is the parameters "PDF;PDF.EncryptionFile=1".

Additional to the mentioned parameters above the following parameters can be used:

Value	Meaning
PageIndexRange	Analog to the print options dialog a range for pages can be set.
JobIndexRange	Analog to the print options dialog a range for the job can be set.
IssueIndexRange	Analog to the print options dialog a range for the issues can be set.

An example of this is the use of " PDF;Export.PageIndexRange=2-3".

With this, only pages 2 and 3 are converted to PDF.

The export to PRN creates a file that is specially prepared for the given printer (parameter "PRN.Device="). This file can be output on the printer by copying it directly. Therefore the printer name (device name) must be explicitly defined.

Return value:

<0: Errorcode
= 0: okay

Hints:

-

Example:

```
HLLSTG hStgOrg;

hStgOrg = LlStgsysStorageOpen("c:\\test\\label1.ll", "",
            FALSE, TRUE);
LlStgsysStorageConvert(hStgOrg, "c:\\test\\label2.pdf", "PDF");
LlStgsysStorageClose(hStgOrg);
```

See also:

LlStgsysStorageOpen, LlStgsysStorageConvert

LlStgsysDeleteFiles

Syntax:

```
void LlStgsysDeleteFiles (HLLSTG hStg);
```

Task:

Erases the preview file(s).

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

Return value:

<0: Error code
= 0: okay

Hints:

This function erases the preview file(s). The only call that makes sense after this call is *LlStgsysStorageClose()*.

See also:

LlStgsysStorageOpen, LlStgsysStorageClose

LlStgsysDestroyMetafile

Syntax:

```
INT LlStgsysDestroyMetafile (HANDLE hMF);
```

Task:

Releases the metafile handle.

Parameter:

hMF: (enhanced) metafile handle

Return value:

<0: Error code
= 0: okay

Example:

See LlStgsysGetPageMetafile

See also:

LlStgsysGetPageMetafile

LlStgsysDrawPage

Syntax:

```
void LlStgsysDrawPage (HLLSTG hStg, HDC hDC, HDC hPrnDC,
   BOOL bAskPrinter, _PCRECT pRC, INT nPageIndex, BOOL bFit,
   LPVOID pReserved);
```

Task:

Paints a preview page to a screen or printer device.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

hDC: DC for printing (usually a printer or screen device). Can be NULL (see below).

hPrnDC: Reference DC which can be used to get the unprintable area etc. For a screen DC, this is the (default) printer DC, for a printer DC it is the same as hDC above. Can be NULL (See below).

bAskPrinter: If hPrnDC is NULL, this flag defines whether the user is asked about the printer for the reference DC. If it is TRUE, he will be asked, if it is FALSE, the default printer will be used.

pRC: Points to a RECT structure containing the device coordinates for printing. If this is NULL, the printer's values will be used. Must not be NULL when printing to a non-printer DC!

nPageIndex: Page index (1..*LlStgsysGetPageCount()*)

bFit: Defines whether the print should fit into the area (TRUE) or whether the original size should be kept (FALSE), although the latter might result in clipped output due to differences in paper size, unprintable area etc..

pReserved: NULL

Return value:

Error code

Hints:

If hDC is NULL, it will be set to hPrnDC after the reference DC has been created.

See Also:

LlStgsysPrint, LlStgsysStoragePrint

LlStgsysGetAPIVersion

Syntax:

```
int LlStgsysGetAPIVersion (HLLSTG hStg);
```

Task:

Returns the version of the Stgsys API.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

Return value:

The version number of the Stgsys API in List & Label C?LL26.DLL and C?LS26.DLL

Hints:

The current version number is 26.

This function should be used to test the API version. Newer APIs might have a larger set of functions available, older ones less.

See also:

LlStgsysGetFileVersion

LlStgsysGetFilename

Syntax:

```
int LlStgsysGetFilename (HLLSTG hStg, INT nJob, INT nFile,
   LPTSTR pszBuffer, UINT nBufSize);
```

Task:

Can be used to get the "real" name(s) of the preview file(s). If a path has been provided to *LlStgsysStorageOpen()* this path will also be included.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

nJob: Job Index: 1: first Job,... (1..*LlStgsysGetJobCount()*)

nFile: Page number

Value	Meaning
-1	Management file
0	Printer configuration file

>0	Page Metafile for this page (1.. *LlStgsysGetPageCount()*)

lpszBuffer: Initialized buffer for the file name with file extension

nBufSize: Size of the buffer

Return value:

Error code

Hints:

The nFile Parameter distinguishes the type of file for which the name is to be returned.

In the case of *LL_STG_STORAGE*, its name is returned regardless of the nFile parameter, as this is the one and only file that contains all information.

Example:

```
CString  sFilename, sOutput;
LlStgsysGetFilename(hStg, 1, -1, sFilename.GetBuffer(_MAX_PATH),
    _MAX_PATH);
sFilename.ReleaseBuffer();
sOutput = CString(_T("View of file ")) + sFilename;
```

See also:

LlStgsysGetJobCount

LlStgsysGetFileVersion

Syntax:

```
int LlStgsysGetFileVersion (HLLSTG hStg);
```

Task:

Returns the version of the preview file.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

Return value:

The version number of the preview file and the type:

Value	Meaning
Bits 0..7	The current version number is 26

Hints:

This call is also very important for finding out about properties of the storage file and for dealing with possible differences.

See also:

LlStgsysGetAPIVersion

LlStgsysGetJobCount

Syntax:

```
INT LlStgsysGetJobCount (HLLSTG hStg);
```

Task:

Returns the number of jobs stored in the preview.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

Return value:

>0: Number of jobs
<0: Error code

Example:

see LlStgsysSetJob

See also:

LlStgsysStorageOpen

LlStgsysGetJobOptionStringEx

Syntax:

```
INT LlStgsysGetJobOptionStringEx (HLLSTG hStg, LPCTSTR pszKey,
   LPTSTR pszBuffer, UINT nBufSize);
```

Task:

Returns project parameter values.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

pszKey: Option name

pszBuffer: Address of buffer for the value

nBufSize: Size of buffer (incl. string termination)

Return value:

<0: Error code
= 0: okay

Hints:

The available option names depend on the parameters which the creating application has made available via *LlPrintSetProjectParameter()* or *LlSetDefaultProjectParameter()* as PUBLIC. Note that you need to prefix these parameters with "ProjectParameter" in order to query the values. See also chapter "Project Parameters".

See also:

LlStgsysSetJobOptionStringEx

LlStgsysGetJobOptionValue

Syntax:

```
INT LlStgsysGetJobOptionValue (HLLSTG hStg, INT nOption);
```

Task:

Returns certain numerical parameters for the current job.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

nOption: Chooses the meaning of the return value

Return value:

>=0: Value
<0: Error code

Hints:

These values are invaluable if you wish to create your own preview and print management, especially if the destination printers are different from the original.

To get the correct value, set the job with *LlStgsysSetJob()* before calling this API function.

nOption can have the following values:

LS_OPTION_BOXTYPE

Returns the style of the meter box used at the time of the preview print (and which should also be used during printing). This is one of the constants *LL_BOXTYPE_xxx* (see *LlPrintWithBoxStart()*), or -1 if no box had been used (*LlPrintStart()*).

LS_OPTION_UNITS

Returns the units chosen for the project, see *LL_PRNOPT_UNIT*.

LS_OPTION_PRINTERCOUNT

Number of printers used

See also:

LlStgsysSetJob

LlStgsysGetLastError

Syntax:

```
INT LlStgsysGetLastError (HLLSTG hStg);
```

Task:

Returns the error code of the last call to a *LlStgsys()*-API function.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

Return value:

<0: Error code
= 0: no error

Hints:

Can be used for functions that return NULL as return value in case of an error (i.e. *LlStgsysGetPageMetafile()*).

See also:

LlStgsysGetPageMetafile

LlStgsysGetPageCount

Syntax:

```
INT LlStgsysGetPageCount (HLLSTG hStg);
```

Task:

Returns the number of pages in the current job.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

Return value:

>0: Number of pages
<0: Error code

Hints:

The page numbers (the numbers that can be written on the paper!) can be queried by calling *LlStgsysGetPageOptionValue()* with the parameter *LS_OPTION_PAGENUMBER*.

Example:

See LlStgsysSetJob

See also:

LlStgsysSetJob, LlStgsysJobGetOptionValue

LlStgsysGetPageMetafile

Syntax:

```
HANDLE LlStgsysGetPageMetafile (HLLSTG hStg, INT nPageIndex);
```

Task:

Returns an enhanced metafile handle that can be used to display or print page data.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

nPageIndex: Page index (1..*LlStgsysGetPageCount()*)

Return value:

NULL: error; else: handle of (enhanced) metafile

Hints:

The handle needs to be released using *LlStgsysDestroyMetafile()*.

Example:

Excerpt from the code of *LlStgsysDrawPage()*:

```
HANDLE       hMF;
BOOL         b16bit;

hMF = LlStgsysGetPageMetafile(hStg, nPageIndex);
if (hMF == NULL)
{
   hMF = LlStgsysGetPageMetafile16(hStg, nPageIndex);
}
if (hMF == NULL)
   ret = LL_ERR_STG_CANNOTGETMETAFILE;
else
{
   POINT ptPixels;
   POINT ptPixelsOffset;
   POINT ptPixelsPhysical;
   POINT ptPixelsPerInch;
```

```
   ptPixels.x = LlStgsysGetPageOptionValue(hStg, nPageIndex,
      LS_OPTION_PRN_PIXELS_X);
   ptPixels.y = LlStgsysGetPageOptionValue(hStg, nPageIndex,
      LS_OPTION_PRN_PIXELS_Y);
   ptPixelsOffset.x = LlStgsysGetPageOptionValue(hStg, nPageIndex,
      LS_OPTION_PRN_PIXELSOFFSET_X);
   ptPixelsOffset.y = LlStgsysGetPageOptionValue(hStg, nPageIndex,
      LS_OPTION_PRN_PIXELSOFFSET_Y);
   ptPixelsPhysical.x = LlStgsysGetPageOptionValue(hStg, nPageIndex,
      LS_OPTION_PRN_PIXELSPHYSICAL_X);
   ptPixelsPhysical.y = LlStgsysGetPageOptionValue(hStg, nPageIndex,
      LS_OPTION_PRN_PIXELSPHYSICAL_Y);
   ptPixelsPerInch.x = LlStgsysGetPageOptionValue(hStg, nPageIndex,
      LS_OPTION_PRN_PIXELSPERINCH_X);
   ptPixelsPerInch.y = LlStgsysGetPageOptionValue(hStg, nPageIndex,
      LS_OPTION_PRN_PIXELSPERINCH_Y);
   <Paint Metafile>
   LlStgsysDestroyMetafile(hMF);
}
```

See also:

LlStgsysDestroyMetafile

LlStgsysGetPageOptionString

Syntax:

```
INT LlStgsysGetPageOptionString (HLLSTG hStg, INT nPageIndex,
   INT nOption, LPTSTR pszBuffer, UINT nBufSize);
```

Task:

Returns character strings that are stored in the preview file.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

nPageIndex: Page index (1..*LlStgsysGetPageCount()*)

nOption: Chooses the meaning of the return value

pszBuffer: Address of the buffer for the string

nBufSize: Length of the buffer (including the terminating 0-character)

Return value:

Error code

Hints:

You can use the following values for nOption:

LS_OPTION_PROJECTNAME

Returns the name of the project file that has been used to create this page

LS_OPTION_JOBNAME

Returns the name of the job (see *LlPrintWithBoxStart()*)

LS_OPTION_USER

Returns the user-specific string (see *LlStgsysSetPageOptionString()*)

LS_OPTION_CREATION

Creation date/time

LS_OPTION_CREATIONAPP

Application that created this file

LS_OPTION_CREATIONDLL

DLL that created this file

LS_OPTION_CREATIONUSER

User and computer name of the person that created this file

LS_OPTION_PRINTERALIASLIST

See also LL_OPTIONSTR_PRINTERALIASLIST: this represents the printer alias list valid at the time of the creation of the preview file. This is one string, lines separated by a line break "\n".

LS_OPTION_USED_PRTDEVICE

Returns the device of the original printer (for example "HP LaserJet 4L")

See also:

LlStgsysGetPageOptionValue, LlStgsysSetPageOptionString

LlStgsysGetPageOptionValue

Syntax:

```
INT LlStgsysGetPageOptionValue (HLLSTG hStg, INT nPageIndex,
   INT nOption);
```

Task:

Returns page-dependent information.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

nPageIndex: Page index (1..*LlStgsysGetPageCount()*)

nOption: Chooses the meaning of the return value

LS_OPTION_PAGENUMBER

Returns the page number of the selected page.

LS_OPTION_COPIES

Returns the number of copies that the page should be printed with.

LS_OPTION_PRN_ORIENTATION

Returns the page orientation (*DMORIENT_PORTRAIT* or *DMORIENT_LANDSCAPE*)

LS_OPTION_PHYSPAGE

Returns whether the project should be printed using the physical paper size (1) or only the printable area (0).

LS_OPTION_PRN_PIXELSOFFSET_X

Returns the horizontal offset of the printable area in relation to the paper edge (of the original printer).

LS_OPTION_PRN_PIXELSOFFSET_Y

Returns the vertical offset of the printable area in relation to the paper edge (of the original printer).

LS_OPTION_PRN_PIXELS_X

Returns the horizontal size of the printable area (of the original printer).

LS_OPTION_PRN_PIXELS_Y

Returns the vertical size of the printable area (of the original printer).

LS_OPTION_PRN_PIXELSPHYSICAL_X

Returns the horizontal size of the paper (of the original printer).

LS_OPTION_PRN_PIXELSPHYSICAL_Y

Returns the vertical size of the paper (of the original printer).

LS_OPTION_PRN_PIXELSPERINCH_X

Returns the horizontal printer resolution (DPI).

LS_OPTION_PRN_PIXELSPERINCH_Y

Returns the vertical printer resolution (DPI).

LS_OPTION_PRN_INDEX

Returns the index of the printer used for the current page (0 means first page-printer, 1 for the printer for the other pages).

LS_OPTION_ISSUEINDEX

Returns the issue index (1…) of the page.

Return value:

>=0: Value
<0: Error code

Hints:

"Printer" or "original printer" means the printer selected when the preview file was created.

These values are invaluable if you wish to create your own preview and print management, especially if the destination printers are different from the original.

To get the correct value, set the job with *LlStgsysSetJob()* before calling this API function.

See also:

LlStgsysGetPageOptionString

LlStgsysGetPagePrinter

Syntax:

```
INT LlStgsysGetPagePrinter (HLLSTG hStg, INT nPageIndex,
   LPTSTR pszDeviceName, UINT nDeviceNameSize, PHGLOBAL phDevmode);
```

Task:

Returns the printer and the settings that would be used for this page.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

nPageIndex: Page index (1..*LlStgsysGetPageCount()*)

pszDeviceName: Pointer to a buffer for the device name

nDeviceNameSize: Size of the buffer

phDevmode: Pointer to a global handle where the DEVMODE structure will be stored. If NULL, the DEVMODE structure is not queried. If a pointer to a handle is passed, it must be a valid global handle or NULL.

Return value:

Error code (LL_ERR_BUFFERTOOSMALL if the device name's buffer is too small)

See also:

LlGetPrinterFromPrinterFile, LlSetPrinterInPrinterFile

Example:

```
HGLOBAL dev(NULL);
TCHAR*pszPrinter = new TCHAR[1024];
int iRet = LlStgsysGetPagePrinter(m_hStgOrg, 1, pszPrinter, 1096,
&dev);
LPVOID pDevmode = GlobalLock(dev);
DEVMODE aDEVMODE = *((DEVMODE*)pDevmode);
…
// tidy-up
GlobalUnlock(dev);
GlobalFree(dev);
```

LlStgsysPrint

Syntax:

```
HLLSTG LlStgsysStoragePrint (HLLSTG hStg, LPCTSTR pszPrinterName1,
   LPCTSTR pszPrinterName2, INT nStartPageIndex, INT nEndPageIndex,
   INT nCopies, UINT nFlags, LPCTSTR pszMessage, HWND hWndParent);
```

Task:

Prints pages from an open preview file job

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

pszPrinterName1: Name of the printer to be used for the first page (can be NULL, see below)

pszPrinterName2: Name of the printer to be used for the following pages (can be NULL, see below)

nStartPageIndex: Index of the first page to be printed

nEndPageIndex: Index of the last page to be printed

nCopies: Number of copies

nFlags: A combination of the following flags:

Flag	Meaning
LS_PRINTFLAG_FIT	Fits the print to the printable area of the printer
LS_PRINTFLAG_-STACKEDCOPIES	Print copies for each page, not the job (111222333 instead of 123123123)

LS_PRINTFLAG_-TRYPRINTERCOPIES	Try to make hardware copies if possible
LS_PRINTFLAG_METER	Show a meter dialog
LS_PRINTFLAG_-ABORTABLEMETER	Show a meter dialog which has a "Cancel" button
LS_PRINTFLAG_SHOWDIALOG	Shows a printer select dialog
LS_PRINTFLAG_FAX	Required for output on fax printer
LS_PRINTFLAG_IGNORE_PROJECT_TRAY	Paper bin will be ignored
LS_PRINTFLAG_IGNORE_PROJECT_DUPLEX	Duplex will be ignored
LS_PRINTFLAG_IGNORE_PROJECT_COLLATION	Page collation will be ignored
LS_PRINTFLAG_IGNORE_PROJECT_EXTRADATA	Printer specific settings will be ignored

pszMessage: Will be shown in the title of the optional meter dialog and is also used as document name for the print job. If NULL, the entry from the preview file (parameter of *LIPrintWithBoxStart()*) is used.

hWndParent: Window handle to be used as parent for the meter dialog

Return Value:

Error code

Hints:

Use this API routine if you want an easy way to print a page range from the current storage job. If a printer name is NULL, List & Label tries to get the printer and its settings from the values stored in the preview file (i.e. the printer settings selected during creation). If no printer with the specified device name is present, the default printer is selected.

You need to set the job (*LIStgsysSetJob()*) before you call this function.

See Also:

LIStgsysPrint, LIStgsysSetJob

LlStgsysSetJob

Syntax:

```
INT LlStgsysSetJob (HLLSTG hStg, INT nJob);
```

Task:

Sets the job index for the following API calls

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

nJob: Job index (1..*LlStgsysGetJobCount()*)

Return value:

Error code

Hints:

The following API calls that return job-dependent data will use this job index to return the corresponding values.

Example:

```
// calculates the total amount of pages
int nPages = 0;
INT nJob;
for (nJob = 1; nJob<LlStgsysGetJobCount(hStg); ++ nJob)
{
   LlStgsysSetJob(hStg, nJob);
   nPages + = LlStgsysGetPageCount(hStg);
}
```

See also:

LlStgsysGetJobCount

LlStgsysSetJobOptionStringEx

Syntax:

```
INT LlStgsysSetJobOptionStringEx (HLLSTG hStg, LPCTSTR pszKey,
   LPCTSTR pszValue);
```

Task:

Sets project parameter values.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

pszKey: Option name

pszValue: Value

Return value:

<0: Error code
= 0: okay

Hints:

Can be used to write values into the preview file (unless it was opened as READ-ONLY). Do not use reserved option names starting with "LL.".

See also:

LlStgsysGetJobOptionStringEx

LlStgsysSetPageOptionString

Syntax:

```
INT LlStgsysSetPageOptionString (HLLSTG hStg, INT nPageIndex,
   INT nOption, LPCTSTR pszBuffer);
```

Task:

Set string values.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

nPageIndex: Page index (1..*LlStgsysGetPageCount()*)

nOption: Chooses the meaning of the contents value

Value	Meaning
LS_OPTION_JOBNAME	New job name
LS_OPTION_USER	A user-specific value The user can insert a user-specific string, for example a user name, print date etc. into the storage file.

pszBuffer: Address of a 0-terminated string

Return value:

Error code

Hints:

Of course, the storage file may not be opened with bReadOnly = TRUE!

Example:

```
LlStgsysSetJob(hStg, 1);
LlStgsysSetPageOption(hStg, 1, LS_OPTION_USER, "Letters A-B");
```

See also:

LlStgsysGetPageOptionValue

LlStgsysSetUILanguage

Syntax:

```
INT LlStgsysSetUILanguage (HLLSTG hStg, INT nLanguage);
```

Tasks:

Sets the UI language.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

nLanguage: LCID of the language.

Return value:

0: okay, <0: Error code

Example:

```
LlStgsysSetJob(hStg,1);
LlStgsysSetUILanguage(hStg,1033);
```

LlStgsysStorageClose

Syntax:

```
void LlStgsysStorageClose (HLLSTG hStg);
```

Task:

Closes the access handle to the preview.

Parameter:

hStg: The handle returned by *LlStgsysStorageOpen()*

See also:

LlStgsysStorageOpen

LlStgsysStorageConvert

Syntax:

```
INT LlStgsysStorageConvert (LPCTSTR pszStgFilename,
   LPCTSTR pszDstFilename, LPCTSTR pszFormat);
```

Task:

Converts a preview file to another format.

Parameter:

pszStgFilename: Name of the preview file

pszDstFilename: Name of the target file

pszFormat: Target format. For valid values and additional options, see LlStgsysConvert().

Return value:

<0: Errorcode
= 0: okay

Hints:

-

Example:

```
LlStgsysStorageConvert("c:\\test\\label2.ll", "c:\\test\\label2.pdf",
"PDF");
```

See also:

LIStgsysStorageOpen, LIStgsysConvert

LlStgsysStorageOpen

Syntax:

```
HLLSTG LlStgsysStorageOpen (LPCTSTR lpszFilename,
   LPCTSTR pszTempPath, BOOL bReadOnly, BOOL bOneJobTranslation);
```

Task:

Opens a preview file

Parameter:

lpszFilename: The name of the project or preview (List & Label does not take account of the extension, as it will always be set to .LL)

pszTempPath: A temporary path (can be NULL or empty)

bReadOnly: TRUE: The file will be opened in read-only mode. FALSE: The file can be written to

bJobTranslation: TRUE: The Stgsys API takes account of multiple jobs and shows you all data as one job. FALSE: You can (and must!) manage multiple jobs yourself

Return value:

Job handle for all other LIStgsys functions, 0 means error

Hints:

If you use a path for temporary data, this will be used as directory for the preview files, otherwise the path of the project file name will be used. This convention is compatible with the calls to *LIPrint(<Project file>)* and *LIPreviewSetTempPath(<Temporary Path>).*

Note that the functions *LIStgsysAppend()* and *LIStgsysSetPageOptionString()* need the file to be opened with bReadOnly = FALSE!

bJobTranslation = TRUE is convenient if you don't want to take account of multiple jobs. If you want to show your users whether the file contains multiple jobs, you need to set this to FALSE and manage a list of jobs with their properties.

See also:

LlStgsysClose

LlStgsysStoragePrint

Syntax:

```
INT LlStgsysStoragePrint (LPCTSTR lpszFilename,
   LPCTSTR pszTempPath, LPCTSTR pszPrinterName1,
   LPCTSTR pszPrinterName2, INT nStartPageIndex,
   INT nEndPageIndex, INT nCopies, UINT nFlags,
   LPCTSTR pszMessage, HWND hWndParent);
```

Task:

Prints pages from an open preview file job

Parameter:

lpszFilename: The name of the project or preview (List & Label does not take account of the extension, as it will always be set to .LL)

pszTempPath: A temporary path (can be NULL or empty)

pszPrinterName1: Name of the printer to be used for the first page (can be NULL, see below)

pszPrinterName2: Name of the printer to be used for the following pages (can be NULL, see below)

nStartPageIndex: Index of the first page to be printed

nEndPageIndex: Index of the last page to be printed

nCopies: Number of copies

nFlags: A combination of the following flags:

Flag	Meaning
LS_PRINTFLAG_FIT	Fits the print to the printable area of the printer
LS_PRINTFLAG_STACKEDCOPIES	Prints copies for each page, not the job (111222333 instead of 123123123)
LS_PRINTFLAG_TRYPRINTERCOPIES	Tries to make copies by printer feature, if possible
LS_PRINTFLAG_METER	Shows a meter dialog

LS_PRINTFLAG_ABORTABLEMETER	Shows a meter dialog which has a "Cancel" button
LS_PRINTFLAG_SHOWDIALOG	Shows a printer select dialog
LS_PRINTFLAG_FAX	Required for output on fax printer

pszMessage: Will be shown in the title of the optional meter dialog and is also used as document name for the print job. If NULL, the entry from the preview file (parameter of *LIPrintStart()*) is used.

hWndParent: Window handle to be used as parent for the meter dialog

Return value:

Error code

Hints:

Use this API routine if you want an easy way to print a page range from a preview file. If a printer name is NULL, List & Label tries to get the printer and its settings from the values stored in the preview file (i.e. the printer settings selected during creation). If no printer with the given device name is present, the default printer is selected.

See also:

LIStgsysPrint

LsMailConfigurationDialog

Syntax:

```
INT LsMailConfigurationDialog (HWND hWndParent, LPCTSTR pszSubkey,
   UINT nFlags, INT nLanguage);
```

Task:

Opens a configuration dialog for the mail parameters. Can be used if the CMMX26.DLL is used for sending export results by mail.

The settings will be saved in the registry under "HKEY_CURRENT_USER-\software\combit\cmbtmx\<pszSubkey>\<User|Computer>".

Parameter:

hWndParent: Parent window handle for the dialog.

pszSubkey: Subkey that is used for saving the values in the registry. You should use your application's executable name (excluding the path and file extension) here. The values will then be set automatically.

Alternatively, a complete registry key like "HKEY_CURRENT_USER\..." or "HKEY_LOCAL_MACHINE\..." can be passed.

nFlags: Any combination of LS_MAILCONFIG_USER and LS_MAILCONFIG_GLOBAL (at least one must be specified). Optionally LS_MAILCONFIG_PROVIDER can be added for storing the transport provider (added/'or'ed). The data of the transport provider are user specific unless the flag LS_MAILCONFIG_USER was not specified.

Value	Meaning
LS_MAILCONFIG_USER	User-specific data
LS_MAILCONFIG_GLOBAL	Computer-specific data
LS_MAILCONFIG_PROVIDER	Provider selection (SMAPI, SMTP, ...)

All data (also the computer specific data) is saved user-specifically – the flags just define a logical separation for the dialog (server settings and user information).

nLanguage: Language for the dialog

Value	Meaning
CMBTLANG_DEFAULT	System language
CMBTLANG_GERMAN	German
CMBTLANG_ENGLISH	English

Other values can be found in the declaration files.

Return value:

Error code

See also:

-

LsMailGetOptionString

Syntax:

```
INT LsMailGetOptionString (HLSMAILJOB hJob, LPCTSTR pszKey,
   LPTSTR pszBuffer, UINT nBufSize);
```

Task:

Queries the email settings from List & Label.

Parameter:

hJob: List & Label email-API job handle

pszKey: Option name. For valid options, see *LsMailSetOptionString()*.

lpszBuffer: Pointer to a buffer for the value.

nBufSize: Size of the buffer.

Return value:

Error code

See also:

LsMailSetOptionString

LsMailJobClose

Syntax:

```
INT LsMailJobClose (HLSMAILJOB hJob);
```

Task:

Close the DLL job.

Parameter:

hJob: List & Label email API job handle

Hints:

This function must be called after using the email functions or when terminating your application. (paired with *LsMailJobOpen()*.

Example:

```
HLSMAILJOB   hMailJob;

hMailJob = LsMailJobOpen(CMBTLANG_DEFAULT);
...
LsMailJobClose(hMailJob)
```

See also:

LsMailJobOpen

LsMailJobOpen

Syntax:

```
INT LsMailJobOpen (INT nLanguage);
```

Task:

Opens a mail job.

Parameter:

nLanguage: language for user interaction

Value	Meaning
CMBTLANG_DEFAULT	System default language
CMBTLANG_GERMAN	German
CMBTLANG_ENGLISH	English

Further constants in the declaration files.

Return value:

A handle, which is necessary for most functions.

A valid value is greater than 0.

Example:

```
HLSMAILJOB   hMailJob;

hMailJob = LsMailJobOpen(0);
```

See also:

LsMailJobClose

LsMailSendFile

Syntax:

```
INT LsMailSendFile (HLSMAILJOB hJob, HWND hWndParent);
```

Task:

Sends an email with the current settings.

Parameter:

hJob: List & Label email API job handle

hWndParent: Parent window handle for the email dialog. If the window handle is "0", no dialog will be shown and the email will be sent without any user activities.

Return value:

Error code

Example:

```
HLSMAILJOB   hMailJob;

hMailJob = LsMailJobOpen(0);
LsMailSetOptionString(hMailJob, "Export.Mail.To",
"test@domainname.de");
LsMailSetOptionString(hMailJob, "Export.Mail.Subject", "Test!");
LsMailSetOptionString(hMailJob, "Export.Mail.AttachmentList",
   "c:\\test.txt");
LsMailSendFile(hMailJob, 0);
LsMailJobClose(hMailJob)
```

See also:

LsMailSetOptionString

LsMailSetOptionString

Syntax:

```
INT LsMailSetOptionString (HLSMAILJOB hJob, LPCTSTR pszKey,
   LPCTSTR pszValue);
```

Task:

Sets various mail settings in List & Label.

Parameter:

hJob: List & Label email API job handle

pszKey: The following values are possible:

Value	Meaning
Export.Mail.To	Recipient address. Multiple recipients can be separated by semicolons.
Export.Mail.CC	This address will receive a carbon copy. Multiple recipients can be separated by semicolons.
Export.Mail.BCC	This address will receive a blind carbon copy. Multiple recipients can be separated by semicolons.
Export.Mail.Subject	Email subject.
Export.Mail.Body	Mail body text.
Export.Mail.Body:text/plain	Mail body text in plain text format. Identical to *Export.Mail.Body*.
Export.Mail.Body:text/html	Mail body text in HTML format (SMTP and XMAPI only). Optional, otherwise the mail will be sent with the text defined in *Export.Mail.Body* or *Export.Mail.Body:text/plain.*
Export.Mail.AttachmentList	Tabulator-separated attachment list

pszValue: new value

Return value:

Error code

Example:

```
HLSMAILJOB   hMailJob;

hMailJob = LsMailJobOpen(0);
LsMailSetOptionString(hMailJob, "Export.Mail.To",
   "test@domainname.com");
...
LsMailJobClose(hMailJob)
```

See also:

LsMailGetOptionString

LsSetDebug

Syntax:

```
void LsSetDebug (BOOL bOn);
```

Task:

Switches the LS-API debug mode.

Parameter:

bOn: If TRUE, the debug mode will be switched on.

Return value:

-

See also:

-

7. The Export Modules

7.1 Overview

In addition to the output to preview file, List & Label offers some other output formats. These output formats can be created by certain special export modules used by List & Label, with the file extension .LLX (= List & Label extension). This List & Label extension interface is designed to allow multiple output formats in a single extension file. The standard output formats provided with List & Label are DOCX, MHTML, PDF, PICTURE_BMP, PICTURE_EMF, PICTURE_JPEG, PICTURE_PNG, PICTURE_MULTITIFF, PICTURE_TIFF, PPTX, PRES, RTF, SVG, TTY, TXT, TXT_LAYOUT, XLS, XHTML, XML and XPS.

The export output formats ("DOCX", "MHTML", "PDF", "PICTURE_BMP", "PICTURE_EMF", "PICTURE_JPEG", "PICTURE_PNG", "PICTURE_MULTITIFF", "PICTURE_TIFF", "PPTX", "PRES", "RTF", "SVG", "TTY", "TXT", "TXT_LAYOUT", "XLS", "XHTML", "XML", "XPS") can be used in addition to the standard output formats media printer ("PRN"), preview ("PRV") and file ("FILE"). The actual export to one of the new formats can be performed analogously to normal printing.

The output formats which are shown to the end user or which are directly used can be specified by your program.

7.2 Programming Interface

7.2.1 Global (De)activation of the Export Modules

List & Label tries to load the export extension module cmll26ex.llx from the main DLLs path by default. All export formats are thus automatically available when passing LL_PRINT_EXPORT as target to *LlPrint(WithBox)Start*.

If you want to deactivate the export modules, use *LL_OPTIONSTR_LLXPATHLIST* and pass the file name preceded by a ^, i.e. "^cmll26ex.llx". The same option may be used to load the module from a different path.

If you want to load the export modules from a different directory, you should also use this option. For example, you can use "c:\programs\<your application>\cmll26ex.llx, to load the export modules from your application directory.

7.2.2 Switching Specific Export Modules On/Off

Using the option *LL_OPTIONSTR_EXPORTS_AVAILABLE,* you can get a string containing all available export media separated by semicolons. This list also contains the standard output formats "PRN", "PRV" and "FILE". The available export formats can be restricted by setting the option *LL_OPTIONSTR_EXPORTS_ALLOWED*. This setting affects the available output formats in the dialog *LlPrintOptionsDialog()*. Please note

that the print destination parameter in *LlPrint[WithBox]Start()* influences the export media as well. You should therefore use *LL_OPTIONSTR_EXPORTS_ALLOWED* after it.

Example of how to enable certain exporters:

```
LlPrintWithBoxStart(..., LL_PRINT_EXPORT, ...);
//Only print to preview and HTML is allowed:
LlSetOptionString(hJob, LL_OPTIONSTR_EXPORTS_ALLOWED,"PRV;HTML");
//...
LlPrintOptionsDialog(...);
```

Example of how to disable the export modules:

```
LlPrintWithBoxStart(..., LL_PRINT_EXPORT, ...);
//Prohibits all export modules:
LlSetOptionString(hJob, LL_OPTIONSTR_EXPORTS_ALLOWED, "PRN;PRV;FILE");
//...
LlPrintOptionsDialog(...);
```

7.2.3 Selecting/Querying the Output Format

The output format can be selected/queried with a parameter for the methods *LlPrint[WithBox]Start()*. The following list shows the different values for this parameter:

Value	Meaning
LL_PRINT_NORMAL	"Printer" output format will be default.
LL_PRINT_PREVIEW	"Preview" output format will be default.
LL_PRINT_FILE	"File" output format will be default.
LL_PRINT_EXPORT	An export module will be set as default output format. After this you could use the method *LlPrintSetOption-String(LL_PRNOPTSTR_EXPORT)* to specify the export module exactly.

You can also use *LlPrintSetOptionString(LL_PRNOPTSTR_EXPORT)* to specify a certain output format, which will also be the default output format in *LlPrintOptionsDialog().*

Example in C++ of how to set the output format to RTF:

```
...
LlPrintWithBoxStart(..., LL_PRINT_EXPORT, ...);
LlPrintSetOptionString(hJob, LL_PRNOPTSTR_EXPORT, "RTF");
LlPrintOptionsDialog(...);
```

If you wish to prohibit the end user from selecting the output format, you could use the option *LL_OPTIONSTR_EXPORTS_ALLOWED* to disable the other formats. Simply specify the output format you wish to force with this option.

The end user can specify the default output format in the Designer using Project > Page Setup. The selected export module will be set by List & Label using the option *LL_PRNOPTSTR_EXPORT*. Your application should take account of this fact by determining the default output format directly or disabling this configuration opportunity in the Designer. Otherwise your end user could be confused when he selects e.g. "RTF" in the Designer, but then finds "HTML" as a default format for printing.

Example of how to take account of a selected export medium (if no selection has been set by the end user in the Designer, "Preview" will be set as default):

```
LlPrintGetOptionString(hJob, LL_PRNOPTSTR_EXPORT, sMedia.GetBuffer(256),
256);
sMedia.ReleaseBuffer();
if (sMedia == "") //no default setting
{
       LlPrintSetOptionString(hJob, LL_PRNOPTSTR_EXPORT, TEXT("PRV"));
}
LlPrintOptionsDialog(...);
```

Example of how to disable the configuration option in the Designer:

```
LlSetOption(hJob, LL_OPTION_SUPPORTS_PRNOPTSTR_EXPORT, FALSE);
//...
LlDefineLayout(...);
```

Use this option to determine which output format was selected by the end user in the *LlPrintOptionsDialog()*.

Example showing how to query the output format:

```
//...
LlPrintOptionsDialog(...);
LlPrintGetOptionString(hJob, LL_PRNOPTSTR_EXPORT, sMedia.GetBuffer(256),
256);
sMedia.ReleaseBuffer();
//...
if (sMedia == "PRV")
{
       LlPreviewDisplay(...);
       LlPreviewDeleteFiles(...); //optional
}
```

7.2.4 Setting Export-specific Options

The export-specific options can be set using *LlXSetParameter()* and queried with *LlXGetParameter()*. These options are export media-specific, therefore the name of the format must be specified for each function call. Options which are supported by all

export modules can be switched simultaneously for all exporters by passing an empty string as exporter name, ex. "Export.ShowResult". The options supported by the export media will be listed in the following chapters.

Some of the options can be modified in the property dialog of the exporter by the end user. These options will then be saved in the P-file by List & Label.

When using the export format "PRV" an export to a preview file can be done. Please note that in for this format only the options Export.File, Export.Path, Export.Quiet and Export.ShowResult are supported.

7.2.5 Export Without User Interaction

Export without user interaction can be performed very easily using the methods already mentioned.

Example:

If you wish to export HTML without user interaction using the file 'Article.lst' and 'c:\temp' as the destination directory, you should use following code:

```
//...
LlXSetParameter(hJob, LL_LLX_EXTENSIONTYPE_EXPORT, "HTML",
       "Export.File", "export.htm");
LlXSetParameter(hJob, LL_LLX_EXTENSIONTYPE_EXPORT, "HTML",
       "Export.Path", "c:\\temp\\");
LlXSetParameter(hJob, LL_LLX_EXTENSIONTYPE_EXPORT, "HTML",
       "Export.Quiet","1");
LlPrintWithBoxStart(hJob, LL_PROJECT_LIST, "Article.lst",
       LL_PRINT_EXPORT, LL_BOXTYPE_BRIDGEMETER, hWnd,
       "Exporting to HTML");
LlPrintSetOptionString(hJob, LL_PRNOPTSTR_EXPORT, "HTML");
//... normal printing loop ...
```

That's all! The meaning of the export-specific options can be found in the following chapters.

7.2.6 Querying the Export Results

To find out which files have been created as an export result, you can use the option *LL_OPTIONSTR_EXPORTFILELIST*. If you query this option using *LlGetOptionString()* after *LlPrintEnd()*, the result will be a semicolon-separated list of all files (including path) generated by List & Label. In the case of an HTML export, the result would be a list of all HTML and JPEG files and in the case of a print to preview, the result would be a single LL preview file.

The files created by the export will not be deleted automatically and should be deleted by your application.

7.3 Programming Reference

7.3.1 PDF Export

Overview

The PDF export module creates documents in the Portable Document Format. This format can be viewed platform-independently with the free Adobe Acrobat Reader® software.

Note on PDF encryption (PDF.Encryption...): In the past few years, nearly all countries' governments did ease restrictions concerning products with encryption. At the moment there's to our knowledge no country which restricts the redistribution and the usage of worldwide accepted standards which are using encryption. Customers, who install List & Label, should be familiar with all local regulations regarding the use of encryption and should take legal advice in order to be informed about the restrictions of the countries they are operating in.

Limitations

Besides others, the following hints and limitations should be considered:

- Fonts are automatically recognized and dynamically embedded if necessary.
- Not all EMF records can be displayed accurately – if you are using complex EMFs, you should pass them as bitmaps or choose "export as picture" in the designer.
- Lines/Frames that are dashed/dotted in the layout may have a different spacing.
- Note for PDF/A:
  - When using form elements in combination with PDF/A, PDF/A conformity cannot be maintained and the form elements are deactivated.
  - All fonts are always embedded.
  - Encryption is not supported.

Programming Interface

You can find a description of all options used in the PDF export module in this chapter. The options can be modified/read using the methods *LIXSetParameter(..."PDF"...)* and *LIXGetParameter(..."PDF"...).*

PDF.Title: Specifies the title of the generated PDF document.

PDF.Subject: Specifies the subject of the generated PDF document. Default: empty.

PDF.Author: Set the Author tag of the PDF file. Default: empty.

PDF.Creator: Set the Creator tag of the PDF file. Default: empty.

PDF.Keywords: Specifies the keywords of the generated PDF document. Default: empty.

PDF.Conformance: Set the PDF version to be used. If encryption is activated (see *PDF.Encryption.EncryptFile*) the encryption strength will be automatically selected. Various options are available, which are explained below.

Value	Meaning
pdf10	PDF version 1.0
pdf11	PDF version 1.1
pdf12	PDF version 1.2
pdf13	PDF version 1.3
pdf14	PDF version 1.4 (corresponds to Acrobat 5)
pdf15	PDF version 1.5
pdf16	PDF version 1.6 (corresponds to Acrobat 7)
pdf17	PDF version 1.7 (ISO 32000-1)
pdf20	PDF version 2.0 (ISO 32000-2)
pdfa1b	PDF/A-1b (ISO 19005-1, Level B compliance)
pdfa1a	PDF/A-1a (ISO 19005-1, Level A compliance)
pdfa2b	PDF/A-2b (ISO 19005-2, Level B compliance)
pdfa2u	PDF/A-2u (ISO 19005-2, Level U compliance)
pdfa2a	PDF/A-2a (ISO 19005-2, Level A compliance)
pdfa3b	PDF/A-3b (ISO 19005-3, Level B compliance)
pdfa3u	PDF/A-3u (ISO 19005-3, Level U compliance)
pdfa3a	PDF/A-3a (ISO 19005-3, Level A compliance)
Default	pdf17

PDF.Encryption.EncryptFile: If this parameter is set, the result file is encrypted. The encryption type is automatically determined by the selected PDF version (see *PDF.Conformance*). There are several other options available, which are explained below.

Value	Meaning
0	Do not encrypt file
1	Encrypt file Encryption type based on the selected PDF version: pdf10, pdfa[x]: no encryption pdf11, pdf12, pdf13: RC4 with a key length of 40 pdf14: RC4 with a key length of 128 pdf15, pdf16, pdf17: AES with a key length of 128 pdf20: AES with a key length of 256

Value	Meaning
Default	0

PDF.Encryption.EnablePrinting: If this parameter is set, the file can be printed even if it is encrypted. Only effective if *PDF.Encryption.EncryptFile* is set to "1".

Value	Meaning
0	Printing is not enabled
1	Printing is enabled
Default	0

PDF.Encryption.EnableChanging: If this parameter is set, the file can be changed even if it is encrypted. Only effective if *PDF.Encryption.EncryptFile* is set to "1".

Value	Meaning
0	Changing is not enabled
1	Changing is enabled
Default	0

PDF.Encryption.EnableCopying: If this parameter is set, the file can be copied to the clipboard even if it is encrypted. Only effective if *PDF.Encryption.EncryptFile* is set to "1".

Value	Meaning
0	Copying is not enabled
1	Copying is enabled
Default	0

PDF.FileAttachments: With this parameter, additional files can be added to the PDF container. Pass them as follows:

<MyAdditionalFile1>|<MyAdditionalFileDescription1>;<MyAdditionalFile2>|<MyAdditionalFileDescription2>

PDF.OwnerPassword: The owner password for the encrypted file. This password is needed to edit the file. If no password is given, a random password will be assigned. We recommend that you always explicitly choose a suitable password.

PDF.UserPassword: The user password for the encrypted file. This password is needed to access the encrypted file. If no password is given, access is possible without a password, but may be limited (see above).

PDF.ExcludedFonts: Allows you to pass a semicolon-separated list of fonts that will never be embedded, regardless of the PDF font mode setting. Default: "Arial".

Note: If "*" is specified, no fonts are embedded, only the name of the fonts contained. This activates the Windows font mapping of the used PDF viewer, which then uses the most suitable font in the system for displaying. Thus the file size can usually be kept very small.

PDF.ZUGFeRDConformanceLevel: Set the ZUGFeRD Conformance Level.

Value	Meaning
BASIC	Simple invoices with structured data. Additional information possible as free text.
EXTENDED	Intersectoral invoice exchange with extended structured data.
COMFORT	Fully automated invoice processing with structured data. Only relevant for ZUGFeRD 1.0. For ZUGFeRD 2.0/2.1 please use "EN 16931".
EN 16931	Fully automated invoice processing with structured data. Only valid for ZUGFeRD 2.0/2.1. Identical with XRechnung 1.0.
BASIC WL	The profile is also included in ZUGFeRD 2.0/2.1 for reasons of consistency between the two standards ZUGFeRD and Factur-X, but in Germany it does not constitute a full invoice within the meaning of the German UStG (VAT Act).
MINIMUM	The profile is also included in ZUGFeRD 2.0/2.1 for reasons of consistency between the two standards ZUGFeRD and Factur-X, but in Germany it does not constitute a full invoice within the meaning of the German UStG (VAT Act).
Default	BASIC

PDF.ZUGFeRDVersion: Set the ZUGFeRD version.

Value	Meaning
1.0	ZUGFeRD version 1.0 is forced.
2.0	ZUGFeRD version 2.0 is forced.
2.1	Factur-x is forced which corresponds to ZUGFeRD version 2.1.
Default	2.0

PDF.ZUGFeRDXmlPath: Defines the path to a ZUGFeRD compliant XML file, which should be embedded in the final PDF. The XML file must be created before by the application itself.

Picture.JPEGQuality: Specifies the quality and the corresponding compression factor of the generated JPEG graphic. The value lies between 0 and 100, with 100 representing the highest quality (and therefore the least compression). Takes effect

only when the source graphic is not the JPEG format, as encoding of JPEG to JPEG would result in a quality loss. Default: 75

Export.Path: Path where the exported files should be saved. If this option is empty, a file selection dialog will always be displayed.

Export.File: File name of the document.

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	Export with user interaction (dialogs)
1	No dialogs or message boxes – even overwrite warnings - will be displayed (only if *Export.Path* was specified).
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the registered file extension.

Value	Meaning
0	Result will not be displayed automatically
1	Calls *ShellExecute()* with *Export.File.*
Default	0

Export.ShowResultAvailable: Enables you to hide the respective checkbox in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default	1

7.3.2 Excel Export

Overview

The Excel export module creates Microsoft Excel® documents. The creation is independent of any installed version of this product, i.e. the export is native. Depending on your needs, the full layout can be conserved during export, or only the data from table objects is exported unformatted.

Limitations

The following limitations should be considered:

- Excel renders texts with an increased height compared to the standard output. Thus all fonts are scaled down by an optional factor. You may set this factor using the XLS.FontScalingPercentage option.
- Excel does not accept any objects on the non-printable area of your printer. This results in a wider printout compared to List & Label. This effect can be minimized by setting a zoom for the printout (see XLS.PrintingZoom option).
- RTF texts are embedded as pictures. As this increases the file size remarkably, we recommend that you use normal text wherever possible.
- Tabs will be replaced by blanks.
- The option 'Separators fixed' in the table object is not supported.
- The option 'Fixed size' in the table object is not supported.
- Fill patterns available in List & Label cannot be transformed to XLS. All fillings are solid.
- Chart and HTML objects are exported as pictures and thus cannot appear transparently.
- The print order of lines and rectangles is disregarded, lines and rectangle frames always appear in the foreground.
- The print order of texts and rectangles is disregarded; text always appears in the foreground.
- Texts partially overlapping filled rectangles are filled completely.
- Overlapping text and picture objects are ignored.
- Lines that are neither horizontal nor vertical are ignored.
- Picture objects are rendered with a white frame.
- Large filled areas in projects with many different object coordinate values can decrease the speed remarkably.
- Line widths are ignored.
- Rotated RTF objects and pictures are not supported.
- Objects exported as pictures should fit into their rectangle, important for e.g. barcodes.
- If the coordinates of two objects are only slightly different (i.e. a fraction of a millimeter), frame lines might become invisible as Excel is not capable of displaying them correctly.
- Gradient fills are not supported.
- Rotated text (180°) is not supported.
- Custom drawings in callbacks are not supported.
- Paragraph spacing is not supported.
- Negative values for spacing are not supported.
- The maximum number of Excel columns is limited to 256.

- Issue print is not supported.
- Shadow Pages are not supported.
- The wrapping option 'Minimum Size' in the crosstab object is not supported.
- Expandable Regions for crosstabs are not supported.
- Frame inner offsets are not supported.
- Very large amounts of data with various cell formats, RTF texts, graphics, coordinate ranges, etc. can lead to the memory consumption of a 32-bit application reaching its limits. If the pure data can also be output without formatting etc., the less memory intensive simple text-based CSV export can be an alternative. If, however, more memory must be made available, then switching the application to 64-bit can be a solution.

Programming Interface

You can find a description of all options used in the XLS export module in this chapter. The options can be modified using the methods *LIXSetParameter(..."XLS"...)* and read by calling *LIXGetParameter(..."XLS"...).*

Resolution: Defines the resolution in dpi for the generation of pictures. Default: *300 dpi*.

Picture.BitsPerPixel: Defines the color depth of the generated picture. Please note that the picture files will quickly get very large with higher color depths.

Value	Meaning
1	Black & White
24	24bit True Color
Default	24

Picture.JPEGQuality: Specifies the quality and the corresponding compression factor of the generated JPEG graphic. The value lies between 0 and 100, with 100 representing the highest quality (and therefore the least compression). Takes effect only when the source graphic is not the JPEG format, as encoding of JPEG to JPEG would result in a quality loss. Default: 75

Verbosity.Rectangle: Configures how rectangle objects should be exported.

Value	Meaning
0	Ignore object
1	Object as rectangle
2	Object as picture
Default	1

Verbosity.Barcode: Configures how barcode objects should be exported.

Value	Meaning

0	Ignore object
1	Object as picture
Default	1

Verbosity.Drawing: Configures how picture objects should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
Default	1

Verbosity.Ellipse: Configures how ellipse objects should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
Default	1

Verbosity.Line: Configures how line objects should be exported.

Value	Meaning
0	Ignore object
1	Object as line
2	Object as picture
Default	1

Verbosity.Text: Configures how text objects should be exported.

Value	Meaning
0	Ignore object
1	Object as text object
2	Object as picture
Default	1

Verbosity.RTF: Configures how RTF objects should be exported.

Value	Meaning
0	Ignore object
1	As unformatted text
2	Object as picture
Default	1

Verbosity.Table: Configures how table objects should be exported.

Value	Meaning
0	Ignore object
1	As a complete table object
Default	1

Verbosity.LLXObject: Configures how LLX objects (e.g. chart object) should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG
Default	1

XLS.FontScalingPercentage: Scaling factor for the font sizes. Necessary in order to compensate for the increased text height in Excel. Maximum value: 100, Default: 89

XLS.PrintingZoom: Scaling factor for the printout of the project. Necessary in order to compensate for the inability to place any objects in the non-printable area. Default: 88(=88% zoom)

XLS.IgnoreGroupLines: Allows group header and footer lines to be ignored in the resulting Excel file. Only effective if *Export.OnlyTabledata* has been set (see below).

Value	Meaning
0	Group lines are exported
1	Group lines are ignored
Default	1

XLS.IgnoreHeaderFooterLines: Allows header and footer lines to be ignored in the resulting Excel file. Only effective if *Export.OnlyTabledata* has been set (see below).

Value	Meaning
0	Header and footer lines are exported
1	Header and footer lines are ignored
2	Header and footer lines are exported once on the first page. To export the footer lines only on the last page, set the appearance condition to *LastPage()*.
Default	1

XLS.IgnoreLinewrapForDataOnlyExport: Allows line wraps to be ignored. Only effective if *Export.OnlyTabledata* has been set (see below).

Value	Meaning
0	Line wraps are exported to Excel
1	Line wraps are ignored
Default	1

XLS.ConvertNumeric: Allows switching of the automatic conversion of numeric values in the created Excel sheet.

Value	Meaning
0	No automatic conversion
1	Numeric values are formatted according to the setting in the Designer under 'File > Options > Project'.
2	Only columns which actually contain a numeric value (e.g. a price) will be converted. If a numeric column is explicitly formatted in List & Label (e.g. Str$(price,0,0)), then it will not be converted.
3	List & Label tries to transform the output formatting configured in the Designer to Excel as exact as possible. If the "Format" property in the designer is not used, the content will be passed as Text to Excel.
Default	3

XLS.AllPagesOneSheet: Enables the creation of a separate XLS worksheet for each page.

Value	Meaning
0	Create separate worksheet for each page
1	All pages are added to the same worksheet
Default	1

XLS.WorksheetName: Configures the name of the worksheet(s). You can use the format identifier "%d" in the name. It will be replaced by the page number at runtime (ex. "Report page %d").

XLS.FileFormat: Configures the file format.

Value	Meaning
0	Format is recognized automatically by the file extension
1	Office XML (XLSX) format will be used

Value	Meaning
2	Excel (XLS) format will be used
Default	0

Export.Path: Path where the exported files should be saved. If this option is empty, a file selection dialog will always be displayed.

Export.File: File name of the document.

Export.InfinitePage: This "endlessly" increases the size of the page output, you get an export that is not divided by breaks (unless you work with "Pagebreak Before", then the page will still be wrapped there).

Value	Meaning
0	Single pages
1	Endless page
Default	0

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	Export with user interaction (dialogs)
1	No dialogs or message boxes will be displayed (only if *Export.Path* was specified).
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the registered file extension.

Value	Meaning
0	Result will not be displayed automatically
1	Calls *ShellExecute()* with *Export.File.*
Default	0

Export.ShowResultAvailable: Enables to hide the respective checkbox in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default	1

Export.OnlyTableData: Only data from table lines will be exported.

Value	Meaning
0	All objects are exported.
1	Only table cells and their data are exported. The font properties "Bold", "Italic" and the horizontal alignment of the text is used in the result file. Other format options are ignored to ensure best reusability of the result in Excel.
Default	0

7.3.3 Word Export

Overview

The Word export module creates documents in Microsoft Word® format. The creation is executed independently from the installation of the product, it is therefore natively supported. A full layout-preserving export is executed. Tables are created on continuous pages to support editing later.

Limitations

Please note the following limitations and hints for the Word export module:

- Requires .NET Framework 4.7.
- Compatible with Microsoft Word® 2010 and higher.
- It is recommended that the width of all columns of a line matches the total width of the report container. During the design, try always to justify the borders of different cells that occur in multiple table sections (header line, data line etc.) or multiple line definitions. Otherwise, the result can be falsified in Microsoft Word.
- Table lines that contain a picture will be exported with a fixed height.
- A mix of different page formats is not supported. To achieve an export of e.g. portrait and landscape format, all pages of the same format can be each exported to a separate document.
- Due to format restrictions, it might be necessary to adapt the report's layout before exporting to DOCX. We suggest to thoroughly testing the output before redistribution. Also, note the options DOCX.CellScalingPercentageHeight and DOCX.CellScalingPercentageWidth.
- Tabulators are not supported.
- Issue print is not supported.
- The fit option "compress" in the properties of a column is not supported.
- Shadow Pages are not supported.
- The option 'Separators fixed' in the table object is not supported.
- The option 'Fixed size' in the table object is not supported.
- The wrapping option 'Minimum Size' in the crosstab object is not supported.

- When using the option DOCX.FloatingTableMode it is not possible to use a report container that contains multiple objects with different structures.
- Frame inner offsets are not supported.
- Columns of type 'Table' will be exported as picture.
- Anchoring lines is not supported.
- Certain control characters cannot be displayed in Microsoft Word and are therefore filtered out of texts using the. NET Framework method Char.IsControl().
- Table of contents and index are only exported as simple tables without links.
- Line spacing is not directly supported. However, these can be simulated with the help of blank lines and the properties "Inerasable" (Yes) and "Blank Optimization" (No). Alternatively, it is also possible to use "Chr$(13)" without setting the above properties.

Programming Interface

You can find a description of all options used in the DOCX export module in this chapter. These options can be modified/read by the application using the methods *LIXSetParameter(..."DOCX"...)* and *LIXGetParameter(..."DOCX"...)*.

Resolution: Defines the resolution in dpi for the generation of pictures. Default: *96dpi*, screen resolution.

Picture.BitsPerPixel: Defines the color depth of the generated pictures.

Value	Meaning
1	Black & White
24	24bit True Color
Default	24

Picture.JPEGQuality: Specifies the quality and the corresponding compression factor of the generated JPEG graphic. The value lies between 0 and 100, with 100 representing the highest quality (and therefore the least compression). Takes effect only when the source graphic is not the JPEG format, as encoding of JPEG to JPEG would result in a quality loss. Default: 75

Verbosity.Rectangle: Configures how rectangle objects should be exported.

Value	Meaning
0	Ignore object
1	Object as rectangle
2	Object as picture
Default	1

Verbosity.Barcode: Configures how barcode objects should be exported.

Value	Meaning

0	Ignore object
1	Object as picture
Default	1

Verbosity.Drawing: Configures how picture objects should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
Default	1

Verbosity.Ellipse: Configures how ellipse objects should be exported.

Value	Meaning
0	Ignore object
1	Object as ellipse
2	Object as picture
Default:	1

Verbosity.Line: Configures how line objects should be exported.

Value	Meaning
0	Ignore object
1	Object as line
2	Object as picture
Default	1

Verbosity.Text: Configures how text objects should be exported.

Value	Meaning
0	Ignore object
1	Object as text object
2	Object as picture
Default	1

Verbosity.RTF: Configures how RTF objects should be exported.

Value	Meaning
0	Ignore object
1	As formatted text
2	As unformatted text
3	Object as picture
Default	1

Verbosity.Table: Configures how table objects should be exported.

Value	Meaning
0	Ignore object
1	As a complete table object
Default	1

Verbosity.NestedTable: Configures how nested table objects should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
Default	1

Verbosity.LLXObject: Configures how LLX objects (OLE, HTML, chart) should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
Default:	1

DOCX.FontScalingPercentage: Scaling factor to correct font sizes. Default: 100 (=100% font size)

DOCX.CellScalingPercentageHeight: Scaling factor (with decimal places) to correct the cell heights. Default: 100 (=100% cell height)

DOCX.CellScalingPercentageWidth: Scaling factor (with decimal places) to correct the cell widths. Default: 100 (=100% cell width)

DOCX.AllPagesOneFile: Enables creation of a separate Word document for each page.

Value	Meaning
0	A separate Word document is create per page
1	All pages are created in the same Word document
Default	1

DOCX.FloatingTableMode: Enables if tables will be linked. For a larger amount of pages with tables this option should be set to '0', because Microsoft Office Word can only link up to 86 (depending on the Word version) tables.

Value	Meaning
0	Table won't be linked
1	Tables will be linked
Default	1

DOCX.IgnoreCellPadding: Defines wether the Border Spacing will be ignored.

Value	Meaning
0	Border Spacing will not be ignored
1	Border Spacing will be ignored
Default	0

Export.File: Defines the file name of the generated Word document. If empty, the file selection dialog will be displayed.

Export.Path: Defines the path of the generated Word document.

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	Export with user interaction (dialogs)
1	No dialogs or message boxes will be displayed (only if *Export.File* was specified).
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the registered file extension.

Value	Meaning
0	Result will not be displayed automatically
1	Calls *ShellExecute()* with *Export.File* so that usually Microsoft Word® should be executed.
Default	0

7.3.4 PowerPoint Export

Overview

The PowerPoint export module creates documents in Microsoft PowerPoint® format. The creation is executed independently from the installation of the product; it is therefore natively supported. A full layout-preserving export is executed.

Limitations

Please note the following limitations and hints for the PowerPoint export module:

- Requires .NET Framework 4.7.
- Compatible with Microsoft PowerPoint® 2010 and higher.
- It is recommended that the width of all columns of a line matches the total width of the report container. During the design, try to always justify the borders of different cells that occur in multiple table sections (header line, data line etc.) or multiple line definitions. Otherwise the result can be falsified in Microsoft Word.
- Columns cannot be smaller than 0,54 cm (5,4mm). All columns will be automatically resized to this Size by Microsoft PowerPoint.
- Fonts will be reduced by 1%, otherwise the result can be wrong in Microsoft PowerPoint.
- Table lines that contain a picture will be exported with a fixed height.
- A mix of different page formats is not supported. To achieve an export of e.g. portrait and landscape format, all pages of the same format can be each exported to a separate document.
- Tabulators are not supported.
- Issue print is not supported.
- Shadow Pages are not supported.
- Nested tables will be exported as image by default.
- The wrapping option 'Minimum Size' in the crosstab object is not supported.
- PowerPoint adjusts pictures to the height of the line.
- Frame inner offsets are not supported.

Programming Interface

You can find a description of all options used in the PowerPoint export module in this chapter. These options can be modified/read by the application using the methods *LIXSetParameter(..."PPTX"...)* and *LIXGetParameter(..."PPTX"...).*

Resolution: Defines the resolution in dpi for the generation of pictures. Default: *96dpi*, screen resolution.

Picture.BitsPerPixel: Defines the color depth of the generated pictures.

Value	Meaning

1	Black & White
24	24bit True Color
Default	24

Picture.JPEGQuality: Specifies the quality and the corresponding compression factor of the generated JPEG graphic. The value lies between 0 and 100, with 100 representing the highest quality (and therefore the least compression). Takes effect only when the source graphic is not the JPEG format, as encoding of JPEG to JPEG would result in a quality loss. Default: 75

Verbosity.Rectangle: Configures how rectangle objects should be exported.

Value	Meaning
0	Ignore object
1	Object as rectangle
2	Object as picture
Default	1

Verbosity.Barcode: Configures how barcode objects should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
Default	1

Verbosity.Drawing: Configures how picture objects should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
Default	1

Verbosity.Ellipse: Configures how ellipse objects should be exported.

Value	Meaning
0	Ignore object
1	Object as ellipse
2	Object as picture
Default:	1

Verbosity.Line: Configures how line objects should be exported.

Value	Meaning
0	Ignore object
1	Object as line

2	Object as picture
Default	1

Verbosity.Text: Configures how text objects should be exported.

Value	Meaning
0	Ignore object
1	Object as text object
2	Object as picture
Default	1

Verbosity.RTF: Configures how RTF objects should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
Default	1

Verbosity.Table: Configures how table objects should be exported.

Value	Meaning
0	Ignore object
1	As a complete table object
Default	1

Verbosity.NestedTable: Configures how nested table objects should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
Default	1

Verbosity.LLXObject: Configures how LLX objects (OLE, HTML, chart) should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
Default:	1

PPTX.FontScalingPercentage: Scaling factor to correct font sizes. Default: 100 (=100% font size)

PPTX.Animation: Defines the used Transition for a slide change

Value	Meaning
0	No Animation
1	Cut-Animation
2	Fade-Animation
3	Push-Animation
4	Cover-Animation
5	Wipe-Animation
Default	0

Export.File: Defines the file name of the generated PowerPoint document. If empty, the file selection dialog will be displayed.

Export.Path: Defines the path of the generated PowerPoint document.

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	Export with user interaction (dialogs)
1	No dialogs or message boxes will be displayed (only if *Export.File* was specified).
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the registered file extension.

Value	Meaning
0	Result will not be displayed automatically
1	Calls *ShellExecute()* with *Export.File* so that usually Microsoft PowerPoint® should be executed.
Default	0

7.3.5 RTF Export

Overview

The RTF export module creates documents in Rich Text Format based on Microsoft specification 1.5/1.7. The exported files have mainly been optimized for Microsoft Word. Please note that the rendering of the exported files can differ with different word processors.

Limitations

Besides others, the following hints and limitations should be considered:

- Rows that are anchored to each other are not correctly exported.
- The max. color depth is 24 bit (PNG: 32bit).
- Shadows of rectangle objects are not supported.
- Tabs will be replaced by blanks.
- Objects should not be placed too close to the page borders. Some word processors create page breaks in such cases. This means that all following objects will then automatically be placed on the next page.
- The option 'Separators fixed' in the table object is not supported.
- Not all background patterns available in List & Label can be transformed to RTF. The number of patterns available in RTF is much smaller than that of the patterns available in List & Label.
- The chart and HTML object are exported as pictures and thus cannot appear transparently.
- Rotated RTF texts and pictures are not supported.
- Distances within table cells are not supported.
- Paragraph spacing is not supported.
- Frames narrower than ½ pt (about 0.4 mm in List & Label) will not be displayed correctly.
- Frames of the same size in X and Y direction will not be displayed as squares.
- Object frames are not supported.
- Gradient fills are not supported.
- Rotated text is not supported.
- Custom drawings in callbacks are not supported.
- TotalPages$() may not be used in rotated text objects.
- Paragraph distances are not supported.
- Issue print is not supported.
- Shadow Pages are not supported.
- Nested tables are only supported one level (i.e. no subtable support) if they are not exported as picture (see Verbosity.NestedTable below).
- The wrapping option 'Minimum Size' in the crosstab object is not supported.
- Table cells in the crosstab object, that horizontally and vertically overlap multiple other cells cannot be exported exactly.

Known specialities in general:

- Frames smaller than ½ pt will not be displayed correctly.
- Positon frames in Word are handled unusual, because length properties are interpreted incorrect by Word.
- Small line objects may be not displayed because the corresponding bitmap gets an offset causing the line object to be out of the frame.
- Table frames might not always be displayed correctly.
- Distances between cells are not supported.
- Not all colors that can be used in List & Label are interpreted correctly by Word.
- When exporting large images at large resolutions, these images are sometimes not displayed by Word although they are referenced correctly in the RTF.
- We recommend to make any objects and table cells more generous in height and width, because RTF uses additional inner spacing in some areas, which are of course not visible in the Designer.

Programming Interface

You can find a description of all options used in the RTF export module in this chapter. The options can be modified/read using the methods *LIXSetParameter(..."RTF"...)* and *LIXGetParameter(..."RTF"...).*

Resolution: Defines the resolution in dpi for the transformation of the coordinates and the generation of pictures. Default: *96 dpi*, screen resolution.

Picture.BitsPerPixel: Defines the color depth of the generated picture. Please note that the picture files will quickly get very large with higher color depths.

Value	Meaning
1	Black & White
4	16 Colors
8	256 Colors
24	24bit True Color
Default	24

UsePosFrame: Switches the text positioning method.

Value	Meaning
0	Text boxes used for positioning
1	Position frames used for positioning
Default	0

Verbosity.Rectangle: Configures how rectangle objects should be exported.

Value	Meaning
0	Ignore object

1	Object as frame
Default	1

Verbosity.Barcode: Configures how barcode objects should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
Default	1

Verbosity.Drawing: Configures how picture objects should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
Default	1

Verbosity.Ellipse: Configures how ellipse objects should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
2	Object as shape object
Default	2

Verbosity.Line: Configures how line objects should be exported.

Value	Meaning
0	Ignore object
1	Object as picture
2	Object as shape object
Default	2

Verbosity.Text: Configures how text objects should be exported.

Value	Meaning
0	Ignore object
1	Object as text object
2	Object as picture
Default	1

Verbosity.RTF: Configures how RTF objects should be exported.

Value	Meaning
0	Ignore object
1	As formatted RTF text
2	Object as picture
Default	1

Verbosity.Table: Configures how table objects should be exported.

Value	Meaning
0	Ignore object
1	As a complete table object
Default	1

Verbosity.NestedTable: Configures how nested table objects should be exported.

Value	Meaning
0	Ignore object
1	As a complete table object
2	Object as picture
Default	1

Verbosity.LLXObject: Configures how LLX objects (e.g. chart object) should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG
Default	1

Export.Path: Path where the exported files should be saved.

Export.File: File name of the RTF document. If this option is set to an empty string, a file selection dialog will always be displayed.

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	With user interaction (dialogs)
1	No dialogs or message boxes – even overwrite warnings - will be displayed (only if *Export.File* was specified).
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the registered file extension.

Value	Meaning
0	Result will not be displayed automatically
1	Calls *ShellExecute()* with *Export.File*
Default	0

Export.ShowResultAvailable: Enables you to hide the respective checkbox in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default	1

7.3.6 XPS Export

Overview

The XPS export format is available as soon as .NET Framework 3.5 is installed on the computer. The export module uses the installed Microsoft XPS printer driver for the output.

Some limitations must be taken into account here too, for example the driver does not currently support all clipping options of the Windows GDI. This can result in display errors in the XPS file when exporting charts and generally truncated/clipped objects. As a precaution, we recommend that you carefully check the XPS output before delivery to your customers.

Programming Interface

You can find a description of all options used in the XPS export module in this chapter. The options can be modified using the methods *LIXSetParameter(..."XPS"...)* and read by calling *LIXGetParameter(..."XPS"...).*

Export.Path: Path where the exported files should be saved.

Export.File: File name of the document. If this option is empty, a file selection dialog will always be displayed.

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	Export with user interaction (dialogs)
1	No dialogs or message boxes – even a overwrite warning - will be displayed (only if *Export.Path* was specified).

Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the associated file type.

Value	Meaning
0	Result will not be displayed automatically
1	Calls *ShellExecute()* with *Export.File.*
Default	0

Export.ShowResultAvailable: Enables you to hide the respective checkbox in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default	1

7.3.7 XHTML/CSS Export

The XHTML export module creates XHTML code according to the XHTML 1.0 specification and CSS code according to the CSS 2.1 specification.

Overview

The export module collects all List & Label objects for the page currently being printed and orders them according to their height, width and position. The position of an object results from two values: left and top. They specify the distance from the upper left page border. All objects are positioned absolutely on a page, which leads to a more accurate export result.

Limitations

There are limitations set by the target format. The most important are listed now.

- Rows that are anchored to each other are not correctly exported.
- Decimal tabs will be transformed to right-aligned tabs.
- Any other tabs will be transformed to a blank.
- The option 'Line break' in text objects and table columns is always active in the export result.
- The option 'Separators fixed' in table objects is not supported.
- The chart object is exported as a picture and thus cannot appear transparently.
- The transformation of RTF text to HTML code is carried out by an RTF parser. This parser interprets the basic RTF formatting. Extended RTF functionality such as embedded objects will be ignored.

- Diagonal lines are exported as images.
- Gradient fills with more than 3 colors are not supported.
- Objects to be exported as picture should not overlap the object frame. Therefore e.g. barcode objects with fixed bar width must fit in the object rectangle.
- Custom drawings in callbacks must be exported as picture.
- Table frames of neighboring cells are not drawn so that they overlap each other, but discretely. This can double the frame width and needs to be considered during the design.
- Even if the HTML object wraps over several pages, it will be exported in one stream, i.e. no page wrap will occur.
- Embedded scripting functionalities may be lost.
- Issue print is not supported.
- Rotated descriptions in the crosstab object are not supported.
- Shadow Pages are not supported.
- The wrapping option 'Minimum Size' in the crosstab object is not supported.
- The property "Link" is not supported.

Programming Interface

You can find a description of all options used in the XHTML export module in this chapter. These options can be modified/read using the methods *LIXSetParameter(..."XHTML"...)* and *LIXGetParameter(..."XHTML"...).*

Resolution: Defines the resolution in dpi for the transformation of the coordinates and the generation of pictures. Default: 96

Picture.JPEGQuality: Specifies the quality and the corresponding compression factor of the generated JPEG graphic. The value lies between 0 and 100, with 100 representing the highest quality (and therefore the least compression). Takes effect only when the source graphic is not the JPEG format, as encoding of JPEG to JPEG would result in a quality loss. Default: 75

Picture.BitsPerPixel: Defines the color depth of the generated picture. Please note that the picture files will quickly get very large with higher color depths.

Value	Meaning
1	Black & White
24	24bit True Color
Default	24

Verbosity.Rectangle: Configures how rectangle objects should be exported.

Value	Meaning
0	Ignore object

1	Object as JPEG (and also as a complete rectangle for objects with colored background).
Default	1

Verbosity.Barcode: Configures how barcode objects should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG
Default	1

Verbosity.Drawing: Configures how picture objects should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG
Default	1

Verbosity.Ellipse: Configures how ellipse objects should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG
Default	1

Verbosity.Line: Configures how line objects should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG
Default	1

Verbosity.Text: Configures how text objects should be exported.

Value	Meaning
0	Ignore object
1	Object as text object
2	Object as JPEG
Default	1

Verbosity.Text.Frames: Configures how text object frames should be exported.

Value	Meaning
0	Single frames for top, bottom, left, right (uses CSS)
1	Complete frame as box
Default	0

Verbosity.RTF: Configures how RTF objects should be exported.

Value	Meaning
0	Ignore object
1	As formatted RTF text (parsed and converted to HTML)
2	As unformatted text (uses the default font specified in the project file)
3	Object as JPEG
Default	1

Verbosity.RTF.Frames: Configures how RTF object frames should be exported.

Value	Meaning
0	Single frames for top, bottom, left, right (uses CSS)
1	Complete frame as box
Default	0

Verbosity.Table: Configures how table objects should be exported.

Value	Meaning
0	Ignore object
1	As a table object
Default	1

Verbosity.Table.Cell: Configures how table cells should be exported.

Value	Meaning
0	Ignore cell
1	As a cell object using the verbosity settings of the object types specified in the cell.
2	Cells as JPEG
Default	1

Verbosity.Table.Frames: Configures how table frames should be exported.

Value	Meaning
0	Ignore table frame
1	Only horizontal lines of table frames
2	The whole table line including all frames
3	Cell-specific frames (uses CSS)
Default	3

Verbosity.LLXObject: Configures how LLX objects (e.g. chart object) should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG
Default	1

Verbosity.LLXObject.HTMLObj: Configures how the HTML object should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG
2	Object as embedded HTML. Only the HTML text between the <BODY> and </BODY> tags will be exported. Please note the hint on exporting limitations.
Default	2

XHTML.DrawingAsSVG: Specifies the format in which graphics are exported.

Value	Meaning
0	Old mode, graphics are exported as PNG.
1	New mode, graphics are exported as SVG.
Default	1

XHTML.Title: Specifies the title of the generated XHTML document.

XHTML.UseAdvancedCSS: Allows the usage of non-standard CSS formatting styles.

Value	Meaning
0	No non-standard CSS formatting styles are used
1	Non-standard CSS formatting styles may be used, e.g. to create a gradient fill.
Default	1

XHTML.UseOriginalURLsForImages: Specifies from where images will be loaded from.

Value	Meaning
0	Images will be temporarily stored on the local machine.
1	Images will be loaded from their original path.
Default	0

XHTML.ToolbarType: Specifies if an additional toolbar will be created.

Value	Meaning
0	No toolbar will be created.
1	A toolbar with color scheme *Skyblue* will be created.
2	A toolbar with color scheme *Blue* will be created.
3	A toolbar with color scheme *Black* will be created.
4	A toolbar with color scheme *Web* will be created.
Default	4

XHTML.UseSeparateCSS: Specifies if a separate CSS file will be created.

Value	Meaning
0	CSS will be added to the HEAD area of the XHTML file.
1	CSS will be created in a separate file.
Default	0

Layouter.Percentaged: This option configures whether the layout should be defined in absolute values or with values expressed as percentage.

Value	Meaning
0	Layout of the X coordinates in absolute values (pixel)
1	Layout of the X coordinates with values expressed as percentage
Default	0

Layouter.FixedPageHeight: Configures whether all pages should be forced to have the same page height.

Value	Meaning
0	Layout can shrink on the last page (e.g. if no objects have been placed in the page footer)
1	The page height is set as configured
Default	1

Export.Path: Path where the exported files should be saved. If this option is empty, a file selection dialog will always be displayed.

Export.File: File name of the first HTML page. Default: "index.htm". You may also use printf format strings like "%d" in the file name (ex. "Export Page %d.htm"). In this case, the files for the pages will be named by replacing the placeholder with the correctly formatted page number. Otherwise, you will get a simple page numbering for the result files.

Export.InfinitePage: This "endlessly" increases the size of the page output, you get an export that is not divided by breaks (unless you work with "Pagebreak Before", then the page will still be wrapped there).

Export.AllInOneFile: Configures the export result format.

Value	Meaning
0	Every printed page will be exported in a single HTML file.
1	The result is a single HTML file (*Export.File)*, containing all printed pages.
Default	1

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	Export with user interaction (dialogs)
1	No dialogs or message boxes will be displayed (only if *Export.Path* was specified).
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the registered file extension.

Value	Meaning
0	Result will not be displayed automatically
1	Calls *ShellExecute()* with *Export.File.*
Default	0

Export.ShowResultAvailable: Enables you to hide the respective checkbox in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default	1

Hyperlinks

Hyperlinks can be embedded in text, table and RTF objects directly in the Designer using the *Hyperlink$()* function. Dynamic hyperlinks and formulas can be realized in this way.

Another way of creating hyperlinks is via the object name. The names of text and picture objects are parsed for the string "/LINK:<url>". Whenever this string is found, a hyperlink is generated. Thus, if you name your picture object "combit /LINK:https://www.combit.com", this would create a picture hyperlink during export to XHTML.

7.3.8 MHTML Export

Overview

The MHTML (Multi Mime HTML) export module functions analogously to the XHTML export module. However, pictures are embedded MIME encoded into the export file. Thus the result is only one file (.MHT). This is most useful for sending invoices by mail, as the recipient can open the file directly and does not need any access to further (external) picture files.

Programming Interface

All options of the XHTML exporter are supported; pass "MHTML" as module name. The option Export.AllInOneFile will be ignored, as this format always results in one file only.

7.3.9 JSON Export

Overview

The JSON Export module is internally based on Text (CSV) Export and is therefore subject to similar restrictions. Similar to CSV, data from table objects is returned here and transferred to a JSON structure. The table rows form the record, while the header row is used to determine the identifiers used in JSON. Footer lines, group header lines, group footer lines and all objects outside the table, such as freely placed texts, are ignored. The result is a single file in JSON format that contains the data from all table objects. This can then be used for further processing in other applications. Please note that in this mode only data from tables is exported and no layout information is evaluated. This also means, for example, that layout-related breaks are filtered from the exported text. This mode is only available for table projects.

Limitations

- The JSON Export module has the following limitations:
- Issue print is not supported.
- Nested tables are not supported.

- Free objects or texts within the layout are not supported.
- Headers, footers, group headers and group footers are not supported. However, the header lines determine the identifiers in the data lines.
- Line breaks within the table lines are not supported. To exclude this as a possible source of error, the use of infinite pages is recommended.

Programming Interface

You can find a description of all options used in the JSON export module in this chapter. The options can be modified using the methods *LIXSetParameter(…"JSON"…)* and read by calling *LIXGetParameter(…"JSON"…).*

Export.Path: Path where the exported files should be saved.

Export.File: File name of the document. If this option is empty, a file selection dialog will always be displayed, default "export.json".

Export.InfinitePage: This "endlessly" increases the size of the page output, you get an export that is not divided by breaks (unless you work with "Pagebreak Before", then the page will still be wrapped there).

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	Export with user interaction (dialogs)
1	No dialogs or message boxes will be displayed (only if *Export.Path* was specified).
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the associated file type.

Value	Meaning
0	Result will not be displayed automatically
1	Calls *ShellExecute()* with *Export.File.*
Default	0

Export.ShowResultAvailable: Enables you to hide the respective checkbox in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default	1

TXT.Charset: Specifies the character set of the result file. The target code page needs to be passed in addition (e.g. 932 for Japanese) using *LL_OPTION_CODEPAGE*.

Value	Meaning
ANSI	Ansi character set
ASCII	Ascii character set
UNICODE	Unicode character set
UTF8	UTF8 character set
Default	UNICODE

JSON.AutodetectDatatype: This can be used to define whether all table columns should be output as text or with an automatically assigned JSON type.

Value	Meaning
0	Output as text
1	Output with data type (Null, numeric, date, text)
Default	1

7.3.10 Text (CSV) Export

Overview

The CSV-Export exports data from table objects to a text format. The separator and framing character can be optionally set. The result is one single text file containing the data from all table objects. Please note that the layout is not preserved in any way, this is purely a data conversion export.

Limitations

- Issue print is not supported.
- Nested tables are not supported.

Programming Interface

You can find a description of all options used in the TXT export module in this chapter. The options can be modified using the methods *LlXSetParameter(..."TXT"...)* and read by calling *LlXGetParameter(..."TXT"...).*

Export.Path: Path where the exported files should be saved.

Export.File: File name of the document. If this option is empty, a file selection dialog will always be displayed, default "export.txt".

Export.InfinitePage: This "endlessly" increases the size of the page output, you get an export that is not divided by breaks (unless you work with "Pagebreak Before", then the page will still be wrapped there).

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	Export with user interaction (dialogs)
1	No dialogs or message boxes will be displayed (only if *Export.Path* was specified).
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the associated file type.

Value	Meaning
0	Result will not be displayed automatically
1	Calls *ShellExecute()* with *Export.File.*
Default	0

Export.ShowResultAvailable: Enables you to hide the respective checkbox in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default	1

TXT.FrameChar: Specifies the framing character for the columns.

Value	Meaning
NONE	No framing
"	" as framing character
'	' as framing character

TXT.SeparatorChar: Specifies the separator character.

Value	Meaning
NONE	No separator
TAB	Tab as separator
BLANK	Blank as separator
,	, as separator
;	; as separator

TXT.IgnoreGroupLines: Allows group header and footer lines to be ignored in the resulting text file.

Value	Meaning
0	Group lines are exported

1	Group lines are ignored
Default	1

TXT.IgnoreHeaderFooterLines: Allows header and footer lines to be ignored in the resulting text file.

Value	Meaning
0	Header and footer lines are exported
1	Header and footer lines are ignored
2	Header and footer lines are exported once on the first page. To export the footer lines only on the last page, set the appearance condition to *LastPage()*.
Default	1

TXT.Charset: Specifies the character set of the result file. The target code page needs to be passed in addition (e.g. 932 for Japanese) using *LL_OPTION_CODEPAGE*.

Value	Meaning
ANSI	Ansi character set
ASCII	Ascii character set
UNICODE	Unicode character set
UTF8	UTF8 character set
Default	UNICODE

7.3.11 Text (Layout) Export

Overview

The Layout-Export can alternatively create a text file that resembles – as far as the format allows – the layout of the project. Please make sure that you choose a font size that is large enough in the Designer. If the lines of text in your project cannot be assigned to different lines in the text file, lines may be overwritten, resulting in a loss of data in the result file. A font size with a minimum of 12 pt is suggested.

Limitations

- Issue print is not supported.
- Shadow Pages are not supported.
- Nested tables are not supported.

Programming Interface

You can find a description of all options used in the Text export module in this chapter. The options can be modified using the methods *LlXSetParameter(…"TXT_LAYOUT"…)* and read by calling *LlXGetParameter(…"TXT_LAYOUT"…).*

Verbosity.Text: Configures how text typed columns should be exported.

Value	Meaning
0	Ignore cell
1	Cell as text
Default	1

Verbosity.RTF: Configures how RTF typed columns should be exported.

Value	Meaning
0	Ignore cell
1	As RTF stream
2	As unformatted text
Default	2

Verbosity.Table: Configures how table objects should be exported.

Value	Meaning
0	Ignore object
1	As a table object
Default	1

Verbosity.Table.Cell: Configures how table cells should be exported.

Value	Meaning
0	Ignore cell
1	As a cell object using the verbosity settings of the object types specified in the cell.
Default	1

Export.Path: Path where the exported files should be saved.

Export.File: File name of the document. If this option is empty, a file selection dialog will always be displayed, default "export.txt".

Export.InfinitePage: This "endlessly" increases the size of the page output, you get an export that is not divided by breaks (unless you work with "Pagebreak Before", then the page will still be wrapped there).

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	Export with user interaction (dialogs)
1	No dialogs or message boxes will be displayed (only if *Export.Path* was specified).
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the associated file type.

Value	Meaning
0	Result will not be displayed automatically
1	Calls *ShellExecute()* with *Export.File*.
Default	0

Export.ShowResultAvailable: Enables you to hide the respective checkbox in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default	1

Export.AllInOneFile: Configures the export result format.

Value	Meaning
0	Every printed page will be exported in a single TXT file. If the filename contains the format identifier "%d", this identifier will be repaced by the corresponding page number.
1	The result is a single TXT file (*Export.File)*, containing all printed pages.
Default	1

TXT.Charset: Specifies the character set of the result file.

Value	Meaning
ANSI	Ansi character set
ASCII	Ascii character set
UNICODE	Unicode character set
UTF8	UTF8 character set
Default	UNICODE

7.3.12 XML Export

Overview

The XML export module creates XML files. This allows flexible editing by other applications. All available object properties are exported. If you require, you may export data from tables only and ignore all further object properties.

Limitations

- Issue print is not supported.

Programming Interface

You can find a description of all options used in the XML export module in this chapter. These options can be modified/read using the methods *LIXSetParameter(..."XML"...)* and *LIXGetParameter(..."XML"...)*.

Resolution: Defines the resolution in dpi for the transformation of the coordinates and the generation of pictures. Default: *96 dpi*, screen resolution.

Picture.JPEGQuality: Specifies the quality and the corresponding compression factor of the generated JPEG graphic. The value lies between 0 and 100, with 100 representing the highest quality (and therefore the least compression). Takes effect only when the source graphic is not the JPEG format, as encoding of JPEG to JPEG would result in a quality loss. Default: 75

Picture.JPEGEncoding: Specifies how to encode JPEG images

Value	Meaning
0	Save JPEGS as (external) files
1	Include pictures MIME encoded into the XML file
2	Ignore JPEG images
Default	0

Picture.BitsPerPixel: Defines the color depth of the generated picture. Please note that the picture files will quickly get very large with higher color depths.

Value	Meaning
1	Black & White
24	24bit True Color
Default	24

Verbosity.Rectangle: Configures how rectangle objects should be exported.

Value	Meaning
0	Ignore object
1	Complete object information
2	Object as JPEG
Default	1

Verbosity.Barcode: Configures how barcode objects should be exported.

Value	Meaning
0	Ignore object

1	Complete object information and object as JPEG
Default	1

Verbosity.Drawing: Configures how picture objects should be exported.

Value	Meaning
0	Ignore object
1	Complete object information and object as JPEG
Default	1

Verbosity.Ellipse: Configures how ellipse objects should be exported.

Value	Meaning
0	Ignore object
1	Complete object information
2	Object as JPEG
Default	1

Verbosity.Line: Configures how line objects should be exported.

Value	Meaning
0	Ignore object
1	Complete object information
2	Object as JPEG
Default	1

Verbosity.Text: Configures how text objects should be exported.

Value	Meaning
0	Ignore object
1	Object as text object
2	Object as JPEG
Default	1

Verbosity.RTF: Configures how RTF objects should be exported.

Value	Meaning
0	Ignore object
1	As RTF stream
2	As unformatted text (uses the default font specified in the project file)
3	Object as JPEG
Default	1

Verbosity.Table: Configures how table objects should be exported.

Value	Meaning
0	Ignore object
1	As a table object
Default	1

Verbosity.Table.Cell: Configures how table cells should be exported.

Value	Meaning
0	Ignore cell
1	As a cell object using the verbosity settings of the object types specified in the cell.
2	Cells as JPEG
Default	1

Verbosity.LLXObject: Configures how LLX objects (e.g. chart object) should be exported.

Value	Meaning
0	Ignore object
1	Complete object information and object as JPEG
Default	1

XML.Title: Specifies the title of the generated XML document.

Export.Path: Path where the exported files should be saved. If this option is empty, a file selection dialog will always be displayed.

Export.File: File name of the first XML page. Default: "export.xml". You may also use printf format strings like "%d" in the filename (ex. "Export Page %d.xml"). In this case, the files for the pages will be named by replacing the placeholder with the correctly formatted page number. Otherwise, you will get a simple page numbering for the result files.

Export.AllInOneFile: Configures the export result format.

Value	Meaning
0	Every printed page will be exported in a single XML file.
1	The result is a single XML file (*Export.File)*, containing all printed pages.
Default	1

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	Export with user interaction (dialogs)
1	No dialogs or message boxes – including overwrite warnings - will be displayed (only if *Export.Path* was specified).
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the registered file extension.

Value	Meaning
0	Result will not be displayed automatically.
1	Calls *ShellExecute()* with *Export.File.*
Default	0

Export.ShowResultAvailable: Enables you to hide the respective checkbox in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default	1

Export.OnlyTableData: Only data from table lines will be exported.

Value	Meaning
0	All objects are exported.
1	Only table cells and their data are exported.
Default	0

7.3.13 Picture Export

Overview

This module creates a graphics file (JPEG, BMP, EMF, TIFF, Multi-TIFF, PNG) for every printed page. The file names of the created graphics will be enumerated. If the file name contains the format identifier "%d", this identifier will be replaced by the page number.

Limitations

- Issue print is not supported.

Programming Interface

You can find a description of all options used in the picture export module in this chapter. The options can be modified/read using the methods *LIXSetParameter(..."<Exportername>"...)* and *LIXGetParameter(..."<Exportername>"...). <Exportername>* can be "*PICTURE_JPEG*", "*PICTURE_BMP*", "*PICTURE_EMF*", "PICTURE_TIFF", "PICTURE_MULTITIFF" or "PICTURE_PNG" depending on the graphic format.

Resolution: Defines the resolution in dpi for the transformation of the coordinates and the generation of pictures. Default: *96 dpi*, screen resolution.

Picture.JPEGQuality: Specifies the quality and the corresponding compression factor of the generated JPEG graphic. The value lies between 0 and 100, with 100 representing the highest quality (and therefore the least compression). Takes effect only when the source graphic is not the JPEG format, as encoding of JPEG to JPEG would result in a quality loss. Default: 75

Picture.BitsPerPixel: Defines the color depth of the generated picture. Please note that the picture files will quickly get very large with higher color depths. Not all picture formats can display all color depths.

Value	Meaning
1	Black & White
4	16 Colors
8	256 Colors
24	24bit True Color
Default	JPEG, PNG: 24, Other: 8

Picture.CropFile: Removes dispensable white frame. Supported export formats: PNG, JPEG and TIFF. This option is not supported when used in services (e.g. IIS) as GDI+ is not available there.

Value	Meaning
0	Image will not be cropped
1	Image will be cropped
Default	0

Picture.CropFrameWidth: Defines the border of a cropped file in pixel.

Export.File: File name containing "%d" as format identifier. The files for the pages will be named by replacing the placeholder by the page number. If you do not set this option, you will get a simple page numbering for the result files. If this option is empty, a file selection dialog will always be displayed.

Export.Path: Path where the exported files should be saved.

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	With user interaction (dialogs)
1	No dialogs or message boxes will be displayed (only if *Export.Path* was specified).
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the registered file extension.

Value	Meaning
0	Result will not be displayed automatically
1	Calls *ShellExecute()* with the first generated graphic file.
Default	0

Export.ShowResultAvailable: Enables you to hide the respective checkbox in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default	1

TIFF.CompressionType: Specifies the compression type for the TIFF export. Please note that not all viewers support compression. For CCITTRLE, CCITT3 and CCITT4 Picture.BitsPerPixel must be set to 1 or to 24 for JPEG.

Value	Meaning
None	No compression
CCITTRLE	CCITT Modified Huffmann RLE
CCITT3	CCITT Group 3 Fax encoding
CCITT4	CCITT Group 4 Fax encoding
JPEG	JPEG DCT compression
ZIP	ZIP compression
LZW	LZW compression
Default	None

TIFF.CompressionQuality: Specifies the compression quality for the TIFF export. Default: 75

7.3.14 SVG Export

Overview

The SVG export module creates SVG code according to the Scalable Vector Graphics (SVG) 1.1 (Second Edition) specification.

Limitations

There are limitations set by the target format. The most important are listed now.

- Rows that are anchored to each other are not correctly exported.
- Decimal tabs will be transformed to right-aligned tabs.
- Any other tabs will be transformed to a blank.
- The option 'Line break' in text objects and table columns is always active in the export result.
- The option 'Separators fixed' in table objects is not supported.
- The list object option "fixed size" is not supported.
- The chart object is exported as a picture and thus cannot appear transparently.
- RTF text will be exported as pictures.
- Spacing before table lines is not supported.
- Diagonal lines are exported as images.
- Custom drawings in callbacks are not supported.
- Objects to be exported as picture should not overlap the object frame.
- Table frames of neighboring cells are not drawn so that they overlap each other, but discretely. This can double the frame width and needs to be taken into account during the design.
- TotalPages$() may not be used in rotated text objects.
- Even if the HTML object wraps over several pages, it will be exported in one stream, i.e. no page wrap will occur.
- Embedded scripting functionalities may be lost.
- Shadow Pages are not supported.
- A mix of different page formats is not supported. To achieve an export of e.g. portrait and landscape format, all pages of the same format can be each exported to a separate document.
- Issue print is not supported.

Programming Interface

You can find a description of all options used in the SVG export module in this chapter. These options can be modified/read using the methods *LIXSetParameter(..."SVG"...)* and *LIXGetParameter(..."SVG"...).*

Export.Path: Path where the exported files should be saved. If this option is empty, a file selection dialog will always be displayed.

Export.File: File name of the first SVG page. Default: "index.svg". You may also use printf format strings like "%d" in the file name (ex. "Export Page %d.svg"). In this case, the files for the pages will be named by replacing the placeholder with the correctly formatted page number. Otherwise, you will get a simple page numbering for the result files.

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	Export with user interaction (dialogs)
1	No dialogs or message boxes will be displayed (only if *Export.Path* was specified).
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the registered file extension.

Value	Meaning
0	Result will not be displayed automatically
1	Calls *ShellExecute()* with *Export.File.*
Default	0

Export.ShowResultAvailable: Enables you to hide the respective checkbox in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default	1

Hyperlinks

Hyperlinks can be embedded in text, table and RTF objects directly in the Designer using the *Hyperlink$()* function. Dynamic hyperlinks and formulas can be realized in this way.

Another way of creating hyperlinks is via the object name. The names of text and picture objects are parsed for the string "/LINK:<url>". Whenever this string is found, a hyperlink is generated. Thus, if you name your picture object "combit /LINK:https://www.combit.com", this would create a picture hyperlink during export to SVG.

7.3.15 TTY Export

Overview

The TTY export format can be used to directly communicate with dot matrix printers. This brings a great performance boost compared to the printer driver approach.

Programming Interface

You can find a description of all options used in the TTY export module in this chapter. The options can be modified using the methods *LIXSetParameter(..."TTY"...)* and read by calling *LIXGetParameter(..."TTY"...).*

Export.Path: Path where the exported PRN file should be saved.

Export.File: File name of the PRN file. If this option is empty, a file selection dialog will always be displayed.

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	Export with user interaction (dialogs)
1	No dialogs or message boxes will be displayed (only if *Export.Path* was specified).
Default	0

TTY.AdvanceAfterPrint: Specifies the behavior when the print job is finished.

Value	Meaning
FormFeed	Form feed
ToNextLabel	Advances to the next label
AfterNextLabel	Leaves one blank label as separator

TTY.Emulation: Specifies the emulation used for the export.

Value	Meaning
ESC/P	ESC/P emulation
ESC/P 9Pin	ESC/P emulation for 9-pin dot matrix printers
PlainTextANSI	Plain text ANSI emulation
PlainTextASCII	Plain text ASCII emulation
PlainTextUNICODE	Plain text Unicode emulation
NEC Pinwriter	NEC Prinwriter emulation
IBM Proprinter XL24	IBM Proprinter XL24 emulation

Value	Meaning
PCL	PCL emulation

TTY.Destination: Export target. Possible values are "LPT1:", "LPT2:",..."FILE:" or "FILE:<Filename>". If "FILE:" is used, a file selection dialog will be displayed.

TTY.DefaultFilename: Default file name for this dialog.

7.3.16 Windows Fax Export

You can send List & Label documents directly as a fax using the fax service of Windows. If you connect a fax modem to such an operating system, the fax driver will be automatically installed in most versions of these operationg systems.

Additional information is needed for automatic fax sending (that is, no dialog will be displayed for the fax destination, cover page etc.). You can preset these parameters using the LL_OPTIONSTR_FAX... option strings (see *LlSetOptionString()*).

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);
LlSetOptionString(hJob, LL_OPTIONSTR_FAX_RECIPNAME,
   "combit");
LlSetOptionString(hJob, L_OPTIONSTR_FAX_RECIPNUMBER,
   "+497531906018");
LlSetOptionString(hJob, LL_OPTIONSTR_FAX_SENDERNAME,
   "John Doe");
LlSetOptionString(hJob, LL_OPTIONSTR_FAX_SENDERCOMPANY,
   "Sunshine Corp.");
LlSetOptionString(hJob, LL_OPTIONSTR_FAX_SENDERDEPT,
   "Development");
LlSetOptionString(hJob, LL_OPTIONSTR_FAX_SENDERBILLINGCODE,
   "4711");
// ...
LlJobClose(hJob);
```

If these options are not set and the user has not entered any expressions in the fax parameters dialog, export to MS FAX will not be available.

This module has no programming interface.

Various established fax applications can be used from List & Label with the corresponding printer (fax) driver. If the fax application supports passing of the fax number by the document, the number input dialog can be suppressed in most cases. To use e.g. David from the company Tobit, you can use the @@-command. Place a text object in the Designer and insert the line:

```
"@@NUMBER "+<fax number resp. field name>+"@@"
```

The fax driver knows the syntax and sends the print job without any user interaction with the placed fax number. Other fax applications offer similar possibilities – we recommend taking a look at the documentation of your fax application.

7.3.17 Unsupported Export Formats

The following export formats are not supported anymore and are only available for compatibility reasons. If you still want to use the format you have to enable it explicitly via LlSetOption(hJob, LL_OPTIONSTR_LEGACY_EXPORTERS_ALLOWED,...) or via LL.Core.LlSetOptionString(...).

HTML Export

The HTML export module creates HTML 4.01 code (with some limitations).

Overview

The export module collects all List & Label objects for the page currently being printed and places them in a large HTML table (the layout grid) corresponding to their physical position on the page. The sizes of the columns and rows in this table are a result of the X and Y coordinates of all objects.

The end user can choose an option from the HTML export settings to determine whether the column widths of the layout grid should be values expressed as percentage (based on the current window size of the browser) or absolute values (in pixels). The advantage of absolute positions is that the result of the export is a more precise representation of the original layout (in the Designer). Representation with percentage positions has the advantage that it is normally more easily printable than the other representations. This is due to the fact that the browsers can resize this kind of representation.

Because each different X and Y coordinate results in another column or row in the layout grid, you should pay attention to the design of your layout. Objects should generally be aligned with the same edges. This results in a less complex layout grid, which can be loaded more quickly by the browser.

The HTML 4.01 standard does not support overlapping objects. When you have objects, which overlap in the original layout, only the object with the lowest order (the object printed first) will be exported. The other overlapping objects will be ignored. Exception: colored rectangle objects in the background. This effect is achieved by filling the cell (in the layout grid) of the next object over the rectangle.

Limitations

There are also other limitations set by the target format. The most important are listed now.

- Rows that are anchored to each other are not correctly exported.
- Overlapping objects (except rectangles) are not supported.
- Rectangles cannot have any frames. Transparent rectangles will be ignored.

- Decimal tabs will be transformed to right-aligned tabs.
- Any other tabs will be transformed to a blank.
- 'Paragraph spacing' and 'Line distance' in text objects are not supported.
- The option 'Line break' in text objects and table columns is always active in the export result.
- The option 'Separators fixed' in table objects is not supported.
- The left offset in the first column of a table line will be ignored.
- The list object option "fixed size" is not supported.
- The chart object is exported as a picture and thus cannot appear transparently.
- The transformation of RTF text to HTML code is carried out by an RTF parser. This parser interprets the basic RTF formatting. Extended RTF functionality such as embedded objects will be ignored.
- Rotated RTF text is not supported.
- Spacing before table lines is not supported.
- Horizontal and vertical lines are exported as images; all other lines are ignored.
- Gradient fills are not supported.
- Rotated text (RTF and plain) is not supported.
- Custom drawings in callbacks are not supported.
- Objects to be exported as picture should not overlap the object frame.
- Table frames of neighboring cells are not drawn so that they overlap each other, but discretely. This can double the frame width and needs to be taken into account during the design.
- Offset of table lines is not supported.
- TotalPages$() may not be used in rotated text objects.
- Shadow Pages are not supported.
- The wrapping option 'Minimum Size' in the crosstab object is not supported.
- The property "Link" is not supported.

The following tags or attributes superseding HTML 4.01 standard will be used:

- Ending the page frame for HTML pages will use browser specific tags.
- Setting line color for the table grid (<table BORDERCOLOR="#ff0000">) is browser specific.
- Setting line color for horizontal table lines (<hr COLOR="#ff0000">) is browser specific.

If the HTML object is not exported as picture but as HTML text, the part of the stream between the <body> and </body> tags will be embedded. This by definition leads to the following limitations:

- Cascading Style Sheets are not completely supported.
- Page formatting such as background color, margins etc. is lost.
- HTML does not allow scaling. The exported result may thus differ from the layout in the Designer, especially when the HTML object contains the contents of a whole web site.
- Even if the HTML object wraps over several pages, it will be exported in one stream, i.e. no page wrap will occur.
- Embedded scripting functionalities may be lost.
- Issue print is not supported.

Programming Interface

You can find a description of all options used in the HTML export module in this chapter. These options can be modified/read using the methods *LIXSetParameter(..."HTML"...)* and *LIXGetParameter(..."HTML"...).*

Resolution: Defines the resolution in dpi for the transformation of the coordinates and the generation of pictures. Default: *96 dpi* screen resolution.

Picture.JPEGQuality: Specifies the quality and the corresponding compression factor of the generated JPEG graphic. The value lies between 0 and 100, with 100 representing the highest quality (and therefore the least compression). Takes effect only when the source graphic is not the JPEG format, as encoding of JPEG to JPEG would result in a quality loss. Default: 75

Picture.BitsPerPixel: Defines the color depth of the generated picture. Please note that the picture files will quickly get very large with higher color depths.

Value	Meaning
1	Black & White
24	24bit True Color
Default	24

Verbosity.Rectangle: Configures how rectangle objects should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG (and also as a complete rectangle for objects with colored background).
Default	1

Verbosity.Barcode: Configures how barcode objects should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG

Default	1

Verbosity.Drawing: Configures how picture objects should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG
Default	1

Verbosity.Ellipse: Configures how ellipse objects should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG
Default	1

Verbosity.Line: Configures how line objects should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG
Default	1

Verbosity.Text: Configures how text objects should be exported.

Value	Meaning
0	Ignore object
1	Object as text object
2	Object as JPEG
Default	1

Verbosity.Text.Frames: Configures how text object frames should be exported.

Value	Meaning
0	Single frames for top, bottom, left, right (uses CSS)
1	Complete frame as box
Default	0

Verbosity.RTF: Configures how RTF objects should be exported.

Value	Meaning
0	Ignore object

1	As formatted RTF text (parsed and converted to HTML)
2	As unformatted text (uses the default font specified in the project file)
3	Object as JPEG
Default	1

Verbosity.RTF.Frames: Configures how RTF object frames should be exported.

Value	Meaning
0	Single frames for top, bottom, left, right (uses CSS)
1	Complete frame as box
Default	0

Verbosity.Table: Configures how table objects should be exported.

Value	Meaning
0	Ignore object
1	As a table object
Default	1

Verbosity.Table.Cell: Configures how table cells should be exported.

Value	Meaning
0	Ignore cell
1	As a cell object using the verbosity settings of the object types specified in the cell.
2	Cells as JPEG
Default	1

Verbosity.Table.Frames: Configures how table frames should be exported.

Value	Meaning
0	Ignore table frame
1	Only horizontal lines of table frames
2	The whole table line including all frames
3	Cell-specific frames (uses CSS)
Default	3

Verbosity.LLXObject: Configures how LLX objects (e.g. chart object) should be exported.

Value	Meaning
0	Ignore object

1	Object as JPEG
Default	1

Verbosity.LLXObject.HTMLObj: Configures how the HTML object should be exported.

Value	Meaning
0	Ignore object
1	Object as JPEG
2	Object as embedded HTML. Only the HTML text between the <BODY> and </BODY> tags will be exported. Please note the hint on exporting limitations.
Default	2

HTML.Title: Specifies the title of the generated HTML document.

Layouter.Percentaged: This option configures whether the layout should be defined in absolute values or with values expressed as percentage.

Value	Meaning
0	Layout of the X coordinates in absolute values (pixel)
1	Layout of the X coordinates with values expressed as percentage
Default	0

Layouter.FixedPageHeight: Configures whether all pages should be forced to have the same page height.

Value	Meaning
0	Layout can shrink on the last page (e.g. if no objects have been placed in the page footer)
1	The page height is set as configured
Default	1

Export.Path: Path where the exported files should be saved. If this option is empty, a file selection dialog will always be displayed.

Export.File: File name of the first HTML page. Default: "index.htm". You may also use printf format strings like "%d" in the file name (ex. "Export Page %d.htm"). In this case, the files for the pages will be named by replacing the placeholder with the correctly formatted page number. Otherwise, you will get a simple page numbering for the result files.

Export.AllInOneFile: Configures the export result format.

Value	Meaning
0	Every printed page will be exported in a single HTML file.
1	The result is a single HTML file (*Export.File)*, containing all printed pages.
Default	1

Export.Quiet: Use this option to configure the possibility of exporting without user interaction.

Value	Meaning
0	Export with user interaction (dialogs)
1	No dialogs or message boxes will be displayed (only if *Export.Path* was specified).
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the registered file extension.

Value	Meaning
0	Result will not be displayed automatically
1	Calls *ShellExecute()* with *Export.File.*
Default	0

Export.ShowResultAvailable: Enables you to hide the respective checkbox in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default	1

Hyperlinks

Hyperlinks can be embedded in text, table and RTF objects directly in the Designer using the *Hyperlink$()* function. Dynamic hyperlinks and formulas can be realized in this way.

Another way of creating hyperlinks is via the object name. The names of text and picture objects are parsed for the string "/LINK:<url>". Whenever this string is found, a hyperlink is generated. Thus, if you name your picture object "combit /LINK:https://www.combit.net", this would create a picture hyperlink during export to HTML.

JQM Export

Overview

The JQM (jQuery Mobile) export creates HTML formatted reports using the jQuery Mobile framework and Javascript. The created files are optimized for display on mobile devices. Information about JQM can be found on jquerymobile.com. The framework is loaded from a CDN (Content Delivery Network); therefore, an active internet connection is required for display.

Limitations

There are several limitations due to the target format. The most important are:

- The created pages are optimized for display on mobile devices.
- Only tables are exported and therefore only list projects are supported.
- Table of contents and index are not supported.
- Text, RTF text and HTML text in table cells are specially supported. All other objects will be exported as picture.
- Only footer lines of the last page or the last ones of a table are supported, as this export is not page based.
- With local access the according rights (IE) have to exist to be able to load the pages. With some browsers, access to file:// can cause problems. As soon as the pages are accessed via http://, the problem should not occur anymore.
- Issue print is not supported.
- Shadow Pages are not supported.
- Nested tables will be exported as image.

Programming Interface

You can find a description of all options used in the JQM export module in this chapter. These options can be modified/read using the methods *LIXSetParameter(..."JQM"...)* and *LIXGetParameter(..."JQM"...).*

Resolution: Defines the resolution in dpi for the transformation of the coordinates and the generation of pictures. Default: *96 dpi*, screen resolution.

Picture.JPEGQuality: Specifies the quality and the corresponding compression factor of the generated JPEG graphic. The value lies between 0 and 100, with 100 representing the highest quality (and therefore the least compression). Takes effect only when the source graphic is not the JPEG format, as encoding of JPEG to JPEG would result in a quality loss. Default: 75

Picture.BitsPerPixel: Defines the color depth of the generated pictures.

Value	Meaning

1	Black & White
24	24bit True Color
Default	32

Picture.Format: Defines the format of the generated pictures.

Value	Meaning
JPG	JPEG picture
PNG	PNG picture
Default	PNG

Export.Path: Defines the target path for the export with closing backslash "\". If this option is empty, a file selection dialog will always be displayed.

Export.File: File name of the first HTML page to be generated. Default: "index.htm".

Export.Quiet: Defines if the export should be executed with user interaction.

Value	Meaning
0	Interaction/dialogs allowed
1	No file selection dialog for the target path is displayed (in case Export.Path is set) and no "Overwrite?" query is made. Also no summary of overlapping objects that were ignored is displayed.
Default	0

Export.ShowResult: Specifies whether the export result will be displayed automatically. The program that displays the result will be determined by the registered file extension.

Value	Meaning
0	Result will not be displayed automatically.
1	Calls *ShellExecute()* with *Export.File* so that normally a web browser is started.
Default	0

Verbosity.RTF: Defines the way how RTF objects should be exported.

Value	Meaning
0	Ignore object
1	As formatted RTF text (converted to HTML)
2	As unformatted text
Default	1

Verbosity.LLXObject.HTMLObj: Configures how the HTML object should be exported.

Value	Meaning
0	Ignore object
1	Object as embedded HTML. Only the HTML text between the <BODY> and </BODY> tags will be exported. Please note the hint on exporting limitations.
Default	1

JQM.CDN: CDN provider of the CSS and JS files (Content Distribution Network).

Value	Meaning
jQuery	https://code.jquery.com
Microsoft	https://ajax.aspnetcdn.com
Default	jQuery

JQM.Title: Title of the generated HTML files. Default: "".

JQM.ListDataFilter: Specifies if a search filter bar should be displayed in the result.

Value	Meaning
0	No display of the search filter bar
1	Displays a search filter bar and filters the data accordingly
Default	1

JQM.UseDividerLines: Configures the usage of divider lines.

Value	Meaning
0	All lines of a table are output as a "normal" data line
1	Header lines, footer lines and group lines are output as special divider lines with an own style
Default	1

JQM.BreakLines: Configures the wrapping behavior of texts in the result.

Value	Meaning
0	Text won't be wrapped but marked with "..." at the end if the width is insufficient
1	Text is automatically wrapped

Value	Meaning
Default	1

JQM.BaseTheme: Theme of the data lines. Values: "a", "b", "c", "d", "e". See view.jquerymobile.com/master/demos/theme-classic.

Default: "d".

JQM.HeaderTheme: Theme of the headers (Line with navigation and header). Values: "a", "b", "c", "d", "e". See view.jquerymobile.com/master/demos/theme-classic.

Default: "a".

JQM.DividerTheme: Theme of the divider lines (see JQM.UseDividerLines). Values: "a", "b", "c", "d", "e". See view.jquerymobile.com/master/demos/theme-classic.

Default: "b".

7.4 Digitally Sign Export Results

By accessing the products digiSeal® office and digiSeal® server from secrypt GmbH or OpenLimit® CC Sign from OpenLimit® SignCubes, you can digitally sign PDF, TXT (if the option "***Export.AllInOneFile***" is set) and Multi-TIFF files generated with List & Label. Besides the software, you need a card reader and a signature card with a certificate stored on it. Details of hard and software requirements can be found in the signature provider's documentation.

The digiSeal® office package contains the digiSealAPI.dll file or the dsServerAPI.dll file in digiSeal® server, which must also be delivered together with the redistributable files of List & Label. The DLL's corresponding signature file (*.signatur) may also be required. Please note that you also will need a software certificate (*.pfx file) when using digiSeal® server. Detailed information can be obtained directly from secrypt GmbH.

OpenLimit® CC Sign requires an interface DLL that is provided by OpenLimit® SignCubes.

7.4.1 Start Signature

Check the "Digitally sign created files" checkbox in the export target dialog. Please note that this checkbox will only be available if one of the supported software suites is found on the machine.

After creation of the export file, the signature process is started as usual. Please note that this may change the file extension of the result file. If the signature process is aborted or erroneous, the export will continue after displaying the error reason, if applicable.

For legal reasons, a signature in "Quiet" mode is not possible; the PIN always needs to be entered interactively. This is only possible with the product digiSeal® server 2 as it is a server component requiring a mass signature card.

7.4.2 Programming Interface

The signature process can be influenced by many parameters.

Export.SignResult: Activates the signature of export files. This option corresponds to the checkbox in the export target dialog. The value is disregarded if no supported signature software is found on the machine.

Value	Meaning
0	No digital signature
1	Exported files will be signed digitally
Default	1

Export.SignResultAvailable: Can be used to suppress the checkbox for digital signature in the export target dialog.

Value	Meaning
0	Hide checkbox
1	Show checkbox
Default	1

Export.SignatureProvider: Allows selection of the software to be used if more than one of the supported products is installed.

Value	Meaning
0	Default, no explicit choice of signature software
1	Sign using secrypt digiSeal® office
2	Sign using OPENLiMiT® SignCubes software
3	Sign using esiCAPI® V 1.1 (not supported anymore)
4	Sign using secrypt digiSeal® server 2
Default	0

Export.SignatureProvider.Option: Additional options for the signature provider selected by Export.SignatureProvider.

Options for the signature provider "digiSeal® server 2":

This option has only one value and contains the connection data for digiSeal® server 2. The single values are separated with a pipe character each. The following structure applies:

<ServerHost>:<ServerPort>|<File path to the software certificate for identification and authentication >|<Password for the software certificate>

Example:

localhost:2001|secrypt_Testcertificate_D-TRUST_test.pfx|test

.NET component:

```
LL.ExportOptions.Add(LlExportOption.ExportSignatureProvider, "4");
LL.ExportOptions.Add(LlExportOption.ExportSignatureProviderOption,
"localhost:2001| secrypt_Testcertificate_D-TRUST_test.pfx|test");
```

C++:

```
LlXSetParameter(hJob, LL_LLX_EXTENSIONTYPE_EXPORT, _T("PDF"),
_T("Export.SignaturProvider "), _T("4"));
LlXSetParameter(hJob, LL_LLX_EXTENSIONTYPE_EXPORT, _T("PDF"),
_T("Export.SignaturProvider.Option"),
_T("localhost:2001|secrypt_Testcertificate_D-TRUST_test.pfx|test"));
```

Export.SignatureFormat: Can be used to choose the signature format. The available values depend on the file type and signature software.

Value	Meaning
pk7	Signature in pk7 format (container format that contains the signed file and signature). Available for Multi-TIFF, TXT and PDF (the latter only for SignCubes). The resulting file has the extension "pk7".
p7s	Signature in p7s format. An additional file *.p7s is created during the export process. Available for Multi-TIFF, TXT and PDF (the latter only for SignCubes). If the export result is sent via email, both files are attached.
p7m	Signature in p7m format (container format that contains the signed file and signature). Available for Multi-TIFF, TXT and PDF (the latter only for SignCubes). The resulting file has the extension "p7m".
PDF	PDF signature. Available for PDF and Multi-TIFF (only for digiSeal® office and B/W-Tiffs). A Multi-TIFF is converted to a PDF and signed with a special Barcode that allows verifying the signed document even after printing the PDF.
Default	p7s for TXT and Multi-TIFF, PDF for PDF.

7.5 Send Export Results via E-Mail

7.5.1 Overview

The files generated by the export modules can be sent via email automatically. List & Label supports MAPI-capable email clients, as well as direct SMTP mailing. This functionality is supported by all export modules except for TTY/MS fax export.

7.5.2 Setting Mail Parameters by Code

Like the other export options, some of the parameters for sending email can be set by code. The user can also predefine some settings in the Designer (Project > Settings). These are then used automatically. See the chapter "Project Parameters" for further details. A couple of other options can be set or read using *LIXSetParameter(..."<exporter name>"...)* / *LIXGetParameter(..."<exporter name>"...).* Note that the exporter name may be left empty to address all export modules.

Export.SendAsMail: Activates sending of export files via email. This option corresponds to the checkbox "Send exported files by email" for the end user.

Value	Meaning
0:	No mail will be sent
1:	The exported files are sent via the specified provider (see below)
Default:	0

Export.SendAsMailAvailable: Enables you to hide the respective checkbox in the dialog.

Value	Meaning
0:	Checkbox will be hidden
1:	Checkbox will be available
Default:	1

Export.Mail.Provider: This option can be used to switch the mail provider. All options apart from Simple MAPI need the CMMX26.DLL.

Value	Meaning
SMAPI	Simple MAPI
XMAPI	Extended MAPI
SMTP	SMTP
MSMAPI	Simple MAPI (using the default MAPI client)
Default	The default value depends on the system's or application's settings (see below)

If the DLL cannot be found, the mail will be sent using system Simple MAPI (MSMAPI).

The provider is selected by either setting it explicitly using this option or letting the user choose in the *LsMailConfigurationDialog()*.

List & Label first of all tries to retrieve the application-specific mail settings from the registry. These can be set using *LsMailConfigurationDialog()*. If your application wants to support sending report results by email then you should provide the end-user a menu-item (or similiar) in which's handler you call *LsMailConfigurationDialog()* to enable the end-user to specify the mail settings.

Export.Mail.To: Recipient for the email.

Export.Mail.CC: CC- Recipient for the email.

Export.Mail.BCC: BCC- Recipient for the email.

Export.Mail.From: Sender of the email.

Export.Mail.ReplyTo: Target for reply email if different to "From" (SMTP only).

Export.Mail.Subject: Mail subject.

Export.Mail.HeaderEntry:Message-ID: Message-ID of the email.

Export.Mail.Body: Mail body text.

Export.Mail.Body:text/plain: Mail body text in plain text format. Identical to *Export.Mail.Body.*

Export.Mail.Body:text/html: Mail body text in HTML format (SMTP and XMAPI only). Optional, otherwise the mail will be sent with the text defined in *Export.Mail.Body* or *Export.Mail.Body:text/plain.*

Export.Mail.Body:application/RTF: Mail body text in RTF format (XMAPI only).

Export.Mail.SignatureName: The name of an Outlook signature or the path and file name (without file name extension!) of a signature file. Depending on body text type, the file name extension txt, rtf or htm will be appended.

Export.Mail.AttachmentList: Additional attachments (besides the export results) as tab-separated list ("\t", ASCII code 9).

Export.Mail.ShowDialog: Selection for sending the mail without any further user interaction.

Value	Meaning
0:	The mail is sent directly without any further user interaction (at least 1 TO recipient must be set). If no recipient is set the dialog will be shown.
1:	The standard "Send mail" dialog is displayed. The values passed are preset there.
Default:	0

Export.Mail.Format: Set the default for the file format selection dialog in the preview. Valid values are: "TIFF", "MULTITIFF", "LL", "XML", "XFDF", "XPS", "PDF", "JPEG", "PNG", "TTY:<emulation>", "EMF".

Export.Mail.SendResultAs: Allows the result of an HTML export to be sent directly as HTML mail text.

Value	Meaning
text/html	If SMTP or XMAPI is chosen as mail provider, the export result is used as HTML content of the mail. All other mail providers will ignore this option.
empty	The HTML result is sent as attachment.
Default	Empty

Export.Mail.SignResult:

Value	Meaning
0	Email will not be signed.
1	Email will be signed.
Default	0

The SMTP provider offers a set of additional options. These generally do not need to be set explicitly, but should be set in the *LsMailConfigurationDialog()*.

Export.Mail.SMTP.SocketTimeout: Socket timeout, in milliseconds, default 1000.

Export.Mail.SMTP.LogonName: Login name for the server, default: computer name (usually unimportant).

Export.Mail.SMTP.SecureConnection: Connection security.

Value	Meaning
-1	Automatic (use TLS when server supports it)
0	Turn TLS off (even when it is supported by the server)
1	Force SSL (Cancellation when server does not support SSL)
2	Force TLS (Cancellation when server does not support TLS)
Default	-1

Export.Mail.SMTP.ServerAddress: SMTP server IP address or URL

Export.Mail.SMTP.ServerPort: SMTP server port, default 25.

Export.Mail.SMTP.ServerUser: SMTP server user name (if necessary)

Export.Mail.SMTP.ServerPassword: SMTP server password (if necessary)

Export.Mail.SMTP.ProxyType: Proxy type (0=none, 1=Socks4, 2=Socks5)

Export.Mail.SMTP.ProxyAddress: Proxy IP address or URL

Export.Mail.SMTP.ProxyPort: Proxy port, default 1080

Export.Mail.SMTP.ProxyUser: Proxy user name (only Socks5)

Export.Mail.SMTP.ProxyPassword: Proxy password (only Socks5)

Export.Mail.SMTP.POPBeforeSMTP: Some SMTP server need a login via POP before SMTP connection (0=no POP connection will be established, 1= POP connection will be established)

Export.Mail.SMTP.SenderAddress: Mail sender's address (ex. xyz@abc.def) – is also used for the SMTP protocol

Export.Mail.SMTP.SenderName: Real sender's name

Export.Mail.SMTP.ReplyTo: Reply to address (optional)

Export.Mail.SMTP.From: Substitutes the sender's address (combination of "Export.Mail.SMTP.SenderName" and "Export.Mail.SMTP.SenderAddress") in the mail. However, "Export.Mail.SMTP.SenderAddress" will still be used for the SMTP protocol.

Export.Mail.POP3.SocketTimeout: Timeout for socket connection in ms, default: 10000

Export.Mail.POP3.SecureConnection: Connection security.

Value	Meaning
-1	Automatic (use TLS when server supports it)
0	Turn TLS off (even when it is supported by the server)
1	Force SSL (Cancellation when server does not support SSL)
2	Force TLS (Cancellation when server does not support TLS)
Default	-1

Export.Mail.POP3.SenderDomain: Login domain, default: computer name

Export.Mail.POP3.ServerPort: default: 110

Export.Mail.POP3.ServerAddress: URL/IP address of POP3 server, default: "localhost"

Export.Mail.POP3.ServerUser: user for authentication

Export.Mail.POP3.ServerPassword: password for authentication

Export.Mail.POP3.ProxyAddress: proxy server address

Export.Mail.POP3.ProxyPort: proxy server port, default 1080

Export.Mail.POP3.ProxyUser: proxy server user name

Export.Mail.POP3.ProxyPassword: proxy server password

Export.Mail.XMAPI.ServerUser: profile name for authentication

Export.Mail.XMAPI.SuppressLogonFailure: "0" / "1" show (no) dialog for login error

Export.Mail.XMAPI.DeleteAfterSend: "0" / "1" delete mail after sending

Example:

```
LlXSetParameter(hJob,LL_LLX_EXTENSIONTYPE_EXPORT,"","Export.SendAsMail",
"1");
```

This automatically sends the export result as email to the recipient selected via Project > Settings. The globally selected mail settings will be used. If you want to offer a default to your user, this can be done quite simply:

```
LlSetDefaultProjectParameter(hJob,"LL.Mail.To", "EMAIL", 0);
```

This example assumes that your database contains a field called EMAIL. If you want to preset a specific address, please note that you need to use single quotes, as the passed parameter needs to be evaluated as the formula:

```
LlSetDefaultProjectParameter(hJob,"LL.Mail.To", "'abc@xyz.de'", 0);
```

7.5.3 Sending Mail via 64 Bit Application

To be able to send mails over Simple MAPI/XMAPI from a 32 Bit application via a 64 Bit application (e.g. Microsoft Outlook 64 Bit), use the mail proxy in the form of the two files cmMP26.exe/cxMP26.exe. These require registration via /regserver with administrator rights. Register both EXE to be able to address 32 Bit applications as well. For further information, please refer to the file Redist.txt in the List & Label installation directory.

7.5.4 Hints for Selecting the MAPI Server

For sending mail, the default mail application in the system is used. With following registry setting the load strategy of the MAPI DLL can be controlled.

```
HKCU\Software\combit\cmbtmx\<Appname>
MAPILoadStrategy [DWORD]
```

Value	Meaning
0	A direct LoadLibrary("mapi32.dll").
1	It will be tried to directly attach to olmapi32.dll or msmapi32.dll if these are already loaded. If that is not the case, it will be determined and loaded via GetDefaultMapiHandle() of the MAPISTUB code (see github.com/stephenegriffin/MAPIStubLibrary). The code corresponds to the API GetPrivateMAPI() in MAPISTUB. If this fails, MAPILoadStrategy 0 will be used.
2	The LoadDefaultMailProvider() method is used. If this fails, MAPILoadStrategy 1 will be used. It will be tried to use the MAPI-Unicode-API, meaning that with Microsoft Outlook also Unicode can be used in the text or subject.
Default	1 2 (Excemption: XMAPI, if the default mail client is Microsoft Outlook)

7.6 Export Files as ZIP Compressed Archive

Should, for example, the results of a picture or HTML export need to be sent by mail, it is often more practical, to send the export results as a ZIP archive. All export formats support a programming interface for this purpose. Data compression can be set by the user via a dialog, by selecting the option "ZIP archive (*.zip)" from the list of available file filters. Alternatively, the output can be controlled by code. The following options are available:

Export.SaveAsZIP: Activates the compression of exported data. If this option is set, the ZIP-Filter will be selected in the dialog.

Value	Meaning
0	Compression is not performed
1	The export data will be compressed into a ZIP archive
Default	0

Please note, that the user can modify the default settings via the dialog. If this is to be inhibited, set the option "Export.Quiet" to "1".

Export.SaveAsZIPAvailable: Here you can hide the ZIP archive filter within the file select dialog.

Value	Meaning
0	Filter hidden
1	User selection possible
Default	1

Export.ZIPFile: (Default-)Name of the ZIP file to be created e.g. "export.zip". For the file names in the ZIP archive the following rules apply:

- if "Export.File" is not assigned, the name of the ZIP archive is used with a corresponding file extension (e.g. "export.htm")
- if "Export.File" is assigned, this will then be used. If an export format generates one file per page, the placeholder "%d" can be used for the page number e.g. "Invoice Page %d.bmp" for the bitmap exporter

Export.ZIPPath: Path of the created ZIP files

8. Miscellaneous Programming Topics

8.1 Passing NULL Values

You can use NULL values in List & Label by passing a special string to the APIs. This means that this field has no current value, e.g. a delivery date for a shipment that has not yet occurred. Most database drivers may return a field content of NULL, which you need to pass on to List & Label as "(NULL)", although that string can be altered if needed. Basically the List & Label components handle database NULL values automatically.

List & Label handles NULL-values according to the SQL-92 specification where possible. An important effect of that is, that functions and operators, which get NULL-values as parameter or operator generally also return NULL as the result. An example is the following Designer formula:

```
Title+" "+Firstname+" "+Lastname
```

If Title is filled with NULL, the result of the formula is also NULL according to the standard. To change this behaviour please refer to the option *LL_OPTION_NULL_IS_-NONDESTRUCTIVE*.

8.2 Rounding

Please read the following important notes to ensure that you do not run into inconsistencies with the data from your database: sum variables are not rounded, so if you are not using only integers (i.e. invoices), we suggest that you use a rounding function, or (better) do not use multiplication or division for sum variables.

8.3 Optimizing Speed

List & Label's standard settings are a good compromise between file sizes and performance. Change the following settings and mind the following hints to tweak performance for mission critical applications:

- Make sure a job is opened all the time. This prevents the repetitive loading and unloading of DLLs.
- When printing to preview: switch off compression (see *LL_OPTION_COMPRESSSTORAGE*). Keep in mind that this might lead to considerably larger preview files, though.
- Set *LL_OPTION_VARSCASESENSITIVE* to 1. Note that this will mean that all variables and fields are treated case sensitive from the moment of this change. This may render existing projects unusable!

- Avoid using RTF- and HTML-text where possible and use the "normal" text object instead.

8.4 Project Parameters

List & Label enables you to set project specific parameters. The user may set these and the application may query the values at print time.

For example, List & Label uses these parameters itself for the fax and email settings. However, your own application may also save information to the project file in the same way, too.

8.4.1 Parameter Types

There are different types of parameters that are distinguished by the nFlags parameter passed to *LlSetDefaultProjectParameter()*. One of each of the three flag alternatives FORMULA/VALUE, PUBLIC/PRIVATE and GLOBAL/LOCAL needs to be used:

LL_PARAMETERFLAG_FORMULA (default)

The parameter is a formula that is evaluated at print time. The evaluated value is returned with *LlPrintGetProjectParameter()*.

LL_PARAMETERFLAG_VALUE

The parameter is a fixed value. This value is returned with *LlPrintGetProjectParameter()*.

LL_PARAMETERFLAG_PUBLIC (default)

The parameter can be changed within the Designer in the Project>Settings dialog, where the user can enter a formula or value.

LL_PARAMETERFLAG_PRIVATE

The parameter cannot be changed in the Designer.

LL_PARAMETERFLAG_GLOBAL (default)

The parameter is added to the print project parameter list and will be saved to the preview file if applicable (see *LlStgsysGetJobOptionStringEx()*).

LL_PARAMETERFLAG_LOCAL

The parameter is not added to the print project parameter list and will not be saved to the preview file, as it is only valid for the local user or machine.

If the parameters are passed using *LlSetDefaultProjectParameter()*, they represent the default values that the user may change according to his needs (if *LL_PARAMETERFLAG_PUBLIC* is set).

If the project is loaded afterwards (*LlDefineLayout()*, *LlPrint[WithBox]Start()*) the passed parameters will be replaced by those saved to the project file, i.e. they are overwritten. Unchanged parameters are not saved to the project file and thus remain with their default values. If required, you may set the *LL_PARAMETERFLAG_SAVEDEFAULT* to override this behavior. This is especially useful if the project parameter is queried before printing with *LlGetUserParameter()* to offer report parametrization to the user.

Note: you may not pass multiple parameters with the same name but different types!

8.4.2 Querying Parameter Values While Printing

After starting the printout using *LlPrint[WithBox]Start()*, the values for the project parameters can be queried using *LlPrintGetProjectParameter()*.

You may also change these values using *LlPrintSetProjectParameter()* or even add further parameters. As the parameters may be (see above) saved to the preview file and can be extracted from there using *LlStgsysGetJobOptionStringEx()*, you may consistently save your own information in this way. In the preview file, the parameters are saved with the prefix "ProjectParameter" before the actual name.

8.4.3 Predefined Project Parameters

List & Label uses project parameters for sending emails and faxes. The user may change and edit the values. As List & Label expects the contents to be a formula, it will be necessary to mask them as a string value ("…") whenever fixed values are used.

Example:

```
LlPrintSetProjectParameter(hJob,"LL.FAX.RecipNumber","\"+497531906018\"",0);
```

LL.FAX.Queue	LOCAL, PRIVATE
LL.FAX.RecipNumber	GLOBAL, PUBLIC [*LL_OPTIONSTR_FAX_RECIPNUMBER*]
LL.FAX.RecipName	GLOBAL, PUBLIC [*LL_OPTIONSTR_FAX_RECIPNAME*]
LL.FAX.SenderName	GLOBAL, PRIVATE [*LL_OPTIONSTR_FAX_SENDERNAME*]
LL.FAX.SenderCompany	GLOBAL, PRIVATE [*LL_OPTIONSTR_FAX_SENDERCOMPANY*]
LL.FAX.SenderDepartment	GLOBAL, PRIVATE [*LL_OPTIONSTR_FAX_SENDERDEPT*]
LL.FAX.SenderBillingCode	GLOBAL, PRIVATE [*LL_OPTIONSTR_FAX_SENDERBILLINGCODE*]

LL.MinPageCount	GLOBAL, FORMULA, PUBLIC
LL.ProjectDescription	GLOBAL, VALUE, PUBLIC
LL.IssueCount	GLOBAL, FORMULA, PUBLIC
LL.PageCondition	GLOBAL, FORMULA, PUBLIC
LL.PrintJobLCID	GLOBAL, FORMULA, PUBLIC

Further information on the project parameters can be found in the Designer manual.

Analog to the LL.FAX parameters LL.MAIL parameters exists, see chapter "Setting Mail Parameters by Code" for further information.

Parameters that are not defined prior to *LlDefineLayout()* or were defined with the PRIVATE-Flag are not editable.

For example, the application could pass the value for "LL.Mail.To" using an email field (here: "EMAIL") from a database:

```
LlSetDefaultProjectParameter(hJob,"LL.MAIL.To","EMAIL",0);
```

The user may then add a "FIRSTNAME" and "NAME" field in order to enhance the address:

FIRSTNAME + " " + NAME + " <" + EMAIL + ">"

The preview control automatically adopts the values of LL.FAX.* and LL.MAIL.*. In addition, the values are passed to the export modules – these also use the user-defined contents.

Please note a change in the behavior of the export modules up to version List & Label 9: if the user enters an email address in the settings dialog, the export result will always be sent to this address, regardless of the settings made using *LlXSetParameter()*. We recommend setting all mail settings using the project parameter API. Unchanged projects should behave as before.

8.4.4 Automatic Storage of Form Data

If you are using the form elements, it is possible to perform automatic storage after completion of the preview with the project parameters. Besides the automatic storage of form data these parameters can also be used to define the file names for sending e-mail out of the preview and for defining the default settings for saving out of the preview. For this purpose, you can use the following project parameters:

SaveAs.Format	Desired export format, e.g. "XML"

	Supported formats are "TTY", "PDF", "EMF", "XPS", "PRN", "TIFF" resp. "PICTURE_MULTITIFF", "JPEG" resp. "PICTURE_JPEG", "PNG" resp. "PICTURE_PNG", "LL" resp. "PRV", "XML" resp. "XFDF". Further information can be found in chapter "The Export Modules".
SaveAs.Filename	Output file name, e.g. "test.xml"
SaveAs.ShowDialog	Allows the save dialog to be enabled ("1") or disabled ("0")
SaveAs.NoSaveQuery	Disables the request as to whether the file should be saved after completion or not. Note: SaveAs.NoSaveQuery overwrites the value of SaveAs.ShowDialog, i.e. if no dialog should be shown, but the save query should be, the dialog will be shown anyway and vice versa.

Example:

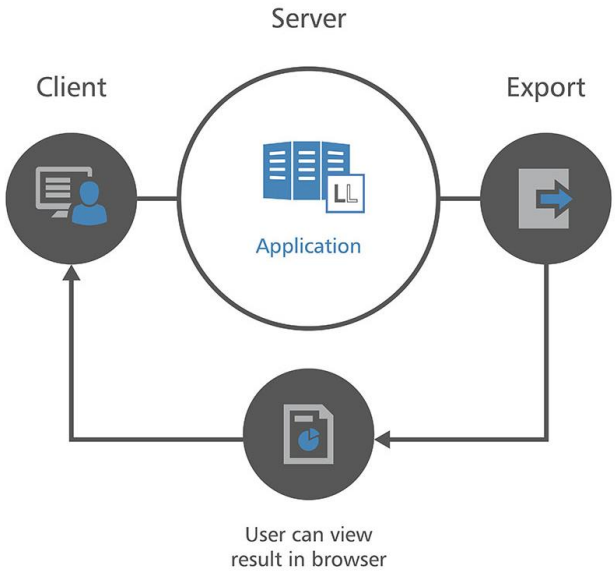
```
LlPrintSetProjectParameter(hJob,"SaveAs.Format","XML",
       LL_PARAMETERFLAG_VALUE);
LlPrintSetProjectParameter(hJob,"SaveAs.Filename","test.xml",
       LL_PARAMETERFLAG_VALUE);
LlPrintSetProjectParameter(hJob,"SaveAs.ShowDialog","0",
       LL_PARAMETERFLAG_VALUE);
LlPrintSetProjectParameter(hJob,"SaveAs.NoSaveQuery","1",
       LL_PARAMETERFLAG_VALUE);
```

8.5 Web Reporting

Using List & Label with ASP.NET is described in detail in chapter "Web Reporting". Of course you can also use other programming languages for web reporting. In general the following requirements apply:

- The server has to be a Windows system, List & Label can only be run on Windows platforms. This restriction of course does not apply to the clients.
- The user account, under which the web application is running, requires access to a printer driver. The dedicated printer does not have to be in the system physically, the simple driver installation is sufficient. Also it has to be ensured that the used user account can load the List & Label DLLs, meaning that the rights for the DLL path were assigned.
- The actual web application carries out a silent export without user interaction (see chapter "Export Without User Interaction") and directs the client to the export file e.g. by a redirect.

The following image visualizes the principle:



Please note the licensing requirements for using List & Label on a web server.

8.6 Hints for Usage in Multiple Threads (Multithreading)

List & Label can be used from multiple threads. This enables the distribution of large print jobs on multiple different processors / cores. Internally, this is used for the designer preview or drill down reporting.

If you plan to use List & Label in a multithreaded environment, keep the following issues in mind:

- Make sure that each List & Label job (resp. a component instance) is only used within a single thread, i.e. the creation, usage and destruction of the job/component needs to be done from the same thread. If you want to use multiple printing threads, each of these threads needs to open and close it's own job. Background: Windows GDI resources like window handles or printer device contexts cannot be used across different threads.
- Make sure to open a job/create a component instance in your application before starting the first print thread and do not close this job before all threads are terminated. Typically, this will be done in your application's start-up and shutdown code. Background: the first job creates a couple of helper objects that need to be destroyed in the same job. Also, this can increase the performance remarkably as it avoids steady loading and unloading of DLLs.
- Threads that open the designer need to use the Single Threaded Apartment (STA) concurrency model. This means you cannot use .NET worker threads from the thread pool for this task, as they are initialized to use the Multi Threaded Apartment (MTA) concurrency model. Background: List & Label needs to call OleInitialize() for drag & drop support within the designer, which requires the current apartment to be STA to succeed.

8.7 Scripting Support

8.7.1 Introduction

Scripting in List & Label provides you with a powerful extension that allows access to variables, fields and more. With the help of scripts you can address them and thus realize many additional functions comfortably in a language of your choice.

Note: A script is a sequence of commands that are processed sequentially during execution. The commands are taken from the "vocabulary" of a particular script language. This command set determines the possibilities offered by the language and how a script must be structured.

Scripts are usually not too extensive and lead to considerable performance with just a few commands. An average script contains maybe 20 to 40 lines of commands. Not least for these reasons script languages are usually very easy to learn.

Although superficially very similar, there are a number of crucial differences between a script and an executable program.

For example, scripts are not executable by themselves, but always need an environment in which they are executed. These so-called hosts are responsible for managing the scripts and usually extend the possibilities of the respective language in the form of additional objects. In this case List & Label itself is the host. By means of the provided frameworks, the user has a powerful interface to extend the functionality of the formula editor. Since List & Label scripts are usually run through frequently within the print loop, they must not contain any GUI elements. For the same reason, they should be executable within a manageable time frame.

Which Script Languages are Supported?

In principle, theoretically all script languages of the Windows Scripting Host are supported. However, the most common are VBScript and JScript, which are offered directly by the manufacturer Microsoft. But there are also other implementations such as Python. The selection of the language to be used is done by a parameter string when calling the script functions.

Note: VBScript and JScript are usually already installed on your system. If not or if other languages are to be used, they must be obtained from the respective manufacturer and installed on the system according to their specifications.

In addition to the classic scripting languages of the Windows Scripting Host, C# is also supported as scripting language.

For C# scripts, the full range of functions of the .NET Framework 4.0 is available to you, which is why at least this or a newer .NET Framework must be installed to execute C# scripts.

Since C# scripts must be compiled before use and this may take longer than executing the script itself, the 30 most recently compiled scripts are cached for re-execution. The corresponding files are located in the folder "%temp%\combitCSharpScriptCacheLL??\". There you will find the file "combitCSharpScriptCache.cache" which contains information about the stored scripts and for each script a folder with a name in the form "combitCSharpScript_[GUID]" (e.g. "combitCSharpScript_e9527e037aa149f3ba79bd408a8232db"). This folder contains all .dll files used by the script, debug information and the script itself (also in form of a .dll file).

How and Where can Scripts be Integrated?

The integration of scripts is easily possible via the formula editor integrated in List & Label. The actual script code can either be a direct embedded part of a formula or alternatively external code can be referenced using the LoadFile$() designer function. Furthermore, additional external code can be included within the script code using #include statements. Especially for larger scripts the use of external text files is the better choice, because they allow an easy reusability elsewhere.

The following designer functions are provided:

Script$: Returns the result of a script as a string

ScriptBool: Returns the result of a script as Boolean

ScriptDate: Returns the result of a script as date

ScriptVal: Returns the result of a script as a number

For more information about the Designer functions, see the Designer Manual.

Support for Scripting Functionalities

The possibilities of the scripting technology are very extensive and their description is therefore certainly material for your own book. Of course, we would like to help you with your questions and wishes within the scope of our support to enable you to use our product in an optimal way. However, we ask for your understanding that we can only answer questions about the object model itself, but not about models of other products or the script languages themselves. A description of the possible script languages can be found in the book program of many larger specialist publishers or online.

8.7.2 Preprocessor and Options

Enable Scripting Support

By default, the script engine is not active for security reasons, because it offers the user the possibility to call system functions via the script language in the context of the current application user. For this reason, the script engine must first be activated. There are three options available for this purpose.

Enable general scripting support (default: False).

```
LlSetOption(hJob, LL_OPTION_SCRIPTENGINE_ENABLED, true);
```

Optionally a user-defined timeout can be set for the maximum runtime of a script (default: 10000 ms). A script with longer runtime will be aborted by the environment. With C# scripting, caution is advised here with too low a timeout, since any compile time that may occur counts towards the execution time.

```
LlSetOption(hJob, LL_OPTION_SCRIPTENGINE_TIMEOUTMS, 15000);
```

Optionally, the formula editor can be set so that the printout is executed directly in real time each time a key is pressed. However, since this can put a lot of load on the system, depending on the script language, the default setting is False.

```
LlSetOption(hJob, LL_OPTION_SCRIPTENGINE_AUTOEXECUTE, true);
```

General for all Languages

All statements for the preprocessor such as pragmas, options, includes must always be placed on a separate line so that they can be handled correctly. Leading spaces and tabs are ignored. It is however possible to comment them out ad-hoc.

Single-line comments are initiated for all preprocessor instructions with // - analogous to C/C++/C#. Multiline comment blocks are not supported in the preprocessor.

Since formula parameters within a List & Label expression are marked with either ' or ", their use within a script is restricted if the source code is embedded directly into the formula. Then only the other character within the source code, which was not used to introduce the formula parameter, can be used for masking in the source code. However, if the source code is loaded from an external file instead using the LoadFile$() designer function, this restriction does not apply.

Selection of the Script Language

Using the command

```
<!--#language="[script language]"-->
```

the script language can also be set explicitly within the code. The usage is optional. If a specification is available, however, it is only checked and warned if different languages are mixed. The actual selection of the language is done via the corresponding parameter at the script call. Possible values are the identifiers needed for the Script$() designer function, such as "CSharpScript", "VBScript" or "JScript".

Nesting of Scripts

Frequently used functions can be stored and used centrally. This way changes affect all scripts based on them. For this purpose, the integration of scripts is supported via a special instruction:

```
<!--#include file="c:\scripts\include.vbs"-->
```

The statement is replaced by the complete contents of the specified file. #include statements like all other pragmas and options must always be placed on a separate line to be handled correctly by the preprocessor.

Note: All scripts included in this way must use the same script language as the main script itself. A mixture of several languages is not possible and leads to syntax errors.

If the scripts are located below the program directory of the application/EXE, the %APPDIR% variable can be used instead of a fixed directory specification:

```
<!--#include file="%APPDIR%\include.vbs"-->
```

Special for use With C#

Requirements

To run C# scripts, at least .NET Framework 4.0 or higher must be installed on the computer. Additionally the files combit.CSharpScript??.Engine.x86.dll and combit.CSharpScript??.Interface.x86.dll or the corresponding 64Bit versions must be delivered with your application and must be located in the search path of the List & Label main DLL.

Logging

To enable logging of a script, the following #pragma statement must be included:

```
<!--#pragma forcelogging-->
```

The log output is written to the file "%temp%\combitCSharpScript.log". Logging can be time consuming and should therefore not be activated by default.

Debug Mode

If a script contains the statement

```
<!--#pragma debugmode-->
```

at the beginning of the script a debugger is started (if one is installed on your system), which can be used to check the script step by step for errors. On the other hand, error texts triggered by exceptions contain more information and possibly the line numbers of the source code.

Adding References

Using the command

```
<!--#include ref="[file path]"-->
```

external references/components, such as the Windows Forms library from Microsoft (System.Windows.Forms.dll), can be added to a script.

If only the file name without path is specified, the system tries to load the reference from the "Global Assembly Cache" (GAC). If a complete path is specified, a copy of the file is created in a temporary folder and loaded from there.

Note: This command must be executed in the script according to the preprocessor directives that apply to all script languages.

Adding Namespaces

Using the command

```
<!--#include using="[namespace]"-->
```

can be added to the script using statements. This has the advantage that namespace names do not always have to be specified explicitly.

Instead of calling "System.Collections.Generic.List<String> obj;" for each individual list, the call after adding "System.Collections.Generic" as a using statement would only be "List<String> obj;".

Note: This command must be executed in the script according to the preprocessor directives that apply to all script languages.

8.7.3 Quick Reference and Examples

A script is called using the designer function Script$(<language>, <code>, <opt:function>,<opt:timeout>) in the project file within the formula editor and returns a string as result. Alternative forms like ScriptVal, ScriptBool and ScriptDate work analogously except for the return type. Details can be found in the Designer Manual.

Script$

Interprets the result of a script as a string.

Parameter:

String	Determines the script language to be used. Primarily CSharpScript as well as VBScript and JScript are supported
String	Script code to be executed
String	(optional) Defines the result of the return under VBScript, it contains either the name of the function/method to be executed or a variable name. For C# this parameter is ignored and values are returned directly by assigning the variable WScript.Result
Number	(optional) Timeout in ms

Return value:

String

Example:

Examples for C#:

```
Script$('CSharpScript',' WScript.Result= "Language: " +
Report.Variable("LL.CurrentLanguage"); ')

Script$('CSharpScript', LoadFile$(ProjectPath$(false) + "Script.cs"))
```

For further reference, the extended example of the sample application include the project "Order list with scripting.srt".

Examples for VBScript:

```
Script$('VBScript',' RetVal= "Language: " +
Report.Variable("LL.CurrentLanguage") ', 'RetVal')
```

```
Script$('VBScript', LoadFile$(ProjectPath$(false) + "Script.vbs"),
RetVal)
```

General Object Model

Within the script, provided variables and methods of the List & Label host can be accessed.

Report Object

The most important methods for accessing the tables and variable contents, temporary variables and the evaluation of formulas are provided by the report object.

When accessing variables and fields, always pay attention to the current context. Only those variables and fields that are actually registered in the current context can be accessed.

Report.Variable

Access a List & Label variable and return its value, read only.

Parameter:

String — Determines the name of the variable to be queried

Return value:

String

Example:

```
Script$('CSharpScript','Report.Variable("LL.CurrentContainer");')
```

Report.Field

Access a List & Label field and return its value, read only.

Parameter:

String — Defines the name of the field to be queried

Return value:

String

Example:

```
Script$('CSharpScript', 'Report.Field("Orders.CustomerID");')
```

Report.Eval

Evaluates a List & Label expression and returns its value, read only.

Parameter:

String — Determines the expression to evaluate

Return value:

String

Example:

```
Script$('CSharpScript', 'Report.Eval("RGBStr$(12345);')
```

The SetVar/GetVar designer functions can be used to indirectly pass intermediate results from one script to another during printing. However, the order of the calls (columns) is of course decisive here. See also the SetVar/GetVar documentation in the Designer Manual.

Example calls C#:

```
var start = Report.GetVar("ResultTmp"); // Get temporary value
var s1 = Report.Variable("LL.CurrentContainer");
var s2 = Report.Field("Orders.CustomerID");
var s3 = s1 + s2 + Report.Eval("RGBStr$(12345)");
Report.SetVar("ResultTmp", s3, false); // Set temporary value
```

Example calls VBScript:

```
start = Report.GetVar("ResultTmp")
s1 = Report.Variable("LL.CurrentContainer")
s2 = Report.Field("Orders.CustomerID")
s3 = s1 + s2 + Report.Eval("RGBStr$(54321)")
call Report.SetVar("ResultTmp", s3, false)
```

Report.SetVar

Sets a virtual List & Label variable.

Parameter:

String	Determines the name of the virtual variable to be set
All	Determines the value to be stored
Boolean	Determines whether the function should also return the value or whether the result should be an empty string. Default setting: Return (True)

Return value:

All

Example:

```
Script$('CSharpScript', 'Report.SetVar("ResultTmp", "MyValue",
false);')
```

Report.GetVar

Returns the value of a virtual variable.

Parameter:

String	Determines the name of the virtual variable to be queried

Return value:

All

Example:

```
Script$('CSharpScript','Report.GetVar("ResultTmp");')
```

WScript Object

The following constants are available for direct access in the script:

Name	Meaning
WScript.Name	Contains name of the application
WScript.Path	Contains path to the application
WScript.Version	Contains internal version number
WScript.FullName	Contains full name of the application

Example:

```
var MyFilePath= WScript.Path + "MyFile.txt";
WScript.Result = MyFilePath; // set the return value for a C# script
```

The variable WScript.Result is of special importance for a C# script, since it always serves as return value. In contrast to VBScript, for example, this return value is fixed and should be assigned at least once within the script. The last assignment defines the result.

In other script languages this variable is not defined and gives a syntax error when using it.

9. Error Codes and Warnings

Below are the possible constants for errors and warnings. The values in brackets represent the decimal specifications that also appear in debug output.

9.1 General Error Codes

Note: The constants all begin with *LL_ERR_*, i.e. the entry *BAD_JOBHANDLE* corresponds to the constant *LL_ERR_BAD_JOBHANDLE*.

Value	Meaning
BAD_JOBHANDLE (-1)	A function is called with a job handle not generated with *LlJobOpen()* or the job was closed.
TASK_ACTIVE (-2)	Only one Designer window can be open for each application; you have tried to open a second window (only if *hWnd* in *LlDefineLayout()* is NULL).
BAD_OBJECTTYPE (-3)	An invalid type was passed to a function which requires the type of object as a parameter. Valid types: *LL_PROJECT_LABEL*, *LL_PROJECT_LIST*, *LL_PROJECT_CARD.*
PRINTING_JOB (-4)	A print function was called, although no print job had been started.
NO_BOX (-5)	*LlPrintSetBoxText()* was called and the print job was not opened with *LlPrintWithBoxStart()*.
ALREADY_PRINTING (-6)	The current operation cannot be performed while a print job is open.
NOT_YET_PRINTING (-7)	*LlPrint[G|S]etOption[String]()*, *LlPrintResetProjectState()*. The print job has not started yet.
NO_PROJECT (-10)	*LlPrint[WithBox]Start()*:There is no object with the given object name. Identical to *LL_ERR_NO_OBJECT*
NO_PRINTER (-11)	*LlPrint[WithBox]Start()*: Printer job could not be started as no printer device could be opened. List & Label requires a printer driver to be installed.
PRINTING (-12)	An error occurred during print. Most frequent cause: print spooler is full. Other reasons: no sufficient disk space, paper jam, general printer failure.

EXPORTING (-13)	An error occurred during print (e.g. no access rights to the destination path, export file already existing and write-protected,…)
NEEDS_VB (-14)	This DLL version requires Visual Basic.
BAD_PRINTER (-15)	*LIPrintOptionsDialogTitle()*: No printer available.
NO_PREVIEWMODE (-16)	Preview functions: No preview mode is set.
NO_PREVIEWFILES (-17)	*LIPreviewDisplay()*: No preview files found.
PARAMETER (-18)	NULL pointer as a parameter is not allowed, other parameter errors are also possible. Please use the debug mode to determine the error.
BAD_EXPRESSION (-19)	New expression mode: an expression in *LIExpr-Evaluate()* could not be interpreted.
BAD_EXPRMODE (-20)	Unknown expression mode in *LISetOption()*.
CFGNOTFOUND (-22)	*LIPrint[WithBox]Start()*: Project file not found.
EXPRESSION (-23)	*LIPrint[WithBox]Start(),LIDefineLayout()*: One of the expressions used has an error. When starting the Designer, these errors will be displayed interactively. When printing, the errors are logged to Debwin. When working with *LIExprEval()* use *LIExprError()* to find the error.
CFGBADFILE (-24)	*LIPrint[WithBox]Start()*:Project file has the wrong format or is defective.
BADOBJNAME (-25)	*LIPrintEnableObject()*: The object name is not correct.
UNKNOWNOBJECT (-27)	*LIPrintEnableObject()*: No object exists with this object name.
NO_TABLEOBJECT (-28)	*LIPrint[WithBox]Start()*: No table available in the list mode.
NO_OBJECT (-29)	*LIPrint[WithBox]Start()*: The project has no objects, and empty pages can be printed another way!
NO_TEXTOBJECT (-30)	*LIPrintGetTextCharsPrinted()*: No text object in this project.
UNKNOWN (-31)	*LIPrintIsVariableUsed()*, *LIPrintIsFieldUsed(*: The given variable does not exist. *LIGetUsedIdentifiers():* The project contains no information about variables and fields used because it has not yet been saved with List & Label 11 or newer.
BAD_MODE (-32)	Field functions were used, although the project is not a list project.
CFGBADMODE (-33)	*LIPrint[WithBox]Start()*, *LIDefineLayout()*: The expression mode of the project file is the new mode, but the old mode is set (see *LISetOption()*).

ONLYWITHONETABLE (-34)	This error code can only be returned if - in the list mode - the OneTable mode (*LL_OPTION_ONLYONETABLE*) is chosen, but more than one table is present when loading the project with *LIPrint[WithBox]Start()*(See *LISetOption()*).
UNKNOWNVARIABLE (-35)	The variable given with *LIGetVariableType()* or *LIGetVariableContents()* has not been defined.
UNKNOWNFIELD (-36)	The field given with *LIGetFieldType()* or *LIGetFieldContents()* has not been defined.
UNKNOWNSORTORDER (-37)	The sorting order given by the ID for the grouping functions has not been defined.
NOPRINTERCFG (-38)	*LIPrintCopyPrinterConfiguration()*: File not found or wrong format.
SAVEPRINTERCFG (-39)	*LIPrintCopyPrinterConfiguration()*: File write error (network rights allow writing/creation, disk full?).
NOVALIDPAGES (-41)	Storage file contains no valid pages.
NOTINHOSTPRINTERMODE (-42)	This command cannot be called in HOSTPRINTER mode (*LISetPrinterInPrinterFile()*, *LIPrintCopyPrinterConfiguration()*).
NOTFINISHED (-43)	One or more objects have not been completely printed.
BUFFERTOOSMALL (-44)	*LI[G|S]etOptionString()*,*LIPrint[G|S]etOptionString()*, ... A buffer passed to List & Label is not large enough for the data that should be stored in it. The data is not complete.
BADCODEPAGE (-45)	*LL_OPTION_CODEPAGE*: The code page is invalid (NLS not installed).
CANNOTCREATETEMPFILE (-46)	A temporary file could not be created.
NODESTINATION (-47)	List & Label has no valid print medium (see *LL_OPTIONSTRING_EXPORTS_ALLOWED*).
NOCHART (-48)	*LIPrintDeclareChartRow()*: No chart object exists in the project.
TOO_MANY_CONCURRENT_-PRINTJOBS (-49)	Only in server/web server applications. The count of current users is above the number of licensed users.
BAD_WEBSERVER_LICENSE (-50)	Only in server/web server applications. The web server license file (*.wsl) is invalid or damaged.
NO_WEBSERVER_LICENSE (-51)	Only in server/web server applications. The webserver license file (*.wsl) could not be found.
INVALIDDATE (-52)	*LISystemTimeFromLocaleString():* invalid date format has been used.
DRAWINGNOTFOUND (-53)	A required drawing file could not be found, see also *LL_OPTION_ERR_ON_FILENOTFOUND*.
NOUSERINTERACTION (-54)	A call would require a user interaction, however the application is running on a web server.

BADDATABASESTRUCTURE (-55)	The database structure at design time and runtime does not match.
UNKNOWNPROPERTY (-56)	The property is not supported by the object.
INVALIDOPERATION (-57)	A DOM function could not be executed (e.g. an object could not be created).
PROPERTY_ALREADY_DEFINED (-58)	The property already exists (DOM).
CFGFOUND (-59)	The selected project already exists or is write protected.
SAVECFG (-60)	Error saving the project file.
ACCESS_DENIED (-65)	An existing object cannot be accessed in the repository.
IDLEITERATION_DETECTED (-66)	The number of print attempts is greater than the value of the *LL_OPTION_IDLEITERATIONCHECK_MAX_ITERATIONS* option.
USER_ABORTED (-99)	The user aborted the print.
BAD_DLLS (-100)	The DLLs required by List & Label are not on the required level.
NO_LANG_DLL (-101)	The required language DLL was not found and a CMLL26@@.LNG is not available.
NO_MEMORY (-102)	Not enough free memory.
EXCEPTION (-104)	An unhandled Exception happened inside a List & Label call. List & Label might be unstable.
LICENSEVIOLATION (-105)	Returned if the action (*LIDefineLayout()*) is not allowed with the current standard license, or if the program passes wrong licensing information in *LL_OPTIONSTR_LICENSEINFO*.

9.2 General Warnings

Note: The constants all begin with *LL_WRN_*, i.e. the entry *TABLECHANGE* corresponds to the constant *LL_WRN_TABLECHANGE*.

Value	Meaning
TABLECHANGE (-996)	The table name is changed in a hierarchical layout. See also chapter "Printing Relational Data".
PRINTFINISHED (-997)	Return value of *LIRTFDisplay():* no more data to print.

REPEAT_DATA (-998)	This is just a hint: present data record did not fit onto the page. This return value is required to remind the programmer that he can, for example, bring the number of pages up to date. The record pointer may not be moved.

9.3 Additional Error Codes of the Storage API

Note: The constants all begin with *LL_ERR_STG_*, i.e. the entry *NOSTORAGE* corresponds to the constant *LL_ERR_STG_NOSTORAGE*.

Value	Meaning
NOSTORAGE (-1000)	The file is not a List & Label preview file.
BADVERSION (-1001)	The version number of the preview file is incompatible.
READ (-1002)	Error reading preview file.
WRITE (-1003)	Error writing preview file.
UNKNOWNSYSTEM (-1004)	Unknown preview file format for the storage system.
BADHANDLE (-1005)	Bad parameter (invalid metafile handle).
ENDOFLIST (-1006)	*LlStgsysGetFilename()*: page not found.
BADJOB (-1007)	*LlStgsysxxx()*: job not found.
ACCESSDENIED (-1008)	Storage has been opened with ReadOnly flag and cannot permit write access.
BADSTORAGE (-1009)	Internal error in preview file, or empty file.
CANNOTGETMETAFILE (-1010)	*LlStgsysDrawPage()*: metafile could not be created (possibly defective preview file).
OUTOFMEMORY (-1011)	Not enough memory for current operation.
SEND_FAILED (-1012)	Error whild sending mail. Further information can be found in the debug logfile.
DOWNLOAD_PENDING (-1013)	An action could not be compled because the file to view could not be loaded completely.
DOWNLOAD_FAILED (-1014)	An action could not be compled because the attempt to load the file failed.
WRITE_FAILED (-1015)	Write or access error when saving a file.
UNEXPECTED (-1016)	Unexpected error. Further information will be displayed.
CANNOTCREATEFILE (-1017)	Write or access error when saving a file.
INET_ERROR (-1019)	Unexpected error. Further information will be displayed.

9.4 Additional Warnings of the Storage API

Note: The constants all begin with *LL_WRN_STG_*, i.e. the entry *UNFAXED_PAGES* corresponds to the constant *LL_WRN_STG_UNFAXED_PAGES*.

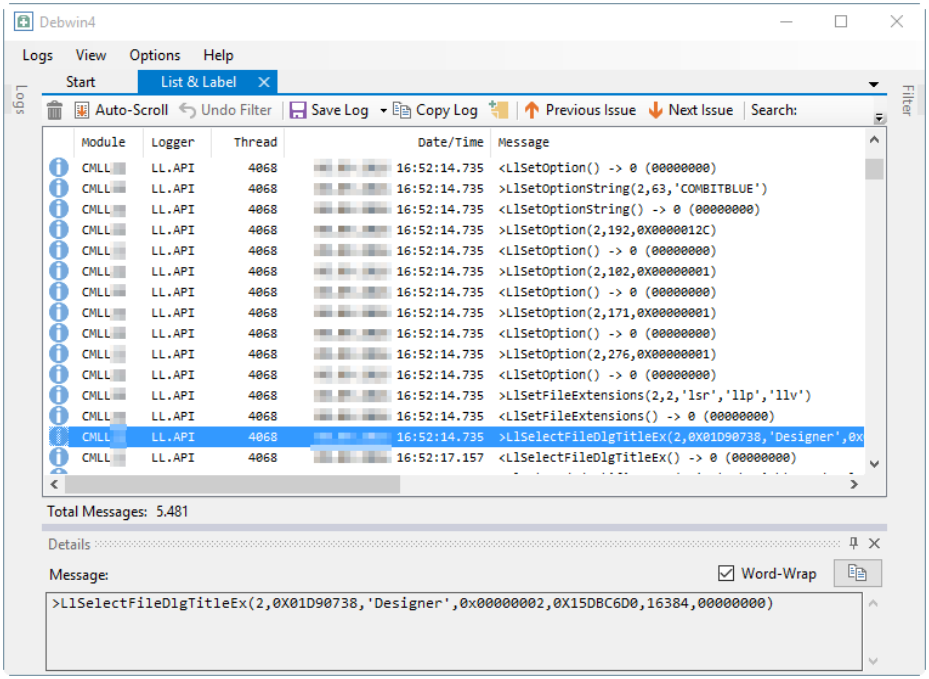
Value	Meaning
UNFAXED_PAGES (-1100)	*LlStgsysPrint() and LlStgsysStoragePrint()* (only while printing to MS FAX): some pages did not include a phone number and could not be faxed.

10. Debug Tool Debwin

Debwin is a tool for manifold debug functions.

If the debug mode is switched on with *LlSetDebug(),* List & Label outputs status information in the Debwin window. To get all available information we suggest to start Debwin before starting the application. As soon as your application is started, it will start sending debug messages to Debwin.

Besides the error codes (see chapter "Error Codes"), you often get additional information that helps to trace the reasons for unexpected program behavior. A typical debug output looks like this:



You can see the called module (CMLL??), timing information, the current thread ID, the called function including all parameters and – in the following line – the return value of the call. A full debug log, which is also often requested by our support team, contains many such output lines.

Further information can be found in our knowledgebase article Troubleshooting Guidance.

We provide support for this tool only for the features that are directly related to the debug output.

11. Redistribution: Shipping the Application

The List & Label DLL and its modules can be installed under Windows for side-by-side use in your application.

The Redistribution Assistant is available for putting together the redistribution files. In just a few steps, you can put together all the files required for your application and copy them directly to the correct target directory, create a ZIP archive or copy the file paths for further batch processing. The Redistribution Assistant can be found in the List & Label installation directory.

You can also find an overview of the files required for distribution in the Redist.txt file in the List & Label installation directory.

11.1 System Requirements

Please refer to "System Requirements", as the redistribution modules of List & Label have the same requirements.

11.2 The Standalone Viewer Application

11.2.1 Overview

LLVIEW26.EXE is a standalone application for viewing and printing the List & Label preview files.

Once the viewer is registered, the file extension ".ll" is linked to this viewer, so whenever a user double-clicks on files with this extension, the viewer is started.

11.2.2 Command Line Parameters

LLVIEW26 <file name>

Loads the file.

No URL can be given here.

LLVIEW26 /p <file name>

Prints the file (with a printer dialog).

LLVIEW26 /pt <file name> <printer name>

Prints the file using the given printer. If the printer name contains spaces, enclose it in quotation marks.

11.2.3 Registration

Your setup program should call the viewer once with the "/regserver" option in order to register it with the file extension "LL".

Unregister using "/unregserver".

11.2.4 Necessary Files

LLVIEW26.EXE needs CMLL26.DLL, CMDW26.DLL, CMCT26.DLL, CMBR26.DLL, CMLS26.DLL and CMUT26.DLL. It also requires at least one language resource file (e.g. CMLL2601.LNG). Depending on the direct export functionality required, you also need:

- CMLL26XL.DLL if direct PDF export functionality should be supported

11.3 List & Label Files

List & Label saves the project definitions in single files. In addition to the basic definition, some special settings such as target printer, printer configuration etc, which can be made in the Designer, are saved in a special, so-called "P-File". A small sketch of the form, which is displayed in the file open dialog, is saved to another extra file (the so-called "V-File").

File extension:	Form	Printer-Definition	Sketch for dialog
Label project	.lbl	.lbp	.lbv
File card project	.crd	.crp	.crv
List project	.lst	.lsp	.lsv

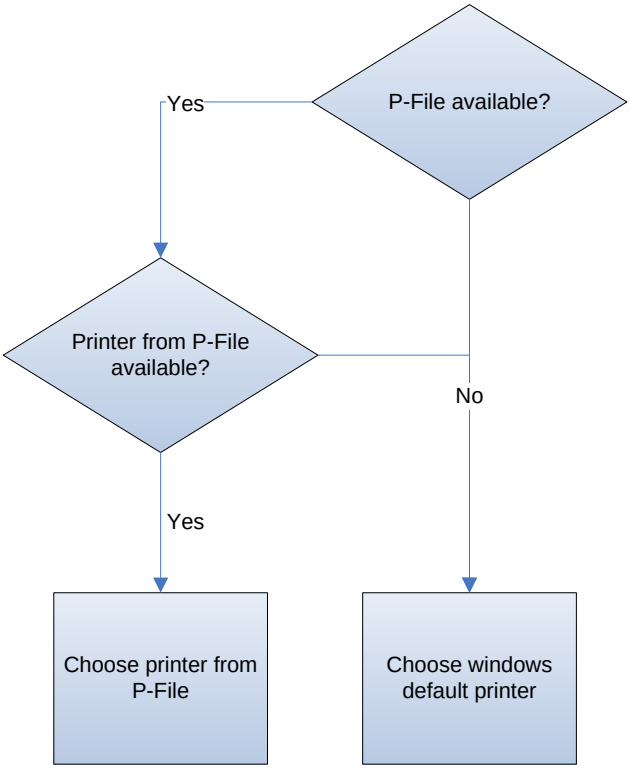
These file extensions are only default values and can be changed using *LlSetFileExtensions()* or *LlSetOptionString()*.

The crucial file is the form definition file.

The "V-File" can be created at any time using *LlCreateSketch()*.

In the printer configuration file ("P-File") are all export specific settings stored as well as the printer specific settings. Usually this file is created by the end user – you should not redistribute it with your application as the user typically will not have your printer available.

If the "P-File" cannot be found, List & Label automatically chooses the Windows default printer and - if the Designer is used - generates a P-file for it. The path where the P-File is searched for or generated can be specified with *LlSetPrinterDefaultsDir()*, which may be relevant for network applications. The logical sequence of choosing the printer is shown in the following graph:



When printing to preview, List & Label generates a file which contains all printed pages as graphics. This file has the fixed extension .LL and may be viewed at any time with the stand-alone application LLVIEW. The path where the LL-file is created may be specified with *LlPreviewSetTempPath()*.

Printing to Export will create files in a directory that can be defined by the host application and/or the user. Please refer to the export module's documentation for further details. A list of files created can be queried using *LL_OPTIONSTR_EXPORTFILELIST* after the print has been finished.

Whenever a project file is saved, a backup file is created. The name of the backup file's extension consists of a ~ and the project's extension (eg. "~lst" for the default list project extension).

If the target does not support long file names, the extension will be cut off after the third character (e.g. "~ls").

11.4 Web Designer Setup

You can find the List & Label Web Designer Setup in the "Redistributable Files" directory in your List & Label installation directory both as a conventional interactive installation program (file extension: .exe) and in a Windows Installer variant for installation via command line, e.g. for distribution via group policy (file extension: .msi).

11.4.1 Command Line Options for Windows Installer Setup

Note: The following calls must be executed with administrator rights.

Installation with user interface (please note that the interface is only available in English):

```
msiexec /i "C:\Program Files (x86)\combit\LL26\Redistributable Files
\LL26WebDesignerSetup.msi"
```

Installation without user interface into the default installation directory "C:\Program Files (x86)\\Web Designer 26":

```
msiexec /i "C:\Program Files (x86)\combit\LL26\Redistributable Files
\LL26WebDesignerSetup.msi" /q
```

Installation without user interface into an own installation directory, here "C:\Program Files (x86)\Test\Web Designer 26":

```
msiexec /i "C:\Program Files (x86)\combit\LL26\Redistributable Files
\LL26WebDesignerSetup.msi" /q INSTALLDIR="C:\Program Files (x86)\Test\Web
Designer 26"
```

For more information about the Windows Installer command line options, see docs.microsoft.com/en-us/windows/desktop/msi/command-line-options.

11.5 Other Settings

The following data are managed dependent on the main application (i.e. the program name of the task):

- Language
- Dialog design
- Dialog positions
- Designer configurations (colors, fonts)

This means that the configurations set by you or your program regarding language and dialog design or the dialog positions and the Designer configurations chosen by the user are only valid for your application. In this way, your application does not need to pay attention to whatever configurations are used in other applications.

These settings are stored in the registry at HKEY_CURRENT_USER/Software/combit/CMBTLL/<application name>.

12. Update Information for Version 26

Update information for older versions can be found in the separate PDF document "Update-Information". The overview of new features and changes in Report Server can be found in the Report Server manual.

12.1 Overview of new Features

- Different projects can be included in a print job by passing the project names semicolon-separated (requires data provider).
- Support for automated n-pass (requires data provider).
- Revised Drag & Drop UI for columns in tables.
- Support for sub reports (requires report container).
- New PDF export library. Improved output especially for complex graphics, smaller result files, CMYK image support, improved handling of Unicode fonts, selection of PDF version.
- Copies and page changes can be defined in label projects depending on formulas.
- The Viewer for preview files can now also display simple PDF files. No support for advanced PDF features such as form controls, signature fields or annotations.
- Greatly improved performance for C# scripts.
- Multi-page and poster printing.
- Print options dialog revised.
- Office 2019 ribbon UI for designers.

12.2 Overview of Changes

12.2.1 .NET

- The .NET Framework 4.0 is no longer supported, instead the .NET Framework 4.7 is the new base framework for .NET 4.x.
- In addition to .NET Core 3.1 now also support for .NET 5.
- More NuGet packages for data providers.
- Ad-hoc Designer with new features: Filter data source in charts, horizontal bar charts, support for images.
- SQL data sources can now be sorted by 1:1 fields.
- Parameterized data sources can work with NULL values as parameter defaults.
- The WebDesigner class names have been changed, a search and replace of WebDesigner → WindowsClientWebDesigner is required for the conversion.

- The namespaces no longer contain a version number. All namespaces whose names would have started with "combit.ListLabel26" now start with "combit.Reporting". This makes the version changeover much easier in the future.
- The default for SupportedElementTypes for many data providers has been changed from SupportedElementTypes to SupportedElementTypes.Tables | SupportedE-lementTypes.Views, so that the Designer now also displays the views by default.
- Casing of the parameters in the RegisterRoutes method of the WindowsClient-WebDesignerConfig class adapted.
- DOM collections now no longer use CollectionBase, but Collection<T> as base class.
- MySqlDataProvider is now contained in a separate assembly and is no longer part of combit.ListLabel26.dll.

12.2.2 General

- The export formats HTML and jQuery Mobile (JQM) are no longer supported and are only included for compatibility reasons. By default, they are no longer displayed in the export dialog. If you still need these formats, for example, for sending mails (HTML) or for display on a mobile device (jQuery Mobile), you have to enable them explicitly via LlSetOption(hJob, LL_OPTIONSTR_LEGACY_EXPORTERS_ALLOWED,...) or via LL.Core.LlSetOptionString(...).

12.2.3 API

- The default of the LL_PRNOPT_JOBPAGES option has been changed from 16 to INT_MAX.
- The LL_PRNOPT_PRINTDLG_ALLOW_NUMBER_OF_FIRST_PAGE option has been added. This allows you to set the page number in the print dialog that starts on the first printed page.

12.3 Updating to List & Label 26

12.3.1 General

Please make sure to update your personal license key, since the key is version and user specific.

As with any software update, we highly recommend that you check all project and template files after updating. Improvements can sometimes mean that things are done slightly differently, which might have an unexpected impact on your projects.

12.3.2 Updating .NET Projects

Generally, the only thing to do is to replace the reference to the combit.ListLabel25.dll with a reference to the combit.ListLabel26.dll and to update the namespace references (from combit.ListLabel25... to combit.Reporting...). Additionally, the older components should be removed from the toolbox and be replaced with the new components.

12.3.3 Updating Projects Using the OCX (e.g. Visual Basic)

- Load the Visual-Basic project (*.vbp resp. *.mak) in a text editor. Replace the line

  ```
  Object={2213E283-16BC-101D-AFD4-040224009C18}#25.0#0; CMLL25O.OCX
  ```

  with the following

  ```
  Object={2213E283-16BC-101D-AFD4-040224009C19}#26.0#0; CMLL26O.OCX
  ```

  and the line

  ```
  Module= CMLL25; CMLL25.BAS
  ```

  with the line

  ```
  Module=CMLL26; CMLL26.BAS
  ```

- After saving your changes, load the form (*.frm) in a text editor, which contains the List & Label OCX. Replace the line

  ```
  Object="{2213E283-16BC-101D-AFD4-040224009C18}#25.0#0";"CMLL26O.OCX"
  ```

  with the following

  ```
  Object="{2213E283-16BC-101D-AFD4-040224009C19}#26.0#0";"CMLL26O.OCX"
  ```

- If you wish to convert older List & Label versions, change the corresponding entries analogously. If you use the UNICODE OCX control, you need to adapt the control GUID. The new GUID is {2213E283-16BC-101D-AFD4-040224009DF9}.
- You can now load your projects in Visual Basic. The source code must be significantly adapted according to the original version.
- From VB 5 the .BAS declaration file is not necessary, because the List & Label constants are contained in the OCX control.
- Please note that it is not possible to host differently versioned OCXes (e.g. version 25 and 26) within the same application.

12.3.4 Updating Projects Using the VCL (e.g. Delphi)

See the hints in the Delphi online help.

12.3.5 Updating Projects Using the API (e.g. C/C++)

- Adjust the reference to the declaration file to the current version (e.g. for C/C++ #include "cmbtll25.h" to #include "cmbtll26.h")

- Adjust the reference to the corresponding import library analogously (e.g. in C/C++ in the linker settings cmbtll25.lib to cmbtll26.lib)

13. Help and Support

Many tips and tricks can be found in our knowledgebase at https://forum.combit.net/c/knowledgebase/english. The knowledgebase will be updated regularly and more articles will be added - so it's worth taking a look!

Information on the support concept can be found at https://www.combit.com/reporting-tool/support/.

Requirements:

Before contacting us, please check the following points or obtain the required information:

- First, read the latest notes in the Service Pack's "What's new" PDF document. You can find it during the Service Pack installation or in the Service Pack download area at https://support.combit.net/en/list-label-service-packs/.
- Please also note the information in our knowledgebase article Troubleshooting Guidance.
- For written inquiries, use the support form at https://support.combit.net/en/support-center/.

14. Index

.

.NET 14

1

1:1 relations 109, 116, 126
1:n relations 109, 113, 116

A

Abort box 242, 250
Access 30
AdoDataProvider 26
Alias 290
API Reference 153
AutoDefineField 34
AutoDefineNewLine 34, 58
AutoDefineNewPage 34, 58
AutoDefineVariable 34
AutoDestination 25
AutoFileAlsoNew 25
AutoMasterMode 25, 32
AutoProjectFile 25
AutoProjectType 25, 35
AutoShowPrintOptions 25
AutoShowSelectFile 25

B

Barcode 33
Barcode size 281
Barcode Variables 99
Boolean Variables 97

C

C/C++ 14
C++ Builder 14
Callback 129
Callbacks 128, 132, 145
Cards 36
Charts 58
  Handling 143
  Programming 108
Code page 269
Component
  Properties 22, 23, 25
Components 18
Concepts 25
Copies 36, 107, 233, 240, 243
Create log file 43
Create new project 25
Cross tables 58
Crosstabs 108
CSS 379
Currency symbol 273, 283

D

Data provider 25, 116
Data Source 21
Data transfer 21
Data type
  Barcode 33
  Date 33
  Drawing 33
  HTML 34
  Logic 33
  Numeric 33
  RTF 33
  Text 33
Data types 31
Database independent 58
DataBinding 21
DataMember 32
DataProviderCollection 27
DataSource 18, 27
DataTable 26

DataView 26
DataViewManager 26
Date Variables 96
DateTime 33
DB2 61
DbCommandSetDataProvider 28
Debug mode 261
Debugging 16, 42, 161, 262
Debwin 16, 43, 444
Decimal char 283
Declaration files 15
Default export format 25
Default printer 202
Default project file 25
Delphi 14
Design 21
Designer 11
  Direct printing 133
  Edit 36
  Extend 36, 62
DesignerFunction 62
Dialog styles 153
Digital Signature 413
DOC export 365
DOM 40
  Examples 149
  Functions 145
  Units 148
Drawing Variables 98
DrawObject 34
DrawPage 34
DrawTableField 34
DrawTableLine 34
Drilldown reports 136

E

E-mail export 416
Entity Framework 28
EntityCollection 28
Error Codes 438
Events 34
Examples 53
Excel export 358
Expandable Regions 141
Export 14, 23, 61, 350, 427
  Digital Signature 413
  Excel 358
  Fax 402
  Formats 24
  HTML 403
  JQM 410
  MHTML 386
  PDF 354
  Picture 396
  PowerPoint 370
  Restrict formats 62
  RTF 373
  Send via E-Mail 416
  SVG 399
  Text (CSV) 388
  Text (Layout) 386, 390
  TTY 401
  Without user interaction 61
  Word 365
  XHTML/CSS 379
  XML 392
  XPS 378, 403
  ZIP 421
Export media 284
Export modules 284
Export options 295, 352
Extended MAPI 416
External$ 299

F

Fax export 402
Faxing 286
Fields 13, 25, 31, 35
File Extension 446
File extensions 263
File selection dialog 25
File types 22
FileExtensions 22
Filter 225, 248
FINALIZE event 139
First page 243

Font 270, 272, 283
Footer 229
Free content 109, 113
Functions
  Script$ 434, 435

G

Group header option 272

H

Help file 202
HTML 34, 98
HTML export 403

I

IDbCommand 28
IEnumerable<T> 28
IListSource 28
ILLDataProvider 116
  ApplyFilter 125
  ApplySortOrder 124
  DefineDelayedInfo 121
  DefineRow 122
  Dispose 123
  GetOption 126
  GetRowCount 120
  MoveNext 122
  OpenChildTable 120
  OpenTable 119
  SetOption 125
  SetUsedIdentifiers 123
Import-Libraries 92
Instantiation 19
Integration 19
Interactive Sorting 142
Invoice 32
Invoice Merge 55
Issues 36
ITypedList 28

J

Java 15
Job Handle 347, 438
Job number 234
JQM export 410
JScript 430

L

Label 54
Labels 35
Landscape 36
Language 202
Languages 13
Last page 243
Licensing 20
LINQ 28
List 55
List<T> 28
ListLabel 18
ListLabelDocument 18
ListLabelPreviewControl 18
ListLabelRTFControl 18
Lists 35
LL_BARCODE_ 99
LL_BARCODE_...
  SSCC 99
LL_BOOLEAN 97
LL_CHAR_...
  LOCK 95
  NEWLINE 95
  PHANTOMSPACE 95
LL_CMND_...
  DRAW_USEROBJ 296
  EDIT_USEROBJ 297
  ENABLEMENU 299
  EVALUATE 299
  GETVIEWERBUTTONSTATE 300
  HELP 301
  HOSTPRINTER 281
  MODIFYMENU 179, 302
  OBJECT 302
  PAGE 304

PROJECT 305
SAVEFILENAME 306
SELECTMENU 307
TABLEFIELD 307
TABLELINE 309
VARHELPTEXT 310
LL_DATE 96, 97
LL_DATE_...
DELPHI 97
DELPHI_1 97
MS 97
OLE 97
VFOXPRO 97
LL_DESIGNEROPTSTR_...
PROJECTDESCRIPTION 182
PROJECTFILENAME 182
WORKSPACETITLE 182
LL_DRAWING 98
LL_DRAWING_...
HBITMAP 99
HEMETA 99
HICON 99
HMETA 99
USEROBJ 99
USEROBJ_DLG 99
LL_ERR_...
ACCESS_DENIED (-65) 441
ALREADY_PRINTING (-6) 438
BAD_DLLS (-100) 441
BAD_EXPRESSION (-19) 439
BAD_EXPRMODE (-20) 439
BAD_JOBHANDLE (-1) 438
BAD_MODE (-32) 439
BAD_OBJECTTYPE (-3) 438
BAD_PRINTER (-15) 439
BAD_WEBSERVER_LICENSE (-50) 440
BADCODEPAGE (-45) 440
BADDATABASESTRUCTURE (-55) 441
BADOBJNAME (-25) 439
BUFFERTOOSMALL (-44) 286, 440
CANNOTCREATETEMPFILE (-46) 440
CFGBADFILE (-24) 439
CFGBADMODE (-33) 439
CFGFOUND (-59) 441
CFGNOTFOUND (-22) 439
CONCURRENT_PRINTJOBS (-49) 440
DRAWINGNOTFOUND (-53) 440
EXCEPTION (-104) 441
EXPORTING (-13) 439
EXPRESSION (-23) 439
IDLEITERATION_DETECTED (-66) 441
INVALIDDATE (-52) 440
INVALIDOPERATION (-57) 441
LICENSEVIOLATION (-105) 441
NEEDS_VB (-14) 439
NO_BOX (-5) 438
NO_LANG_DLL (-101) 441
NO_MEMORY (-102) 441
NO_OBJECT (-29) 439
NO_PREVIEWFILES (-17) 439
NO_PREVIEWMODE (-16) 439
NO_PRINTER (-11) 438
NO_PROJECT (-10) 438
NO_TABLEOBJECT (-28) 439
NO_TEXTOBJECT (-30) 439
NO_WEBSERVER_LICENSE (-51) 440
NOCHART (-48) 440
NODESTINATION (-47) 440
NOPRINTERCFG (-38) 440
NOT_YET_PRINTING (-7) 438
NOTFINISHED (-43) 440
NOTINHOSTPRINTERMODE (-42) 440
NOUSERINTERACTION (-54) 440
NOVALIDPAGES (-41) 440
ONLYWITHONETABLE (-34) 440
PARAMETER (-18) 439
PRINTING (-12) 438
PRINTING_JOB (-4) 438
PROPERTY_ALREADY_DEFINED (-58) 441
SAVECFG (-60) 441
SAVEPRINTERCFG (-39) 440
TASK_ACTIVE (-2) 438
UNKNOWN (-31) 439
UNKNOWNFIELD (-36) 440
UNKNOWNOBJECT (-27) 439
UNKNOWNPROPERTY (-56) 441

UNKNOWNSORTORDER (-37) 440
UNKNOWNVARIABLE (-35) 440
USER_ABORTED (-99) 441
LL_ERR_STG...
ACCESSDENIED (-1008) 442
BADHANDLE (-1005) 442
BADJOB (-1007) 442
BADSTORAGE (-1009) 442
BADVERSION (-1001) 442
CANNOTCREATEFILE (-1017) 442
CANNOTGETMETAFILE (-1010) 442
DOWNLOAD_FAILED (-1014) 442
DOWNLOAD_PENDING (-1013) 442
ENDOFLIST (-1006) 442
INET_ERROR (-1019) 442
NOSTORAGE (-1000) 442
OUTOFMEMORY (-1011) 442
READ (-1002) 442
SEND_FAILED (-1012) 442
UNEXPECTED (-1016) 442
UNKNOWNSYSTEM (-1004) 442
WRITE (-1003) 442
WRITE_FAILED (-1015) 442
LL_HTML 98
LL_INFO_...
METER 311
PRINTJOBSUPERVISION 312
LL_NTFY_...
DESIGNERPRINTJOB 313
EXPRERROR 315
EXPRERROR_EX 315
FAILSFILTER 316
VIEWERBTNCLICKED 316
VIEWERDRILLDOWN 317
LL_NUMERIC 96
LL_NUMERIC_...
INTEGER 96
LOCALIZED 96
LL_OPTION_...
ADDVARSTOFIELDS 268
ALLOW_COMBINED_COLLECTING_OF_DATA_FOR_COLLECTIONCONTROLS 268
ALLOW_LLX_EXPORTERS 268
CALC_SUMVARS_ON_PARTIAL_LINES 269
CALCSUMVARSONINVISIBLELINES 268
CALLBACKMASK 269
CALLBACKPARAMETER 269
CODEPAGE 269
COMPRESSRTF 270
COMPRESSSTORAGE 270
CONVERTCRLF 270
DEFAULTDECSFORSTR 270
DEFDEFFONT 270, 283
DEFPRINTERINSTALLED 202
DELAYTABLEHEADER 270
DESIGNEREXPORTPARAMETER 270
DESIGNERPREVIEWPARAMETER 271
DESIGNERPRINT_SINGLETHREADED 271
ERR_ON_FILENOTFOUND 271
ESC_CLOSES_PREVIEW 271
EXPRSEPREPRESENTATIONCODE 271
FONTPRECISION 272
FONTQUALITY 271
FORCE_DEFAULT_PRINTER_IN_PREVIEW 272
FORCEFIRSTGROUPHEADER 272
FORCEFONTCHARSET 272
HELPAVAILABLE 202, 272
IDLEITERATIONCHECK_MAX_ITERATIONS 272
IMMEDIATELASTPAGE 272
INCLUDEFONTDESCENT 273
INCREMENTAL_PREVIEW 273
INTERCHARSPACING 273
LANGUAGE 202
LCID 273, 283, 290
LOCKNEXTCHARREPRESENTATIONCODE 273
MAXRTFVERSION 274
METRIC 274
NOAUTOPROPERTYCORRECTION 274
NOFAXVARS 274
NOFILEVERSIONUPGRADEWARNING 275
NOMAILVARS 275
NONOTABLECHECK 275
NOPARAMETERCHECK 92, 275
NOPRINTERPATHCHECK 275
NOPRINTJOBSUPERVISION 275
NOTIFICATIONMESSAGEHWND 275
NULL_IS_NONDESTRUCTIVE 276

PARTSHARINGFLAGS 276
PHANTOMSPACEREPR...CODE 276
POSTPAINT_TABLESEPARATORS 277
PREVIEW_SCALES_RELATIVE_TO_PHYSICAL_SIZE 277
PRINTERDCCACHE_TIMEOUT_SECONDS277
PRINTERDEVICEOPTIMIZATION 277
PROHIBIT_USERINTERACTION 277
PROHIBITFILTERRELATIONS 269
PROJECTBACKUP 277
PRVRECT_HEIGHT 278
PRVRECT_LEFT 278
PRVRECT_TOP 278
PRVRECT_WIDTH 278
PRVZOOM_HEIGHT 278
PRVZOOM_LEFT 278
PRVZOOM_PERC 278
PRVZOOM_TOP 278
PRVZOOM_WIDTH 278
REALTIME 278
RESETPROJECTSTATE_FORCES_NEW_DC 278
RESETPROJECTSTATE_FORCES_NEW_PRINTJOB 278
RETREPRESENTATIONCODE 278
RIBBON_DEFAULT_ENABLEDSTATE 278
RTFHEIGHTSCALINGPERCENTAGE 279
SCALABLEFONTSONLY 279
SETCREATIONINFO 279
SHOWPREDEFVARS 279
SKETCH_COLORDEPTH 279
SKIPRETURNATENDOFRTF 279
SORTVARIABLES 279
SPACEOPTIMIZATION 280
SUPERVISOR 280
SUPPORTS_PRNOPTSTR_EXPORT 280
TABLE_COLORING 280
TABREPRESENTATIONCODE 280
UNITS 280
USE_JPEG_OPTIMIZATION 281
USEBARCODESIZES 281
USECHARTFIELDS 281
USEHOSTPRINTER 281
VARLISTDISPLAY 281
VARSCASESENSITIVE 282
XLATVARNAMES 282
LL_OPTIONSTR_...
CARD_PRJDESCR 286
CARD_PRJDESCR_SINGULAR 286
CARD_PRJEXT 283
CARD_PRNEXT 283
CARD_PRVEXT 283
CURRENCY 283
DECIMAL 283
DEFDEFFONT 270, 283
EMBEDDED_EXPORTS 284
EXPORTEDFILELIST 285
EXPORTS_ALLOWED 284
EXPORTS_ALLOWED_IN_PREVIEW 284
EXPORTS_AVAILABLE 284
FAX 286
GTC_PRJDESCR 286
GTC_PRJDESCR_SINGULAR 286
HELPFILENAME 286
IDX_PRJDESCR 286
IDX_PRJDESCR_SINGULAR 286
LABEL_PRJDESCR 286
LABEL_PRJDESCR_SINGULAR 286
LABEL_PRJEXT 287
LABEL_PRNEXT 287
LABEL_PRVEXT 287
LICENSINGINFO 287
LIST_PRJDESCR 286
LIST_PRJDESCR_SINGULAR 286
LIST_PRJEXT 287
LIST_PRNEXT 287
LIST_PRVEXT 287
LLFILEDESCR 287
LLXPATHLIST 268, 284, 287
MAILTO 288
MAILTO_BCC 288
MAILTO_CC 288
MAILTO_SUBJECT 288
NULLVALUE 288
PREVIEWFILENAME 289
PRINTERALIASLIST 289
PROJECTPASSWORD 289

SAVEAS_PATH 290
SHORTDATEFORMAT 290
THOUSAND 290
TOC_PRJDESCR 286
TOC_PRJDESCR_SINGULAR 286
VARALIAS 290
LL_PRINT_...
FILE 102
NORMAL 102
PREVIEW 102
USERSELECT 103
LL_PRNOPT_...
COPIES 243
COPIES_SUPPORTED 233
DEFPRINTERINSTALLED 233
FIRSTPAGE 243
JOBID 234
JOBPAGES 243
LASTPAGE 243
OFFSET 244
PAGE 244
PRINTDLG_ALLOW_NUMBER_OF_FIRST_PAGE 244
PRINTDLG_ONLYPRINTERCOPIES 244
PRINTORDER 233
UNIT 233
UNITS 244
USE2PASS 233
LL_PRNOPTSTR_...
EXPORT 245
ISSUERANGES 245
PAGERANGES 245
PRINTDST_FILENAME 245
PRINTJOBNAME 245
LL_PROJECT_...
CARD 103
LABEL 103
LIST 104
LL_QUERY_...
DESIGNERACTIONSTATE 319
EXPR2HOSTEXPRESSION 319
LL_RTF 98
LL_TEXT 95
LL_WRN_...
PRINTFINISHED (-997) 441
REPEAT_DATA (-998) 442
TABLECHANGE (-105) 441
LL_WRN_STG...
UNFAXED_PAGES (-1100) 443
LlAssociatePreviewControl 153
LlCreateSketch 154
LlDbAddTable 109, 117, 154
LlDbAddTableEx 155
LlDbAddTableRelation 109, 117, 156
LlDbAddTableRelationEx 157
LlDbAddTableSortOrder 109, 117, 159
LlDbAddTableSortOrderEx 159
LlDbSetMasterTable 160
LlDebugOutput 161
LlDefineChartFieldExt 162
LlDefineChartFieldStart 189, 237
LlDefineField 103, 163
LlDefineFieldExt 164
LlDefineFieldExtHandle 165, 188
LlDefineFieldStart 167, 188, 189, 191, 238
LlDefineLayout 101, 129, 167, 276, 439
LlDefineSumVariable 169
LlDefineVariable 103, 169
LlDefineVariableExt 171
LlDefineVariableExtHandle 172, 188
LlDefineVariableStart 173, 188, 190, 191
LlDesignerAddAction 174
LlDesignerFileOpen 175
LlDesignerFileSave 177
LlDesignerGetOptionString 177
LlDesignerInvokeAction 178
LlDesignerProhibitAction 101, 178
LlDesignerProhibitEditingObject 180
LlDesignerProhibitFunction 180
LlDesignerRefreshWorkspace 181
LlDesignerSetOptionString 178, 181
LlDlgEditLineEx 182
LlDomCreateSubobject 146, 183
LlDomDeleteSubobject 147, 184
LlDomGetObject 145, 184
LlDomGetProject 185
LlDomGetProperty 148, 185

LIDomGetSubobject 146, 186
LIDomGetSubobjectCount 146, 186
LIDomSetProperty 147, 187
LIEnum...
GetEntry 187
GetFirstChartField 188
GetFirstField 189
GetFirstVar 190
GetNextEntry 190
LIExportOption 24
LIExprError 191, 315
LIExprEvaluate 191, 290, 439
LIExprFree 193
LIExprGetUsedVars 193
LIExprGetUsedVarsEx 194
LIExprParse 194
LIExprType 196
LIGetChartFieldContents 196
LIGetDefaultPrinter 197
LIGetDefaultProjectParameter 198, 424
LIGetErrortext 198
LIGetFieldContents 199
LIGetFieldType 200, 208
LIGetLastErrorText 200
LIGetNotificationMessage 131, 201
LIGetOption 201
LIGetOptionString 202, 288
LIGetPrinterFromPrinterFile 203
LIGetProjectParameter 204, 424
LIGetSumVariableContents 205
LIGetUsedIdentifiers 167, 206, 237
LIGetUsedIdentifiersEx 206
LIGetUserVariableContents 207
LIGetVariableContents 208
LIGetVariableType 208, 209
LIGetVersion 209
LIJobClose 210, 211
LIJobOpen 94, 100, 202, 210, 268
LIJobOpenLCID 94, 100, 202, 212
LIJobStateRestore 212
LIJobStateSave 213
LILocAddDesignLCID 214
LILocAddDictionaryEntry 214
LIPreviewDeleteFiles 216
LIPreviewDisplay 216, 439
LIPreviewDisplayEx 217
LIPreviewSetTempPath 218
LIPrint 219, 248
LIPrint[WithBox]Start 167, 173
LIPrintAbort 219
LIPrintCopyPrinterConfiguration 220
LIPrintDbGetCurrentRelation 114
LIPrintDbGetCurrentTable 111, 222
LIPrintDbGetCurrentTableFilter 222
LIPrintDbGetCurrentTableRelation 223
LIPrintDbGetCurrentTableSortOrder 113, 224
LIPrintDbGetRootTableCount 221
LIPrintDeclareChartRow 224
LIPrintDidMatchFilter 225
LIPrintEnableObject 439
LIPrintEnd 227, 285
LIPrinterSetup 227
LIPrintFields 228, 248, 270
LIPrintFieldsEnd 229
LIPrintGetChartObjectCount 230
LIPrintGetCurrentPage 231
LIPrintGetFilterExpression 231
LIPrintGetItemsPerPage 232
LIPrintGetOption 231, 232
LIPrintGetOptionString 234
LIPrintGetPrinterInfo 234
LIPrintGetProjectParameter 235, 424
LIPrintGetTextCharsPrinted 439
LIPrintIsChartFieldUsed 236
LIPrintIsFieldUsed 167, 237, 439
LIPrintIsVariableUsed 173, 238, 439
LIPrintOptionsDialog 228, 239, 242
LIPrintOptionsDialogTitle 239
LIPrintResetProjectState 240
LIPrintSelectOffsetEx 241
LIPrintSetBoxText 242, 248, 438
LIPrintSetOption 243
LIPrintSetOptionString 244
LIPrintSetProjectParameter 246, 424
LIPrintStart 246, 440
LIPrintUpdateBox 247

LIPrintVariableStart 239
LIPrintWillMatchFilter 248
LIPrintWithBoxStart 249
LIProjectClose 251
LIProjectOpen 251
LIProjectSave 253
LIRTFCopyToClipboard 254
LIRTFCreateObject 254
LIRTFDeleteObject 254
LIRTFDisplay 255
LIRTFEditObject 256
LIRTFEditorInvokeAction 257
LIRTFEditorProhibitAction 258
LIRTFGetText 258
LIRTFGetTextLength 259
LIRTFSetText 260
LISelectFileDlgTitleEx 106, 260
LISetDebug 92, 261
LISetDefaultProjectParameter 263, 424
LISetFileExtensions 263
LISetNotificationCallback 129, 264
LISetNotificationCallbackExt 265
LISetNotificationMessage 131, 267
LISetOption 103, 106, 202, 267, 439, 440
LISetOptionString 106, 203, 264, 282
LISetPrinterDefaultsDir 291
LISetPrinterInPrinterFile 291
LISetPrinterToDefault 293
LLStaticTable 109, 113
LIStgsys...
Append 323, 342
Convert 323
DeleteFiles 325
DestroyMetafile 325, 332
DrawPage 326
GetAPIVersion 327
GetFilename 327
GetFileVersion 328
GetJobCount 329
GetJobOptionStringEx 329
GetJobOptionValue 330
GetLastError 331
GetPageCount 331
GetPageMetafile 331, 332
GetPageOptionString 333
GetPageOptionValue 332, 334
GetPagePrinter 336
Print 337
SetJob 338
SetJobOptionStringEx 339
SetPageOptionString 334, 340, 342
SetUILanguage 340
StorageClose 341
StorageConvert 341
StorageOpen 342
StoragePrint 343
LLVIEW??.EXE 445
LIViewerProhibitAction 294
LIXGetParameter 294
LIXSetParameter 295
LoadFinished 75
Localization 290
Lock functions 37
Lock menu items 37
Lock objects 37
Locked objects 280
LS_OPTION_...
BOXTYPE 330
COPIES 335
CREATION 334
CREATIONAPP 334
CREATIONDLL 334
CREATIONUSER 334
ISSUEINDEX 336
JOBNAME 334
PAGENUMBER 335
PHYSPAGE 335
PRINTERALIASLIST 334
PRINTERCOUNT 331
PRN_INDEX 336
PRN_ORIENTATION 335
PRN_PIXELS_X 335
PRN_PIXELS_Y 335
PRN_PIXELSOFFSET_X 335
PRN_PIXELSOFFSET_Y 335
PRN_PIXELSPERINCH_X 335

PRN_PIXELSPERINCH_Y 335
PRN_PIXELSPHYSICAL_X 335
PRN_PIXELSPHYSICAL_Y 335
PROJECTNAME 333
UNITS 330
USED_PRTDEVICE 334
USER 334
LsMailConfigurationDialog 344, 417
LsMailGetOptionString 345
LsMailJobClose 346
LsMailJobOpen 346
LsMailSendFile 347
LsMailSetOptionString 348
LsSetDebug 349

M

MAPI 416
Menu 198, 294
Menu ID 301
Menu items 178
MENUID.TXT 76, 179, 257, 258, 301
Messages 129
Meter dialog 242
MHTML export 386
Multi Mime HTML 386
Multiple print 32
Multiple tables 226
Multiple tables 109, 116
Multithreading 429
MySQL 61

N

NuGet 19
NULL-values 423
Numerical Variables 96

O

Object model 40
ObjectDataProvider 28
ObjectReportContainer 41
Objects 38, 41
Barcode 39
HTML 39
Picture 39
RTF-Text 39
Text 38
ObjectText 41
OCX component
data transfer 69
Designer functions 72
Designer objects 74
Events 70
Integration 68
Language selection 70
Preview control 71
Preview files 71
print-and-design methods 68
OCX Event
'BtnPress' 77
'LoadFinished' 78
'PageChanged' 78
OCX Method
'GetOptionString' 77
'GotoLast' 77
'GotoNext' 77
'GotoPrev' 76
'PrintAllPages' 77
'PrintCurrentPage' 77
'PrintPage' 77
'RefreshToolbar' 77
'SaveAs' 77
'SendTo' 77
'SetOptionString' 77
'SetZoom' 77
'ZoomReset' 77
'ZoomRevert' 77
'ZoomTimes2' 77
OCX Properties
'AsyncDownload' 75
'BackColor' 75
'CanClose' 76
'CurrentPage' 75
'Enabled' 75
'FileURL' 75

'Pages' 75
'SaveAsFilePath' 76
'ShowThumbnails' 76
'ToolbarButtons' 76
'ToolbarEnabled' 75
'Version' 76
OCX Viewer Control 74
OleDbConnectionDataProvider 30
Oracle 61
OracleConnection 30
OracleConnectionDataProvider 30

P

Page break 107, 219
Page number 244
Parameter92, 178, 183, 186, 188, 191, 192, 193, 194, 197, 198, 199, 200, 203, 204, 205, 206, 207, 208, 209, 224, 232, 234, 235, 236, 258, 261, 295, 439
Pass additional data 34
PDF export 354
P-file 22
Picture 33
Picture export 396
Placeholders 56
Portrait 36
PostgreSQL 61
PowerPoint Export 370
PPTX export 370
Preview 12, 64, 216
Preview API 322
Preview files 322
Convert 64
Join 64
PreviewFile 64
Print 22
Network 66
Print Engine 12
Print files 322
Print options 240, 243
Print options dialog 25
Printer configuration 220, 446
Printer device 246
Printer selection window 227
Printer settings 22
Printers 36
Printing
Job 93, 94
Proceeding 103
Progress 15
ProhibitedActions 37
ProhibitedFunctions 37
Project files
In database 66
Project includes 56
Project parameters 424
Project type 25
Project types 12, 35
Cards 36
Labels 35
Lists 35
ProjectCard 41
ProjectLabel 41
ProjectList 41
Property 'Pages' 78

R

ReadOnlyObjects 37
Real data preview 216
Regions 36, 41
Report Container 40, 108, 109, 116
Report Parameter 140
Return Value 93
Rounding 423
RTF 98
RTF control 270, 274
RTF Editor, calling 254
RTF export 373

S

Scripting 434, 435
Sending e-Mail 65
Settings 448
SMTP 416
Space Optimization 280

Speed Optimization 108
SqlConnection 30
SqlConnectionDataProvider 30
SQLite 61
START-Event 138
Sub reports 57, 109, 116
SubItemTable 41
Sum variable 169, 205, 268, 423
Suppress data 59
SVG export 399
System requirements 9, 445

T

Text (CSV) export 388
Text (Layout) export 386, 390
Text blocks 31
Text Variables 95
Thousands separator 290
Threading 429
Translate$ 215
Translation 290
TTY 401

U

Unbound data 58
Update 452
User Drawn Object 99
User information 279
User variable 208

V

VariableHelpText 34
Variables 13, 25, 31, 35, 166
VBScript 430
VCL 14
VCL component
  Data binding 80
  Data transfer 84
  Designer objects 88
  Events 85
  Language selection 85
  Preview control 85
  Preview files 85
  Print-and-design methods 83
  Relational links 81
VDF 15
Viewer Application 445
Viewer OCX Control 74
Visual Basic 14
Visual C++ 14
Visual DataFlex 15
Visual dBase 15
Visual FoxPro 15
Visual Studio 17
VLC component
  Integration 80

W

Web Reporting 47, 48, 428
Windows Scripting Host 430
Word export 365

X

Xbase++ 15
XHTML export 379
XLS export 358
XMAPI 416
XML 31
XML export 392
XmlDataProvider 31
XPS export 378, 403

Z

ZIP export 421
Zoom factor 278