List & Label 12

Programmer's Reference

1.Introduction

Congratulations on your purchase of List & Label. 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. Overview and Basic Concepts you should know about

1.1.1. Database Independent Concept

List & Label works database independent, i.e. List & Label itself does not access the database and does not have own database drivers. This offers you an enormous amount of advantages, but - you get nothing for free – this does come with some minor disadvantages.

The advantages:

- No unnecessary overhead by duplicate database drivers, thereby less memory consumption so that your application may execute faster.
- Less risk of problems with additional database drivers.
- More flexible, as this allows controlling the data in one place only.
- Can be used without database at all.
- 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 a program. List & Label needs to be fed with data. This is very simple and needs little coding in standard cases. This is not valid for the .NET, the VCL control as well as the VB OCX. These can be used as data bound control giving you easy access to standard databases.

1.1.2. Basic Principles

As far as the technical side is concerned List & Label consists of different components. A detailed list of available modules can be found in the file "redist.txt" of your List & Label installation.

To allow easy incorporation of List & Label into your projects, several controls and applications are shipped:

- A standard OCX control for development environments capable of inserting OCX controls
- A data bound VB-Control

- A VCL-Control to be used in Delphi
- A .NET assembly for usage with the .NET languages

And for the preview files:

- An application to view the preview files.
- An OCX/ActiveX control to view and print preview files.

The main part, the engine, has three functional parts:

- An interactive WYSIWYG designer that can also be used by your customers without having to pay any additional license fee.
- The print engine to print a project.
- The preview engine which allows you to view and print the preview data.

The Designer

The designer is the program component with which the layout for the three basic types of projects can be defined: reports / lists, labels and file cards.

This designer can be used by the end user so that the end user can define individual projects or adapt the projects offered by you to personal requirements.

Single objects of a project can be locked so that the user does not have access to those objects (for example, so that the user cannot delete a logo which you always want on the print-out). Additionally, the usage of the layout dialog can be limited by hiding menu items and consequently making these functions inaccessible to the user.

As List & Label works database independent, the Designer cannot "know" which fields and variables should be available to your users without being integrated into an application. Thus, your application needs to pass all variables and fields before calling the Designer.

The Print Module

This module has the task to print (the data passed by you) to a printer or preview file or exporting it to PDF, HTML, RTF, TIFF, JPEG, Plaintext, bitmap and others.

Optionally, a dialog is offered in which the user can determine the print configurations required, e.g. output medium, number of pages, etc.

All important features, e.g. an open file dialog or an abort window with a meter bar as well as partial printing are supported.

The Viewer Control

This control can be used to view List & Label preview files, print them, store them as preview or PDF file, send them as email, and more. It can be inserted into your own projects (assuming you have a programming environment that can insert OCX controls, or you use .NET) or added to a HTML Internet/Intranet page.

1.1.3. Adaptable & Intelligent Dialog Styles

List & Label uses a special module which offers an adaptation of the dialog styles. In this manner, bitmap-buttons can be switched on and off and the display of dialogs can be changed from windows-standard to various different three-dimensional styles.

The dialogs and windows automatically remember their position.

1.1.4. Available Languages

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

1.1.5. Program Dependent Configurations

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 manner, your application does not need to pay attention to whatever configurations are used in other applications. When a foreign application requires for example English 3D dialogs, you can choose German 2D dialogs without having to take anything else into consideration.

1.1.6. 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 DEBWIN2.EXE which is included in the package. This application can also be used to log your own debugging output.

1.2. Modules Required for the Program

Please refer to the file REDIST.TXT that has been installed in your List & Label installation directory for further information.

1.2.1. System Requirements

Windows Version 98 SE or newer or NT 4 or newer (NT4, Windows 2000, XP or 2003).

Some functions can only be used in recent operating systems. You'll find such restrictions documented in the corresponding chapters.

Hint: 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.

1.3. Important Annotations

- Examples are in C-syntax, as far as DLL-calls are mentioned. It should be simple to translate them. For this reason the separate chapter "Examples" is in Basic-Syntax.
- Texts enclosed in '<' and '>' should naturally be translated into the corresponding programming language.
- When using path strings in your C or C++ programs please remember to use '\\' instead of '\'!
- Separate online help files are available for the components (OCX, VCL, .NET) and the Designer
- If in doubt, it often helps to use the debugging facilities! See *LlSetDebug()/LsSetDebug()* and *LlDebugOutput()* for further information.

Remember that

- Buffer sizes passed to List & Label are bytes (SBCS/DBCS) or WCHARs (Unicode) – if you use C, replace "sizeof(buf)" by "sizeof(buf)/sizeof(TCHAR)" as calculation which works for all cases.
- If you use the "A" API of the Unicode DLL or the "W" API of the SBCS DLL, the string parameters are converted using the *CP_ACP* codepage. You can use *LL_OPTION_CODEPAGE* to change this.
- Each List & Label DLL writes the project file in its native format: the Unicode DLL writes the project file in Unicode format. Please keep that fact in mind if you ever want to look at or modify the project files. Anyway, each DLL can still read the "other" format. For conversion the codepage mentioned above will be used.
- The printer definition files (P-Files) and the Sketches (V-Files) are compatible.
- In MBCS environments the "new expression mode" should only be used.

2. Introduction to Programming

This chapter is meant as a fast introduction into the development with List & Label. Here you can find some easy examples that will allow you a fast success using the component. The demonstrated concepts will be more thoroughly explained in the next chapter.

Another good start is to work with one of the examples that we supply for various programming languages. You will find those in the start menu of your List & Label installation.

2.1. How Should I Integrate List & Label?

List & Label can be used from a wide range of programming languages. With most of the languages, there are several more integration types. You can work with databinding, event based reporting or a complete custom print loop. This chapter will aid you in choosing the integration type that suits best.

2.1.1. Databinding

This option is available fort he VCL (Delphi, C++ Builder, Borland Developer Studio) and the .NET component (Visual Studio .NET, Delphi.NET, C#-Builder, Borland Developer Studio etc.). Most reporting requirements can be covered with databinding. We recommend using it wherever possible.
It's also the easiest way to access the report container object. The List & Label relevant code of your application will typically only be a couple of code lines.
You'll find samples for this integration type in the "Programmable Samples" subfolder of your List & Label installation.

2.1.2. Event Based Reporting

This option is available for the OCX, VCL and .NET components. It is described in detail in chapter "2.5. The Print and Design". Passing data is done within an event handler. You can use this method whenever you report unbound data (i.e. data that doesn't come from a database and is not available in a data control), e.g. class arrays or similar data. The report container object is not supported in this mode, i.e. you can not have relational data within the designer.
You'll find samples for this integration type in the "Programmable Samples" subfolder of your List & Label installation.

2.1.3. Custom Print Loop

This option is available for all supported languages. It is the most powerful yet most complex integration type. You have full control of the print loop in this case. You should use it whenever none of the other types is an option or if you cannot solve your requirements with one of the other types (very complex requirements, special layouts etc.).
See chapter "2.6. The Print Loop" for a detailed description. If you want to work with the report container object in order to have multiple tables, charts and crosstabs see chapter "4. Working with the Report Container".

You'll find samples for this integration type in the "Programmable Samples" subfolder of your List & Label installation.

2.2. Project Types

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

2.2.1. Labels and File Cards

Label and file card projects in principle differ only in their initial workspace (design)size and file extension. This difference was introduced in order to enable the file open dialog to distinguish these logically different types.

Label projects consist of an arrangement of objects, which are printed once or multiple times (in columns or rows) per page. The objects can hold variables, which take on different values during printing, depending on the data.

2.2.2. Lists

Lists on the other hand consist of objects with variables, which are printed once per page and one or more list objects, in which the fields with contents varying with the data records are printed. The table/report container object is only available in this mode.

2.3. 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 *LISetFileExtensions()* or *LISetOptionString()*.

The crucial file is the form definition file; the "V-File" can be created at any time using *LICreateSketch()*. If the "P-File" can not 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 or generated can be specified with *LISetPrinterDefaultsDir()*, which may be relevant for network applications.

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 *LIPreviewSetTempPath()*.

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 (ex. "~lst" for the default list project extension).

If the target does not support long filenames, the extension will be cut off after the third character (ex. "~ls").

2.4. Programming Basics

2.4.1. The List & Label Job

To enable List & Label to distinguish between the different applications which 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 (*LIJobOpen()*,*LIJobOpenLCID()*) 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.

2.4.2. Variables and Fields in List & Label

List & Label distinguishes between two kinds of data fields: on the one hand there are data fields, which 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 which 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 variables as well as fields can occur. For the print of an invoice an application typically would declare the invoice header data such as customer name and address as variables, while the item data as article number, price per piece, description etc. would be passed as fields.

The delivery and definition of variables and their contents is performed with the List & Label function *LIDefineVariable(Ext)()*, the delivery and definition of fields and their contents is performed with the List & Label function *LIDefineField(Ext)()*.

You can build hierarchies in the Designer by using the dot "." as a separator. With it you can use e.g. "person.address.street" and "person.address.city" as variables or fields. In the Designer you get a hierarchic structure i.e. below of "person" you will find a folder "address" with the fields "city" and "street".

2.4.3. Treatment of 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 shipping which has not yet happened. 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.

2.4.4. Variable and Field Types

List & Label allows the specification of the following variable- and field-types:

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*: become 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 (like in "1,420,000.00") is not allowed.

Constant:

LL_NUMERIC

Content e.g.:

""1,255.00""

Hint:

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

Date

Constant:

LL_DATE

Content e.g.:

"2451158.5" (equals 12/11/1998 noon) or "5-1-2000" for *LL_DATE_MDY* format or "20000501" 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 analogously to the Julian date, however a different starting 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 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 string to List & Label, there is no necessity 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 (-) by the first three formats.

Constant:

LL_DATE_LOCALIZED

Hint:

The date is being interpreted as localized date, ex. "12/31/2001". Internally the OLE-API VarDateFromStr() is used.

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.

HTML Formatted Text

Constant:

LL_HTML

Content e.g.:

"<html><body><font size=7>Hello World</body></html>"

Hint:

Due to license restrictions GIF images are not supported.

Drawing

Constant:

LL_DRAWING

Content e.g.:

"c:\temp\sunny.wmf"

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 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:

LL_BARCODE_EAN13, LL_BARCODE_EAN8, LL_BARCODE_UPCA, LL_BARCODE_UPCE, LL_BARCODE_3OF9, LL_BARCODE_25INDUSTRIAL, LL_BARCODE_25INTERLEAVED, LL_BARCODE_25DATALOGIC, LL_BARCODE_25MATRIX, LL_BARCODE_POSTNET , LL_BARCODE_FIM, LL_BARCODE_CODABAR , LL_BARCODE_EAN128, LL_BARCODE_CODE128, LL_BARCODE_DP_LEITCODE, LL_BARCODE_DP_IDENTCODE, L_BARCODE_GERMAN_PARCEL, LL_BARCODE_CODE93, LL_BARCODE_MSI, LL_BARCODE_CODE11, LL_BARCODE_MSI_10_CD, LL_BARCODE_MSI_10_10, LL_BARCODE_MSI_11_10, LL_BARCODE_MSI_PLAIN, LL_BARCODE_PDF417, LL_BARCODE_MAXICODE, LL_BARCODE_DATAMATRIX, LL_BARCODE_AZTEC, LL_BARCODE_MAXICODE_UPS, LL_BARCODE_EAN14, LL_BARCODE_UCC14, LL_BARCODE_SSCC

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

2.5. The Print and Design Methods

These two methods represent – apart from databinding - the simplest way to program List & Label. The methods are only contained in the OCX control, the VCL and the .NET-component. An exact description of the parameters can be found in the respective online help files.

A more detailed description for the .NET component and its special handling can be found in the chapter "Working with the .NET component".

Use these methods to program a simple label or list print. For more complex printing tasks (e.g. invoices) we recommend to implement the printing loop on your own. Examples for the most popular IDEs are installed by the setup.

The components offer an event CmdSetPrintOptions (OCX) or OnSetPrintOptions (VCL) in which you can set print options using LlPrintSetOption() and LlPrintSetOptionString().

With the help of these options, you can even program a "background" export. More detailed information on this can be found in the chapter "Export without user interaction". A description of the events themselves can be found in the online help of the components.

The .NET component offers a property "ExportOptions" to set the options. An example is contained in the package.

If you use these methods, you have to support the events CmndDefineVariables, CmndDefineFields (OCX) or OnDefineVariables, OnDefineFields (VCL). The actual declaration of the data to List & Label is done within these events. The usual functions LlDefineField, LlDefineVariable() etc. are used for this task (also in the case of the self implemented print loop).

The following VB example demonstrates the simplicity of these methods:

```
'=========================================================================
Private Sub ButtonPrint_Click()
   'Move to first record
   Form1.Data1.Recordset.MoveFirst
   'Print the project "simple.lbl"
   ret% = ListLabel1.Print(0, LL_PROJECT_LABEL, _"simple.lbl", 1,
LL_PRINT_USERSELECT, _
      LL_BOXTYPE_NORMALWAIT, hWnd, "Print labels ", True,_Environ$("temp"))
End Sub
'=========================================================================
Private Sub ButtonDesign_Click()
   ' Move to first record
   Form1.Data1.Recordset.MoveFirst
   'Start the designer
   ListLabel1.Design(0, hWnd, "Design", LL_PROJECT_LABEL, "simple.lbl", 1)
End Sub
'=========================================================================
Private Sub ListLabel1_CmndDefineVariables(ByVal nUserData As Long, _
      ByVal bDummy As Long, pnProgressInPerc As Long, pbLastRecord As Long)
   'This event is triggered by the List & Label commands Print and Design
and is
   ' called for each record to define the variables and their contents to
List &
   ' Label.
   Dim i As Integer
   'Loop through all fields in the data base
   For i = 0 To Form1.Data1.Recordset.Fields.Count - 1
       'Convert the database field types to List & Label types, see function
          'VarType() in the VB help
       Select Case VarType(Form1.Data1.Recordset.Fields(i))
          '--- numerical variable
          Case 2, 3, 4, 5
             para = LL_NUMERIC
             content$ =  Form1.Data1.Recordset.Fields(i)
             'If data type is "date", convert to a numerical date value
          Case 7
```

```
            para = LL_DATE_MS
            a! = CDate(Form1.Data1.Recordset.Fields(i))
            content$ = a!
            '--- Logical variable (Yes/No), Boolean
         Case 11
            para = LL_BOOLEAN
            content$ = Form1.Data1.Recordset.Fields(i)
            '--- Text variable
         Case Else
            para = LL_TEXT
            content$ = Form1.Data1.Recordset.Fields(i)
      End Select

      nRet = Form1.ListLabel1.LlDefineVariableExt(_
         Form1.Data1.Recordset.Fields(i).Name, content$, para)
   Next i

   ' Is real data needed? (not for designer call)
   If bDummy = 0 Then
      'yes: set progress bar value, ...
      pnProgressInPerc =  Form1.Data1.Recordset.PercentPosition

      'and move to next record
      Form1.Data1.Recordset.MoveNext

      'if last record is reached, set flag for print end
      If Form1.Data1.Recordset.EOF = True Then
         pbLastRecord = 1
      End If
   End If
End Sub
```

2.6. The Print Loop

2.6.1. Supplying Data

In this mode, List & Label works also database independent. This means that the application (i.e. you as a programmer) are responsible for supplying the data values. 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 hold. 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 into 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. There is no real data preview in the designer (because of the data base independence), you may however pass a sample content for each field, which is used during the design in the (design-)preview.

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

2.6.2. 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_USERSELECT*) by giving him the option to choose the print target within the print-dialog called by *LlPrintOptionsDialog()*. You then only have to query if the user selected the preview print and display the preview (using *LlPreviewDisplay()*) in this case, or if he has chosen some of the other export formats.

2.6.3. Export Modules

To use the additional export modules, you can not use the LL_PRINT_xxx flags as mentioned above. If you want to support export destinations other than printer or preview, you need to set them using *LlPrintSetOptionString()* with *LL_OPTIONSTR_EXPORTS_ALLOWED*. For further information, please read the chapter on the export modules.

2.6.4. Basic Proceeding

First a List & Label job is opened (*LlJobOpen()* or *LlJobOpenLCID()*) and, if needed, global List & Label options are set (*LlSetOption()*). Now List & Label has to be informed that the printing should start (*LlPrintWithBoxStart()*). With this call the label or form definition file to be used for printing is passed to List & Label. At this point, the definition file is opened and parsed by List & Label. During this, a syntax check of all used variables, fields and formulas is performed. This requires List & Label to know all variables and fields you are making available to your end users. For this reason, you have to 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).

After the optional display of a print options dialog (*LlPrintOptionsDialog()*) the actual print loop follows.

2.6.5. Label- and File Card Printing

The following simple loop demonstrates the printing of labels while iterating through a (fictitious) data base. The user is left with the choice whether he wants to print to real data preview, printer or file:

```
'Declare the variables for the syntax check (with sample content)
ListLabel1.LlDefineVariableExt("ItemNo", "0815", LL_TEXT)
ListLabel1.LlDefineVariableExt("Description", "sample", LL_TEXT)
```

```
ListLabel1.LlDefineVariableExt("Price", "123", LL_NUMERIC)
ListLabel1.LlDefineVariableExt("Deliverable", "T", LL_BOOLEAN)
ListLabel1.LlPrintWithBoxStart(LL_PROJECT_LABEL, "item.lbl",_
  LL_PRINT_USERSELECT, LL_BOXTYPE_NORMALWAIT, hWnd,_
  "Printing Article Label")

'Show print options dialog:
ListLabel1.LlPrintOptionsDialog(hWnd, "")

'Save chosen printing target
nDest = ListLabel1.LlPrintGetOption( LL_PRNOPT_PRINTDLG_DEST)
Data1.Recordset.MoveFirst()
While (Not Form1.Data1.Recordset.EOF) and nLLError = 0
   ListLabel1.LlPrintSetBoxText("Printing...",_
      Form1.Data1.Recordset.PercentPosition)
   'declare the variables and pass the real data content to List & Label:
   contents$ = Form1.Data1.Recordset.Fields(1)
   ListLabel1.LlDefineVariableExt("ItemNo", contents$, LL_TEXT)
   contents$ = Form1.Data1.Recordset.Fields(2)
   ListLabel1.LlDefineVariableExt("Description", contents$, LL_TEXT)
   contents$ = Form1.Data1.Recordset.Fields(3)
   ListLabel1.LlDefineVariableExt("Price", contents$, LL_NUMERIC)
   contents$ = Form1.Data1.Recordset.Fields(4)
   ListLabel1.LlDefineVariableExt("Deliverable", contents$, LL_BOOLEAN)
   'Now, print the label:
   nLLError = ListLabel1.LlPrint()
   'LL_WRN_REPEAT_DATA can be ignored for labels:
   if nLLError = LL_WRN_REPEAT_DATA then nLLError = 0
   Data1.Recordset.MoveNext()
Wend
ListLabel1.LlPrintEnd(0)

'Did user choose preview print?
if (nLLError = 0) and (nDest = LL_DESTINATION_PRV) then
   'Show preview and delete preview file (.LL)
   ListLabel1.LlPreviewDisplay("item.lbl", "", hWnd)
   ListLabel1.LlPreviewDeleteFiles("item.lbl", "")
end if
```

2.6.6. List Printing

The following example code (without any error handling) for the print of an invoice ("invoice.lst") to real data preview has encapsulated the definition of variables and fields into two sub routines (DefineVars and DefineFields). It uses two data tables, Data1 for the invoice header data as 'InvoiceNo', Customer' etc. which are defined as variables, and Data2 for the respective item data such as 'ItemNo', 'Description' etc. which are defined as fields.

```
'Define variables and fields with sample contents for syntax
'check
DefineVars
DefineFields
```

```
'Start printing
ListLabel1.LlPrintWithBoxStart(LL_PROJECT_LIST, "invoice.lst",_
   LL_PRINT_PREVIEW, LL_BOXTYPE_NORMALWAIT, hWnd, "Printing")
'Show print options dialog:
ListLabel1.LlPrintOptionsDialog(Form1.hWnd, "")

'Move to first item record
Data2.Recordset.MoveFirst
'Fill variables with real data content
DefineVars
'Print variables
ListLabel1.LlPrint
'While the end of the items table is not reached...
While Not Data2.Recordset.EOF
   'Set percentage value in progress bar
   ListLabel1.LlPrintSetBoxText("Printing invoice",_
     Data2.Recordset.PercentPosition)
   'If invoice no. matches
   If Data2.Recordset.Fields(0) = Data1.Recordset.Fields(0) Then
      'Fill fields with real data from the current items record
      DefineFields
      'Print fields, if page break occurred, repeat
      While ListLabel1.LlPrintFields = LL_WRN_REPEAT_DATA
         ListLabel1.LlPrint
      Wend
   End If
   'Move to next item record
   Data2.Recordset.MoveNext
Wend
'Print footer lines
While ListLabel1.LlPrintFieldsEnd = LL_WRN_REPEAT_DATA
Wend

'End printing
ListLabel1.LlPrintEnd(0)
'As we chose printing to preview, show preview and delete preview file (.LL)
ListLabel1.LlPreviewDisplay(buf$, "", hWnd)
ListLabel1.LlPreviewDeleteFiles(buf$, "")
```

For clarity reasons, the DefineVars and DefineFields subroutines are not shown. You can imagine the routines to be analogous to the *LlDefineVariable(Ext)()* calls in the label print example. In the DefineVars-function the data is read from Data1 (the header data) and defined using the *LlDefineVariable[Ext]()* function. In the DefineFields-function the data is read from Data2 (the item data) and defined using the *LlDefineField[Ext]()* function.

3. Further Programming Basics

3.1. Invoking the designer

3.1.1. 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)
<define dialog-settings>*
   (LlSetDlgboxMode)
<which file?>*
   (LlSelectFileDlgTitleEx)
<define variables>
   (LlDefineVariableStart,
    LlDefineVariable,
    LlDefineVariableExt,
    LlDefineVariableExtHandle)
<define fields>*                ; only LL_PROJECT_LIST
   (LlDefineFieldStart,
    LlDefineField,
    LlDefineFieldExt,
    LlDefineFieldExtHandle)
<define Sorting>*
   (LlDefineSortOrderStart,
    LlDefineSortOrder)
<disable functions>*
   (LlDesignerProhibitAction,
    LlDesignerProhibitFunction)
<call designer>
   (LlDefineLayout)
<close Job>
   (LlJobClose)
```

It is sufficient for job management (non-marked parts) 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()* (e.g. dialog design, debugging or expression mode) and making local settings like disabling menu items directly before calling the designer or printing.

3.1.2. Annotations

If setting of certain options is required then these must of course be undertaken before calling the designer. For a special 3D dialog mode, for example

```
LlSetDlgboxMode(LL_DLGBOXMODE_ALT1);
```

can be called, or the corresponding property in the OCX or VCL can be set.

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's important that 'buf' is pre-initialized - either to an empty string (""), or to a file name suggestion (which also sets the path!):

```
strcpy(buf,"c:\\mylabel.lbl");
LlSelectFileDlgTitle(hJob,hWindow,"Choose label", LL_PROJECT_LABEL,
buf,
   sizeof(buf), NULL);
```

When using the OCX control, no buffer is necessary:

```
sFilename = "c:\mylabel.lbl";
LL1.LlSelectFileDlgTitleEx(me.hWnd,"Choose label",
   LL_PROJECT_LABEL,sFilename,"")
```

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 is not 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 the variable buffer is cleared (in case variables have been previously defined. The call is also recommended to make sure then variable buffer is empty, and no variables from the previous print job are remaining that might possibly 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 on the designer work sheet in the preview

```
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 also has to make the fields needed available. This happens analogous 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 far enough for the user to edit his label or file card by starting the designer:

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

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

Please note that you'll always only see one record in the Designer. This of course is due to the database independency – you have only passed one record before calling the Designer. To see the final result, you need to start a print job. See the following chapter.

It is generally recommended to check for error codes.

3.2. Print Processing

3.2.1. Basic Scheme

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)
<define dialog-settings>*
   (LlSetDlgboxMode)
<print>     ; see below
<close Job>
   (LlJobClose)
```

Most of this is familiar from the previous chapter, and holds also for the print. What is - of course - different is the underlined part:

Printing Labels and File Cards

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

```
<define all necessary variables>
   (LlDefineVariableStart,
    LlDefineVariable,
    LlDefineVariableExt,
    LlDefineVariableExtHandle)
     LlSetPrinterDefaultsDir

<start print>
   (LlPrintStart,
    LlPrintWithBoxStart)
<define options>
   (LlPrintSetOption,
    LlPrintSetOptionString,
    LlPreviewSetTempPath)       ; only Preview
<let user change options>*
   (LlPrintOptionsDialog,
    LlPrintOptionsDialogTitle,
    LlPrintSelectOffset,
    [LlPrinterSetup])
<define constant variables>
   (LlDefineVariable,
    LlDefineVariableExt,
    LlDefineVariableExtHandle,
    LlPrintIsVariableUsed)
<check sortorder>*
   (LlPrintGetSortOrder)
<get printer info for progress-box>*
   (LlPrintGetOption,
    LlPrintGetPrinterInfo)
<skip unwanted labels>*

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

<end print>
   (LlPrintEnd)

<if preview>
   {
   <show preview>*
      (LlPreviewDisplay)
```

```
   <delete preview-files>*
      (LlPreviewDeleteFiles)
   }
```

Printing Lists

And for printing a report (*LL_PROJECT_LIST*):

```
<define all necessary variables>
   (LlDefineVariableStart,
    LlDefineVariable,
    LlDefineVariableExt,
    LlDefineVariableExtHandle)
<define all possible fields>
   (LlDefineFieldStart,
    LlDefineField,
    LlDefineFieldExt,
    LlDefineFieldExtHandle)
<start print>
   (LlPrintStart,
    LlPrintWithBoxStart)
<define options>
   (LlPrintSetOption,
    LlPrintSetOptionString,
    LlSetPrinterDefaultsDir,
    LlPreviewSetTempPath)
<let user change options>*
   (LlPrintOptionsDialog,
    LlPrintOptionsDialogTitle,
    LlPrintSelectOffset,
    [LlPrinterSetup])
<define constant variables>
   (LlDefineVariable,
    LlDefineVariableExt,
    LlDefineVariableExtHandle,
    LlPrintIsVariableUsed)
<check sortorder>*
   (LlPrintGetSortOrder)
<get printer info for progress-box>*
   (LlPrintGetOption,
    LlPrintGetPrinterInfo)
<while no error or end of file or user abort>
   {
   <give progress-report>*
      (LlPrintSetBoxText,
       LlPrintGetCurrentPage,
       LlPrintGetOption)
   <define variables>
      (LlDefineVariable,
       LlDefineVariableExt,
       LlDefineVariableExtHandle,
       LlPrintIsVariableUsed)
   <hide objects>*
      (LlPrintEnableObject)
   <print variables>           ; print all objects
      (LlPrint)
   <repat until warnings>
```

```
      (LlPrint)
   <repeat>
      {
      <define fields>
         (LlDefineField,
          LlDefineFieldExt,
          LlDefineFieldExtHandle,
          LlPrintIsFieldUsed)
      <print lines>
         (LlPrintFields)
      <no warning or user abort: next data record>              }
   <until
      - error or
      - end of file or
      - user abort or
      - page full-warning>
   }
<Test if footer can be printed on the expected last page>
   (LlPrintFieldsEnd)
<if page full warning repeat>
      (LlPrintFieldsEnd)
<end print>
   (LlPrintEnd)

<if preview>
   {
   <show preview>*
      (LlPreviewDisplay)
   <delete preview-files>*
      (LlPreviewDeleteFiles)
   }
```

3.2.2. 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 have to 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.

Additionally there are special options the user may wish to enter, like first page number, number of copies, choice of printer, etc. That's what the print options dialog of List & Label is for.

Print Meter Bar and Multitasking

Most users would like the possibility of interrupting the print. It is customary to supply the user with a small "abort" dialog box on the display with which it is possible to stop the print. It's even better if the user is informed in the box of how far the print has advanced.

List & Label offers a function which starts a print-abort dialog box with meter bar and the essential multitasking overhead. This box can then later be provided with the present values (progress in percent, page number, etc.)

By using this function you don't have to use a message-loop. Thus, if you call

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

and no error is returned from this function then 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, "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 "permanent changes" has been checked in the dialog then the chosen printer setting is saved in the corresponding printer-definition file *.??p. Initially this is determined in the designer under project page layout. If this file is not found then the windows standard printer is used.

Limitations for Print Pages by the User

If the user chooses a first and a last page then it's a bit more complicated. If you want to check this particular case it can be tested - the values for the first and last page have to be "starting page" and INT_MAX (2147483647) if you want to be able to "print all pages". "Starting page" is the page number of the first page.

List & Label can - even without the support of the programmer - evaluate the limitations of the number of pages so that in the simplest case the programmer does not have to consider this.

However we recommend aborting the printing loop when the current page is larger than the last page!

Copies

"Copies" can mean two different kinds of copies:

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

To support this, get the number of copies before the first *LlPrint()* call to 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.

Important: After setting the copies, you always need to query List & Label whether the printer supports the number of copies, even in the case of 1, as calling LL_PRNOPT_COPIES_SUPPORTED will set the number of requested copies in the printer driver.

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

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

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.

It would be very easy if the printer itself would just handle this... but not all printers really do support copies.

Now you have to decide: do you want to make the copies yourself if the printer does not support them, or do you like to offer copies only when the printer can handle them?

If you only want to support copies if the printer does, use *LlPrintSetOption(hJob, LL_PRNOPT_PRINTDLG_ONLYPRINTERCOPIES,TRUE)* to disable the 'copies' edit box in the print setup dialog if the printer does not support copies.

If you want to do the copies yourself, you must ask whether the printer supports it:

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

// set our own counter to 1
nCopies = 1;
if (LlPrintGetOption(hJob, LL_PRNOPT_COPIES_SUPPORTED) == 0)
{
   // we have to do it ourselves: get the amount of copies
   nCopies = LlPrintGetOption(hJob, LL_PRNOPT_COPIES);
   // and reset LL copy count to 1 (being cautious)
   LlPrintSetOption(hJob, LL_PRNOPT_COPIES, 1);
}
```

After this, you have to issue the overall printout of your project nCount times, that is, the whole printing including *LlPrint[WithBox]Start()* and *LlPrintEnd()*. The call with

LL_PRNOPT_COPIES_SUPPORTED also sets the copies in the printer driver to the required value and is crucial, even if you are sure that the printer supports hardware copies.

List & Label has an option for programmers who do not want their user to be able to choose a "copies" number at all: when *LL_OPTION_COPIES* is set to *LL_COPIES_HIDE* then the query does not appear at all in the dialog-box.

Optimizing Speed

a) Program optimization

Variable definitions which remain constant during the print can be taken out of the print loop. If, for example, you always wish to have your company name on the top of a list, then this can best be defined outside the loop before *LlPrintStart()*. *LlPrintWithBoxStart()*. Don't forget – DO NOT delete the variable buffer in the loop with *LlDefineVariableStart()*!

b) Is the variable or field used?

It is also possible to ask which variables or fields are used in the print loop. If the amount of variables or fields is notably higher than the number actually used in the project then it is worth using this function.

This can only be done if the print has been started, that is after *LlPrint[WithBox]Start()*, as it is precisely these functions which read the project, check the use of variables and mark them as used!

When defining variables/fields during the print you ask List & Label if these variables are needed:

```
if LlPrintIsVariableUsed(<variable>) then
   <define variable>
```

Also with fields:

```
if LlPrintIsFieldUsed(<field>) then
   <define field>
```

c) Reserved characters in variable or field names

Don't use reserved characters in variable or field names. Then you can switch off the check by calling *LlSetOption(LL_OPTION_XLATVARNAMES,0)* and reduce the overhead when defining the variable contents.

d) Parameter check

When your application is finished and tested, you can switch off List & Label's parameter check, which also reduces the overhead on each API call: LlSetOption(HLLJOB, *LL_OPTION_NOPARAMETERCHECK*, 1)

e) RTF control

Use this sparse, if at all. This control takes a large amount of time to process its formatted text and so a print will take much longer if you use such a control.

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 which are not lists are printed, as well as the list headers (if the option *LL_OPTION_DELAYTABLEHEADERLINE* is not set, else 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 list. If this is not the case, then *LL_WRN_REPEAT_DATA* is returned which suggests that normally a new page should be started.

When the lists have been filled the variables for linked objects have to be defined before returning to the beginning of the loop (therefore calling *LlPrint()*), as with this *LlPrint()* the linked objects are filled, the new page is started and - see above - the objects on the new page including list headers printed.

Important: fields for footer lines must be defined at the same time as the fields for data lines, for example for invoices the total amount should include the price data of the current record. List & Label takes care that - if a record does not fit at all on the current page - the previous value will be used.

The print scheme is then:

```
<print start>
   (LlPrint[WithBox]Start)
<while not error or finished>
   {
   <define 'normal' variables>

   <start new page, print objects>
      (LlPrint)
   <while not
      - error or
      - LL_WRN_REPEAT_DATA or
      - finished>
      {
      <define fields>

      <print fields>
         (LlPrintFields)
      <if not LL_WRN_REPEAT_DATA>
         <load next record>
      }
   <define variables for linked objects>
   }

<end table>
   (LlPrintFieldsEnd)
<while LL_WRN_REPEAT_DATA>
   (LlPrintFieldsEnd)
<done>
   (LlPrintEnd)
```

And in the case of multiple lists:

```
<print start>
   (LlPrint[WithBox]Start)
<while not error or finished>
   {
   <define 'normal' variables>

   <start new page, print objects>
      (LlPrint)

   <disable list 2>
   <while not
      - error or
      - LL_WRN_REPEAT_DATA or
      - finished>
      {
      <define fields>

      <print fields>
         (LlPrintFields)
      <if not LL_WRN_REPEAT_DATA>
         <load next record>
      }
   <enable list 2>

   <disable list 1>
   <while not
      - error or
      - LL_WRN_REPEAT_DATA or
      - finished>
      {
      <define fields>

      <print fields>
         (LlPrintFields)
      <if not LL_WRN_REPEAT_DATA>
         <load next record>
      }
   <enable list 1>

   <define variables for linked objects>
   }

<end table>
   (LlPrintFieldsEnd)
<while LL_WRN_REPEAT_DATA>
   (LlPrintFieldsEnd)
<done>
   (LlPrintEnd)
```

A forced page break is possible by leaving the field definition loop, as *LlPrint()* ends the present page if this has already been partially filled.

Group footer lines will only be printed if a data line has been printed in the same group (line(s) with the same result in the group condition formula).

3.2.3. Multiple Table Objects

To print 2 different tables on one page you can use the following scheme:

Place both table objects and assign a name to them, so you can access the objects. To fill the tables follow these steps:

- Start a normal print job and call *LlPrint()*.
- Disable table 2, enable table 1 (see *LlPrintEnableObject*()).
- Fill table 1 until *LL_WRN_REPEAT_DATA* is returned or no data should be printed.
- Disable table 1, enable table 2.
- Fill table 2 until *LL_WRN_REPEAT_DATA* is returned or no data should be printed.
- Enable both tables and call *LlPrint()*.

For most cases more suitable and the recommended strategy is the usage of the specialized API functions described in the following chapter.

3.2.4. Linked Objects

In List & Label it is possible to make objects inter-dependent. Spatial dependence is of little interest to our program as List & Label handles this automatically, but the temporal sequence can be important! This also and especially may hold for chart objects.

If the project to be printed is not a list project then even the temporal dependence is not of importance, as with *LlPrint()* all objects are filled in their defined order and then printed. It is not possible to change variables in between, so this is program-technically not important.

It gets more interesting when objects are defined before and after a list, as in this case it is possible to print objects which are linked to a list after filling in the corresponding list.

```
<print start>
   (LlPrint[WithBox]Start)
<while no error or finished>
   {
   <define 'normal' variables, that is: variables that
    are used in objects before the list>
   <page break (if needed), print objects before list>
      (LlPrint)
   <while not
      - error or
      - LL_WRN_REPEAT_DATA or
      - finished>
      {
      <define fields>
      <print fields>
         (LlPrintFields)
      <if not LL_WRN_REPEAT_DATA>
         <load next record>
      }
   <define variables for linked objects>
   }

<end table>
   (LlPrintFieldsEnd)
<while LL_WRN_REPEAT_DATA>
   (LlPrintFieldsEnd)
```

```
<done>
   (LlPrintEnd)
```

List & Label uses the following pattern (caution: this has been modified slightly since List & Label 5).

(1) *LlPrint()* prints all objects which are not linked to a list. With list objects the header lines are being printed (if the option *LL_OPTION_DELAYTABLEHEADERLINE* is not set, else the table header will be delayed until the first data line is to be printed)

(2) With each call of *LlPrintFields()* one line is appended to the table. When a table is full - the data record can not or only partially be printed - then List & Label returns *LL_WRN_REPEAT_DATA* to the program.

(3) The following *LlPrint()* prints the footer line and the linked objects before a page break is done (and, see 1, the "first" objects are being printed).

(4) *LlPrintFieldsEnd()* prints the footer line (if any) and the linked objects. If those do not fit on the page - partially or at all), List & Label returns *LL_WRN_REPEAT_DATA*

(5) *LlPrintEnd()* finishes the print.

Some examples which can be easily realized using the linking:

- A page can be produced which, as in dictionaries, has the first and last record in the list displayed somewhere on the page. The first record is written into a variable which is passed in an "object designed before the list". After the list is full the last record is then written into a variable which is used in an object linked to this list. In this example the variable is "first author" in a non-connected object, and "last author" in an object linked to the list.

```
LlPrintStart()
< ... the normal (see above) ...>

<while data records available and error = 0>
   {
   LlDefineVariable("First author",
      <Author of the present data record>)
   LlPrint();
   Last author = "";
   <while data records available and no error>
      {
      <define fields>
      LlDefineVariable("Last author",
         Last author)
      error = LlPrintFields();
      if (error <> LL_WRN_REPEAT_DATA)
         {
         if LlPrintDidMatchFilter()
            last author = <present author>;
         <next data record>
         }
      }
   if (error = LL_WRN_REPEAT_DATA)
      error = 0
   }
```

```
while LlPrintFieldsEnd() = LL_WRN_REPEAT_DATA
   LlPrintFieldsEnd()

LlPrintEnd()
```

- Invoices with variable list lengths are no problem as the linked object (in this case a text object with the subtotal and total of the invoice amount) can always be linked to the list, and only has to be filled when the partial total of the list on this page is known because these data records have already been passed. There is one restriction: in contrast to a footer line, only items are added to the sum for objects that have been successfully finished. In a footer line, List & Label manages this so that the sum includes the current item if this has been printed at least partially.

```
LlPrintStart()
< ... the normal (see above) ...>
error = 0
total = 0
<while data records available and error = 0>
   {
   subtotal = 0
   <define variables which are required before table>
   LlPrint();
   <while data records available and no error>
      {
      <define fields>
      <define variables which are required after table>
      error = LlPrintFields();
      if (error <> LL_WRN_REPEAT_DATA)
         {
         if LlPrintDidMatchFilter()
            {
            total = total + amount in the data record
            subtotal = subtotal + amount in the data record
            LlDefineVariable("subtotal", subtotal);
            LlDefineVariable("Total", total);
            }
         <next data record>
         }
      }
   if (error = LL_WRN_REPEAT_DATA)
      error = 0
   }
while LlPrintFieldsEnd() = LL_WRN_REPEAT_DATA
   LlPrintFieldsEnd()

LlPrintEnd()
```

Please note that table objects cannot be linked to other table objects. Use the "multiple table lines" definition feature to define different "virtual" tables.

3.2.5. Preview

It is very easy to include the option of a print-preview. You only need very few changes in the print routines, so it is advisable to place both in one routine and to show the difference using a flag:

First difference: *LL_PRINT_PREVIEW* or *LL_PRINT_USERSELECT* instead of *LL_PRINT_NORMAL* in the corresponding parameter of *LlPrint[WithBox]Start()*.

In avoidance of network rights management conflicts you can set the path of the preview files by using *LlPreviewSetTempPath*(). The path which is used as standard is the one where the definition file is stored.

This path - if used explicitly - is then also needed as parameter for the final preview calls (*LlPreviewDisplay()*, *LlPreviewDeleteFiles()*).

After *LlPrintEnd()*, as long as no error has occurred (don't forget error checks), you can show the preview using *LlPreviewDisplay()*.

After the display, you should delete the preview files as you will not need them any more, except if you want to archive or print/send them outside the preview.

A call could look like:

```
LlPrintWithBoxStart(hJob, LL_PROJECT_LABEL, "test",
  LL_PRINT_PREVIEW, LL_BOXTYPE_NORMALMETER, hWnd, "page display");

LlPreviewSetTempPath(hJob, "t:\\");

<Option-Dialog as before, with abort>

<print as before, with error checks>

LlPrintEnd(...);
if (!Error)
   LlPreviewDisplay(hJob, "test", "t:\\");
LlPreviewDeleteFiles(hJob, "test", "t:\\"); // optional
```

As you can see, with these small differences the print routine can be changed in its function to a preview routine.

As alternative you can give the user a chance to select the output medium (printer, file, export or preview). Thus your code will be absolutely independent of the medium:

```
LlPrintWithBoxStart(hJob,LL_PROJECT_LABEL,"test",
  LL_PRINT_USERSELECT, LL_BOXTYPE_NORMALMETER, hWnd, "output");
LlPreviewSetTempPath(hJob,"t:\\");

<Option-Dialog as before, with abort>

<print as before, with error checks>

bPreview =
LlPrintGetOption(hJob,LL_PRNOPT_PRINTDLG_DEST)==LL_DESTINATION_PRV;
LlPrintEnd(...);

if (bPreview)
   {
   LlPreviewDisplay(hJob, "test", "t:\\");
   LlPreviewDeleteFiles(hJob, "test", "t:\\"); //optional
   }
```

As preview files can be printed or viewed without any quality loss, you can archive or send them to give others access to them. Just copy or send the <project name>.LL file to the destination before deleting it with *LlPreviewDeleteFiles()*.

For additional usage of the export modules you should use the functions *LlPrintGetOptionStr()* and *LlPrintSetOptionStr()* with the flags *LL_OPTIONSTR_EXPORTS_ALLOWED*, *LL_PRNOPTSTR_EXPORT*. For further information, please read the chapter on the export modules.

3.3. Handling Chart and Crosstab Objects

The easiest way to work with charts and crosstabs is to insert them into the report container. See chapter "4. Working with the Report Container" 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" does also apply.

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.

3.3.1. The Standard Mode (Default)

This mode can only be used with list projects and does not need any changes to existing projects. The chart objects are fed with the same data as the table objects, a *LlPrintFields()* both sends data to chart objects and table objects. However, the charts only accumulate the data and aren't printed right away. This mode is contained for compatibility reasons and can be used to easily fill charts that are linked to table objects.

3.3.2. The 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 not   error or finished>
    {
    <define variables>
    <while not
    - error or
    - 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)
    <next record>
    }
<done>
    (LlPrintEnd)
```

Of course, all used chart fields have to be declared before calling the designer as well in order to enable your users to use them at all.

4. Working with the Report Container

List & Label offers a comfortable possibility to design 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. As a basic principle you use the project type *LL_PROJECT_LIST* for such projects.

In the following we use "table" as a synonym for a group of matching fields in the List & Label-Designer resp. in the report structure 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", resp. all member variable of a class. In the List & Label-Designer you will work just with one "report container object". This object can contain multiple tables, crosstabs and charts.

Once you have added single tables with *LlDbAddTable()* your users can work with a new tool window – the report structure 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.

The .NET and VCL components offer a comfortable way to bind to a database – if your data can be represented in a dataset with relations (.NET) or a TDataSet descendant (VCL), you don't need to care about anything else. In that case you can skip the following sections.

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

4.1. API functions needed

The name of the API functions needed to control this functionality begin with *LlDb…* resp. *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()*, *LlPrintDbGetCurrentRelation()*). You will find detailed descriptions later in this chapter.

4.2. Calling the Designer

At first all tables have to be declared to List & Label, so that they can be inserted in 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 resp. an empty string, the table ID will be used as display name as well.
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've 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.
The remaining action when calling the Designer is analogous to the "normal" call, i.e. the whole scheme for calling the Designer with multiple tables looks like:

```
<open job>
   (LlJobOpen, LlJobOpenLCID)
<define List & Label-settings>
   (LlSetOption,
    LlSetOptionString,
    LlSetDebug,
    LlSetFileExtensions,
    LlSetNotificationMessage,
    LlSetNotificationCallback)
<define dialog-settings>
   (LlSetDlgboxMode)
<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 to 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 "4.4. Handling 1:1 relation".

4.3. Controlling the print engine

The control of hierarchical reports with List & Label happens more or less analogous 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 particular 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.

4.3.1. Multiple independent tables on the same level

An example for this would be a listing 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 of the last chapter – with one difference. Usually, you tell List & Label that a table is finished (no more data) by calling *LlPrintFieldsEnd()*. Now you can possibly 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 the print of a table could be started. For ease of reading the optional part of the print loop is not shown here, as it already has 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,
     LlPrintIsVariableUsed)
<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,
        LlPrintIsFieldUsed)
       <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 contains the information 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 preview>
{
   <show preview>
      (LlPreviewDisplay)
   <delete preview files>
      (LlPreviewDeleteFiles)
}
```

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.

4.3.2. Simple 1:n relations

The typical example for this case is the already discussed 1:n relation order – order details. After each record with order data the order details for that data shall be printed.
The print of a data line is triggered by *LlPrintFields()*. Analogous to the behavior of *LlPrintFieldsEnd()* in the last section the function returns *LL_WRN_TABLECHANGE* if the user 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 has to 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,
       LlPrintIsFieldUsed)
```

```
 <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>
}
```

With this code any sequence of tables and sub tables can be printed. The recursion makes sure that it works fine with any "deepness", i.e. this code can control arbitrary multilevel relations.

4.3.3. 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 two last 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 of the data source as percentage. This attempt does not work as soon as the user puts two tables in series. Therefore List & Label offers the possibility 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 percent position in the current table. For properly updating 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)>
     }
   ...
  }
...
}
```

4.3.4. Supplying master data as variables

In case of an order with the corresponding 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. Thus, at the root level of the table object you want to have access to the sub tables of the master table. In order to achieve this, call *LlDbSetMasterTable()*. The needed 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 4.3.1.):

```
<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)
<if preview>
{
   <display preview>
      (LlPreviewDisplay)
   <delete preview files>
      (LlPreviewDeleteFiles)
}
```

4.4. Handling 1:1 relations

When reporting 1:1 relations, the data is usually combined with a JOIN, so that the data is available in one table. If this is not the case or you don't 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.

4.4.1. 1:1 relations without a key field definition

If the key fields for the relation are not relevant, for example it is 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:

You have to make sure that while printing the OrderDetails table you fill this field for each record with the corresponding value, of course.

4.4.2. 1:1 relation with key field definition

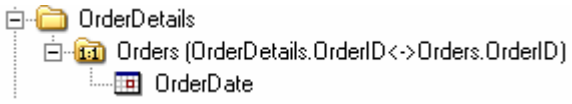
In case of multiple 1:1 connections it might be important for the user to see which of the key fields are linking the tables together. In this case you can declare the fields as follows:

<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, keep in mind to update the field contents when the parent table is being printed!

4.4.3. 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 used, you can call

```
LlPrintIsFieldUsed(hJob, "OrderDetails.OrderID@Orders.OrderID*");
```

Is the result is 0, no field of the table orders is used and it is not necessary to update the values.

5. Working with the .NET Component

Apart from the OCX and VCL components for Visual Basic and Delphi/C++ Builder, an additional component, the List & Label .NET Assembly, is for development with Microsoft .NET. You can skip this chapter if you do not use the .NET framework.

5.1. Main differences

The component is optimally integrated into the framework and can be distinguished from normal APIs. Among other things, the assembly offers the following special features:

- Extensive DataBinding, similar to common Windows and Web Forms Controls
- Easy transfer of any data as variables and fields
- Use of .NET exceptions instead of return values
- Simplification of multiple method calls by means of overloads

5.2. Integrating components

Add a reference to the file listlabel12.dll resp. listlabel12unicode.dll in order to integrate List & Label in the Visual Studio .NET development environment. For Visual Studio 2005 separate 2.0 Framework Assemblies are available which are optimized for this environment.

Subsequent to integration, multiple symbols will appear in the component area of the toolbar. In some cases, the symbols need to be added manually. In this case, just select them in the menu via *Tools > Customize Toolbox*.

If the component is displayed in the toolbox, it can be moved to the active form's work area per drag & drop. This is possible with Windows and web forms. Since the component itself is not visualized in the form, a corresponding icon is displayed in the form tray. When you click on it, the properties window will display the component's properties. Here, you can make multiple important settings, each of which are briefly described below.

If the component is integrated, you are advised to reference the combit.ListLabel12 name space. Depending on the language and place of function, it appears as follows:

```
using combit.ListLabel12; // C#
Import combit.ListLabel12 'VB.NET
<% @Import Namespace="combit.ListLabel12" %> <!—ASP.NET ->
```

Now you can start adapting List & Label to your individual requirements with the properties offered, and implement the required program logic. Three different approaches are offered for this purpose:

- DataBinding
- Easy print and design methods

- A separate, repeating print loop

The first two options are presented subsequently. The repeating approach largely corresponds to the direct application of DLL and is, therefore, covered by the general description of the List & Label API.

5.3. DataBinding

The List & Label component for Microsoft .NET has extensive options for direct DataBinding. The implemented system is modeled on known Windows and web form controls. Accordingly, two important properties are available. DataSource indicates the data source and, optionally, DataMember indicates a member of this source. There is flexibility in the selection of the data source. Among others, the following data types are supported:

- DataSet (optional information of the desired table via DataMember)
- DataTable
- DataView
- DataViewManager
- OleDbDataReader and SqlDataReader (via IDataReader)
- Generic list classes (Framework 2.0)
- IListSource, IList, and IEnumerable interfaces

The IEnumerable interface, in particular, enables the transfer of almost any data, even custom collections, for instance. The object (IEnumerator.Current) provided by the enumerator is declared through reflections. Here, the name and content of all the properties and the public fields offered by the respective object are used.

5.3.1. Binding List & Label with a data source

DataBinding occurs with the property DataSource. You can assign this programmatically or implicitly with the SetDataBinding method. Alternatively, the component offers UI support to the Visual Studio .NET development environment.

If you have already placed a DataSet on your form, you can select it with the properties window. The necessary link is created automatically.

You can now implement the program code in order to start designing and printing. Here, for instance, use the component's Print resp. Design method in the Click-event of a button without additional parameters. The data of the assigned source is provided automatically.

```
// Assign data source
listLabel1.SetDataBinding(myDataSet, "Name of the table");

// Display designer
listLabel1.Design();

// Execute printing
listLabel1.Print();
```

A lot of properties are available to you, if you want to modify the standard behavior of databound printing. These begin with "Auto" and are found in the data area of the properties window.

Property	Description
AutoDesignerFile	File name of the print project to be used
AutoDestination	Print format, for instance, printer, preview, PDF, HTML, etc.
AutoProjectType	Print project type (list, card, label)
AutoFileAlsoNew	New project possible in case of designer call
AutoShowPrintOptions	Display of print options when printing is started
AutoShowSelectFile	Display of Select File dialog for printing and designer
AutoBoxType	Kind of progress bar
AutoDialogTitle	Title for the dialogs (e.g. Designer, progress bar)

The DataBinding for Version 11 has been completely rewritten. You can activate the old mode by setting the property "AdvancedDataBinding" to "False". For new projects you should always use the default value "True".

5.3.2. Working with Master Detail Data Records

In connection with DataBinding and list projects, List & Label can automatically evaluate and accept existing relations between multiple tables of a DataSet object. Such relations are established between two columns (DataColumn) with the help of a DataRelation. This, for instance, appears as follows:

```
DataColumn dc1 = mydataset.Tables["invoice"].Columns["billno"];
DataColumn dc2 = mydataset.Tables["items"].Columns["billno"];
DataRelation dr = new DataRelation("invoice2items", dc1, dc2);
mydataset.Relations.Add(dr);
```

The name given clearly identifies the relation. This way, it is possible to establish any number of relations which List & Label can evaluate parallel at runtime.

The data transfer type is determined with the help of the property AutoMasterMode. The underlying enumeration provides the following values:

- None: No Master Detail Relations are evaluated (only in the old "mode")
- AsFields: Master and Detail data are both declared as fields. In this manner, you can create group formation, statistics, and overviews.
- AsVariables: The master data is declared as variables and the detail data as fields. The project is reset internally by means of LlPrintResetProjectState, according to each master data record. In this manner, you can print multiple identical reports with different data one after the other, in a single print job, e.g. multiple invoices.

Also, please note the DataBinding examples given.

5.3.3. Additional DataBinding options

The DataBinding mode of the component provides you with four different events which allow you to influence the result. The table provides an overview.

Event	Description
AutoDefineNewPage	This event is called for every new page and enables the registration of additional variables for this page. The IsDesignMode property of the event argument indicates whether it is being called in design or print mode..
AutoDefineNewLine	This event is called for each line before the automatic declaration of databound fields. Similar to AutoDefineNewPage, you can declare additional fields.
AutoDefineVariable	This event is called for each variable that is displayed automatically through DataBinding. You can individually manipulate the name and content of each variable before transferring it to the print engine via the properties name and value.
AutoDefineField	Similar to AutoDefineVariable for fields
AutoDefineTable	This event is called for each table which is added with *LIDbAddTable()*. You can manipulate e.g. the name or suppress the handing over.
AutoDefineTableSortOrder	This event is called for each sorting which is added with *LIDbAddTableSortOrder()*. You can manipulate e.g. the name or suppress the handing over.
AutoDefineTableRelation	This event is called for each data relation that is added with *LIDbAddTableRelation()*. You can manipulate e.g. the name or suppress the handing over.

5.4. Easy print and design methods

The print and design methods of the .NET component are implemented in a manner similar to the corresponding OCX and VCL methods. The methods implement a standardized print loop, which can be used directly for a majority of easier applications if you don't have your data in an object which can be used for DataBinding. Both methods have a lot of overloads, through which, for instance, you can indicate the print project, print destination, and project type. Standard values are taken for these if the methods are called without parameters.

In the case of this approach, data is transferred to List & Label within the DefineVariables and DefineFields events. You can individually link to any data source in this manner. Event arguments allow access to useful information like the transferred user data, design mode, etc. With the property IsLastRecord, the print loop is notified that the last data record has been reached. If this is not the case, the respective event is repeatedly called in order to poll the data.

You can set additional options in the DefinePrintOptions event. This event is triggered internally after LlPrintWithBoxStart has been called but before the actual printing.

A very easy use of the print method appears as follows:

```
private void button1_Click(object sender, System.EventArgs e)
{
   ListLabel LL = new ListLabel();
   LL.DefineVariables += new DefineVariablesHandler(LL_DefineVariables);
   LL.Print(LlProject.List, "test.lst", false);
}

private void LL_DefineVariables(object sender, DefineElementsEventArgs e)
{
   ListLabel LL = (ListLabel) sender;
   LL.Variables.Add("my variable", "content");
   //Only one record shall be printed
   if(!e.IsDesignMode)
      e.IsLastRecord = true;
}
```

5.4.1. Applying UserData parameters

Overloads of the print and design methods enable the transfer of an object-typed UserData parameter. With the help of this parameter, you can pass any data to the DefineVariables and DefineFields events. There, the transferred object is available within the event arguments.

With this system, you can pass the required data, such as a DataReader, to the subordinate event, without having to declare it as a global variable. The listing shows this with an example.

```
OleDbDataReader reader = …
if(reader.Read())
   LL.Print(reader, LlProject.List, "test.lst", …);

private void LL_DefineVariables(object sender, DefineElementsEventArgs e)
{
   ListLabel LL = (ListLabel) sender;
   OleDbDataReader reader = (OleDbDataReader) e.UserData;

   for(int I=0; I<reader.FieldCount; I++)
      LL.Variables.Add(reader.GetName(I), reader.GetValue(I));
   if(!e.IsDesignMode)
      e.IsLastRecord = (!reader.Read());
}
```

5.5. Passing unbound variables and fields

The transfer of variables and fields differs from the regular List & Label principle and is adapted to the object-oriented approach of .NET. The component offers two

properties, variables and fields, which provide a VariableCollection or FieldCollection type of collection. Both were derived from the same abstract base, thus the subsequent description deals only with the VariableCollection. Everything shown there applies to the FieldCollection too.

The Add method of the VariableCollection allows you to declare a new variable. This internally results in calling the API function LlDefineVariable. Besides the name, the content of the variable is also passed. There exist multiple overloads that convert CLS data types to the internal List & Label types. According to that, the overloads differ in the second parameter, the possible data types of which are shown in the table.

CLS / Framework	C#	VB.NET	List & Label
System.String	string	String	LL_TEXT
System.Char[]	char[]	Char	LL_TEXT
System.Int16	short	Short	LL_NUMERIC
System.Int32	int	Integer	LL_NUMERIC
System.Int64	long	Long	LL_NUMERIC
System.Single	float	Single	LL_NUMERIC
System.Double	double	Double	LL_NUMERIC
System.Decimal	decimal	Decimal	LL_NUMERIC
System.Boolean	bool	Boolean	LL_BOOL
System.DateTime	DateTime	Date	LL_DATE
System.Object	object	Object	- According to the content -

The overload for the uppermost data type, System.Object, implements complicated type conversion. Here, an attempt is made to dynamically convert the passed object into one of the known data types. If this is not possible, a text variable is registered whose content results from applying the ToString method to the object.

The following lines indicate the registration of a string, a numerical value and a date. In each case, the corresponding List & Label variable types are applied and provided in the designer.

```
LL.Variables.Add("mytext", "hello world"); // -> string
LL.Variables.Add("mynumber", 51515); // -> Numerical data
LL.Variables.Add("mydate", DateTime.Now); // -> Date
```

Besides the listed standard data types, overloads also exist for the following classes:

- StringBuilder
- Bitmap
- Icon
- Metafile

Before variables and fields are passed the engine checks internally if they are used in the print layout at all. To reduce the number of calls a cache is constructed with the used fields and variables. You don't have to take care about this process by means of *LlPrintIsFieldUsed()* resp. the counterparts for variables and chart fields – everything happens automatically.

5.5.1. Images

Graphics such as bitmaps and metafiles are transferred via overloads. The following call can be used for instance, to transfer the graphic c::\myimage.jpg to List & Label:

```
LL.Variables.Add("my image", new Bitmap("c:\\my image.jpg"));
```

Tip: For legal, license-related reasons, the GIF format is not directly supported by List & Label. However, the .NET framework's bitmap class recognizes the GIF format, so registration of GIF images is theoretically possible in an indirect manner, provided the legal license-related aspects allow the use of the GIF format.

5.5.2. Barcodes

The transfer of barcodes has been extremely simplified in the .NET component. A barcode is represented with the LlBarcode class. The content of the barcode and the desired format are transferred to the constructor. The required formatting with pipes "|" is automatically applied. The class instance constructed in this manner can be passed to the Add method of the VariablesCollection.

```
LlBarcode bc = new LlBarcode("123456789012", LlBarcodeType.EAN13);
LL.Variables.Add("myBarcode", bc);
```

5.5.3. Dictionaries

You can register various dictionaries with key-value relations via List & Label, using multiple overloads. A special AddFromDictionary method of the VariablesCollection is available for this purpose. You can transfer classes, which have been derived from the abstract base class NameObjectCollectionBase or which implement the IDictionary interface. The NameValueCollection and Hashtable classes belong to the best-selling category.

The method iterates the dictionary and passes all the data contained within. The given key is automatically used as variable name. It is also possible, for instance, to pass a variable prefix for the classification of the variable list.

```
Hashtable customer = new Hashtable();
customer.Add("Firstname", "John");
customer.Add("Lastname", "Doe");
LL.Variables.AddFromDictionary(customer);
```

5.5.4. Streams and TextReader

You can pass extensive data directly from the respective source. For this purpose, the component offers the methods AddFromStream and AddFromTextReader, to which you can pass a derivation of the respective abstract class. In this manner, you can, for instance, register entire files with List & Label (FileStream class).

5.5.5. Data rows

Overloads of the AddFromDataRecord method allow the transfer of classes, which support the DataRecord interface. For instance, the classes OleDbDataReader and SqlDataReader belong to this interface. The respective existing data records are passed in each case and the data record index is not moved (Read method). It is also possible to pass a DataRow class instance to the methods. An additional function AddFromDataRowView allows the transfer of a DataRowView. With it the data types from the scheme are wrapped.

5.5.6. RTF texts and other List & Label data types

When speaking of the methods covered so far, data is automatically converted to the internal List & Label data types. Thus, numerical values are assigned to LL_NUMERIC, dates to LL_DATE, and logical values to LL_BOOL. For certain formats, there are no corresponding data types in the .NET framework, for these objects List & Label has to be explicitly notified. In this connection, LL_RTF and LL_HTML are to be mentioned, in particular.

Also, the methods presented until now always allow you to explicitly describe the List & Label data types to be used with overloads. You can thus, for instance, use the AddFromStream method to register an RTF file.

```
FileStream myFileStream = File.OpenRead("c:\\test.rtf");
LL.Variables.AddFromStream("RTF-Text", myFileStream, LlFieldType.RTF);
```

The other Add… methods also have such an overload. The desired data type originates from the Enumeration LlFieldType.

5.5.7. Handles

The API function LlDefineVariableExtHandle is represented by the AddFromHandle method. Here, you can pass a handle in the form of an IntPtr. It is also possible to determine the description of the desired data type. Generally, it is the LlFieldType. Drawing_hBitmap.

Remark: This method should be used only in special situations, such as connecting with Win32 API, if, for example, a component with unmanaged code delivers such a handle. In other cases, you can directly enter the respective Bitmap. The basic abstract class Image also supports a static method FromHBitmap, with the help of which an instance of the Bitmap class can be indicated on the basis of a handle.

5.5.8. Any other data type

The List & Label component allows the declaration of any type of data. The AddFromObject method is available for this purpose. This method uses reflections to retrieve the public fields and properties of the object and to pass them as variables to List & Label. The name of the corresponding class member is used as name, wherein you can also assign a prefix to a name through an overload. The following line shows passing of a cookie within a Web reporting project:

```
LL.Variables.AddFromObject("MyCookie", Request.Cookies["test"]);
```

The following variables are automatically declared through this call:

- MyCookie.Domain
- MyCookie.Expires (LL_DATE)
- MyCookie.HasKeys (LL_BOOL)
- MyCookie.Name
- MyCookie.Path
- MyCookie.Secure (LL_BOOL)
- MyCookie.Value
- MyCookie.Values

5.5.9. Collection Options

The classes VariablesCollection and FieldsCollection are derived from the basic abstract class ElementBaseCollection. It implements the IDictionary interface and thus, ICollection and IEnumerable indirectly. You can use the standard methods and techniques. These include the following:

- With the items property or indexer, you can store the content of a variable by passing its name. A class derived from the basic abstract element is returned to the user.
- You can cover all variables through a for each…in loop. Here as well, a class derived from the element is returned to the user.
- The Clear method resets the contents of the collection and simultaneously calls the API function LlDefineVariableStart or LlDefineFieldStart internally for this purpose.
- Count provides the number of registered variables
- Contains queries whether a variable has already been registered

When accessing a variable, either through a variable name or through an enumeration, a class derived from the basic abstract element is returned to the user. With the properties name and value, this allows access to the name and the currently assigned value of the variable. With the IsUsed method, you can find out whether the variable is being used in the current print project. This is internally done by calling the API function LlPrintIsVariableUsed or LlPrintIsFieldUsed.

The following source code part shows the enumeration of all the registered variables. For each variable, a MessageBox is displayed with the name and content.

```
foreach(Variable var in listLabel1.Variables)
{
   MessageBox.Show(string.Format("{0} -> {1}",
   var.Name, var.Value));
}
```

5.6. Language selection

The List & Label Designer is available in many languages. As a result, the component gives you intensive support while creating multi-lingual desktop and Web applications. There are several options for choosing the language that is to be used by List & Label.

1. Assign the property Language to the corresponding language:
   ```
   LL.Language = LlLanguage.English;
   ```

2. Pass the language to the constructor of the component:
   ```
   LL = new ListLabel(LlLanguage.English);
   ```

3. Pass the culture to be used, to the constructor of the component from the current thread, for instance:
   ```
   LL = new combit.ListLabel12.ListLabel(CultureInfo.CurrentUICulture);
   ```

   The language is now selected on the basis of the UI language of the current thread and is set automatically by List & Label.

4. You can also flexibly control the language with the application's configuration file (<exe>.config). For this, you must select the default language for the program (LlLanguage.Default).

   ```
   <configuration>
   <appSettings>
   <add key="ListLabel Language" value="English" />
   </appSettings>
   </configuration>
   ```

   If you modify the language in the configuration, it is automatically used the next time the application is started. All the LlLanguage enumeration values are usable.

Remark: Please note that in order to display the respective language you need the corresponding language kit. Find out about the kits we presently offer and their costs from our online shop at www.combit.net.

5.7. Working with events

List & Label offers a multitude of callbacks that are implemented for the List & Label .NET component. Structures used by List & Label are available in standard .NET notation as event arguments. These are always resembled by a class derived from

EventArgs. If a structure is provided to a callback with additional data, this information is available through appropriate properties with the corresponding .NET data types. The original structure is always offered through the CallbackData property.

You will find an extensive description of the available events in the supplied online help for the .NET component.

5.8. Evaluating each printout

List & Label offers an extensive formula parser, which can also easily be used in custom applications. This way you can offer your customer an efficient tool for customization.

The class ExpressionEvaluator offers the parser. You can pass either the instance of a class implemented by the IDictionary sections or the desired variables through the property of the same name to the constructor. Here, the usual options of the VariableCollection are at your disposal.

The formula is passed to the Evaluate method, which returns the result as an object. Now, you can carry out an easy type conversion according to the data type.

The sample listing shows how to call the class. The current date is passed as a variable. List & Label's designer function, AddWeeks, adds two weeks to this date. The result is displayed in a MessageBox.

```
ExpressionEvaluator eval = new ExpressionEvaluator();
eval.Variables.Add("date", DateTime.Now);
DateTime newDate = (DateTime) eval.Evaluate("AddWeeks(date, 2)");
MessageBox.Show(newDate.ToString());
```

If you want to carry out evaluation on the basis of an existing instance of the ListLabel component, simply use the EvaluateExpression method. Through a Boolean parameter, you can indicate whether only the registered variables are to be evaluated or the fields as well.

5.9. Working together with .NET print classes

The .NET framework offers many classes, components and controls for printing and displaying preview windows. In this connection, special mention is to be made of the PrintDocument component and PrintPreviewControl control, which can be used to display a preview in a form.

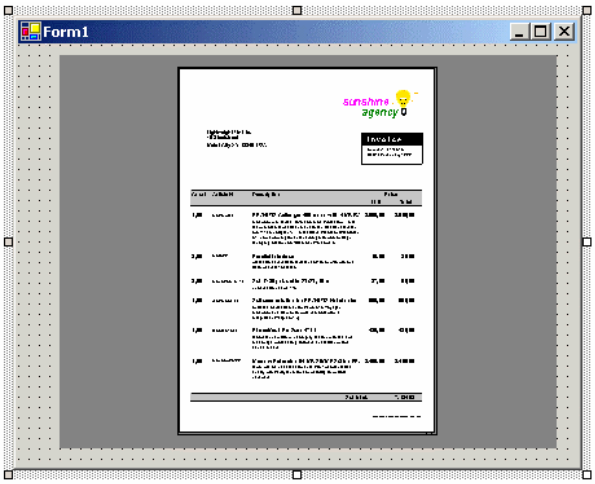
List & Label can work together with the classes offered. A derivation of the class PrintDocument, the class ListLabelDocument, is offered. This represents a created print project. Internally, the class is based on a preview file in the LL format.

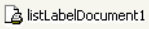
5.9.1. Displaying an existing LL file in a PrintPreviewControl

In order to display an existing List & Label preview file (extension .LL) in a .NET PrintPreviewControl, the ListLabelDocument component is first placed in the form.

Through the PreviewFilename property, you can indicate the file name of the file to be displayed.

In the second step, place the PrintPreviewControl or even a PrintPreviewDialog in your form. Through this control, the preview can be directly displayed in the form. The document to be displayed should be assigned only through the property Document.





The illustration shows the application of the control in connection with the ListLabelDocument component.

5.9.2. Printing and subsequent display in a PrintPreviewControl

The represented .NET classes are supported even while printing. You can assign an existing PrintPreviewControl to List & Label with the property PreviewControl. Similarly, PreviewControl is displayed as the print destination. This is also possible in case of a databound application through the property AutoDestination. After creation, a preview file is automatically displayed in the selected control.

The specialized supplied List & Label PreviewControl (class ListLabelPreviewControl) is more powerful than the .NET-PreviewControl. This PreviewControl is also selectable and allows e.g. a direct export in different formats.

5.10. Working with preview files

The List & Label Storage API provides access to the LL preview files. You can query general information from individual pages, place multiple files together, and store user data. In the case of .NET components, the API options are accessed with two classes PreviewFile and PreviewPage, each of which represent a complete preview file or an individual page of a file.

5.10.1. Calling a preview file

You can pass the file name of the preview file to be opened to the constructor of the class Previewfile. General information on the file is now available with several properties.

Property	Data type	Description
Handle	IntPtr	Preview file handle
FileVersion	int	File version
APIVersion	int	API version
Units	LlUnits	Provides the measurement unit
EmfResolution	int	Triggering internal Metafiles
PrinterCount	int	Number of printers
BoxType	int	Type of progress dialog to be displayed
Filename	string	File name
Count	int	Number of pages

Sample listing:

```
PreviewFile sf = new PreviewFile("c:\\test.ll");
int PageCount = sf.Count;
...
```

An opened file can be closed with the Close method. If files are to be deleted, use the DeleteFiles method. The file can also, of course, be printed. Certain overloads of the Print method allow the selection of one or multiple printers as well as that of the page area to be printed.

5.10.2. Appending multiple preview files

You can append multiple preview files together. In order to do this, you must first open the destination file. Since a write access is required, you must pass the second parameter, ReadOnly, as false. Subsequently, you can pass the file name of the file to be added to the Append method. Alternatively, you can also assign a handle to a preview file in the memory or to another instance of the PreviewFile.

```
PreviewFile sf = new PreviewFile("c:\\test.ll", false);
sf.Append(new PreviewFile("c:\\test2.ll"));
sf.Close();
```

5.10.3. Access to the pages of a preview file

The class indexer provides access to the each page of the preview file. A page is always represented by the PreviewPage class. Alternatively the class supports the IEnumerable interface so that you can cover the pages with a for each…in loop.

```
PreviewFile sf = new PreviewFile("c:\\test.ll");
foreach(PreviewPage pp in sf)
{
   Metafile mf = pp.GetMetafile();
   ...
}
```

Each page provides some information such as the name of the print job, the name of the printer, the set number of copies, and much more. It is also possible to query the page settings. For this, the property PageSettings returns an instance of the .NET framework class PageSettings.

With the GetMetafile method, it is possible to query the internally used Metafile. This provides direct access to the page content. The class ListLabelDocument described above uses this option to pass the preview file to a PrintPreviewControl. Alternatively, the GetBitmap method should be used if a Bitmap is required instead of a Metafile.

5.11. Using List & Label for Web reporting

An interesting option of the List & Label .NET component is Web reporting. In connection with ASP.NET, you can provide online and on-the-fly reports on the server and make them available to the user of your web application.

5.11.1. Preparations for (Web-)Serverside reporting

Web reporting requires a small number of preparations, described below. This is necessary for the example provided as well as for customized web applications. This description assumes that you have already created an application for a directory on the web server.

1. First, indicate an alias for the physical directory in the configuration of the Internet information services. Now, you can access the application through the given alias: http://localhost/<Alias>/
2. Explicitly provide full access control (Full Control) for the user account "ASPNET" of this directory at the Windows ACL (Access Control List) level. The control is necessary, as files have to be read and created while running the application.
3. As long as you have stored List & Label DLLs (for instance, cmll12.dll) in the Windows-System32 directory (standard case), the user account indicated requires at least the read rights for this directory. Alternatively, you can also copy the DLLs mentioned in the redist.txt file into the bin directory of the web application. Please note that these files are required in addition (!) to the .NET component listlabel12.dll, which should be primarily stored in the bin directory or the Global Assembly Cache.

4. Copy the webserver license file into the same directory where the cmll11.dll is located. You can create the license file with the aid of the tool ll11web.exe, which is part of the installation. You need to supply the serial number and the product key of your server / webserver license. You will find further information on server/webserver licensing in the license agreement for List & Label.

You can check your installation with the supplied tool ll12webcheck.exe. This application checks among other things the access rights and the installed printers. All required files are located in the directory "\llwebcheck". A detailed description of the functionality is also located in this directory.

5.11.2. Printer Driver and Web Server

The user account "ASPNET" works in the context of a Windows service. Generally, it does not have access to the printer installed on the system. However, this is necessary, as List & Label also requires a printer driver as reference while creating export and preview files. If access is not possible, you receive the error code –11 or a corresponding exception in .NET.

The Microsoft knowledge base offers an article Q184291 which is to be followed in order to install printers for the SYSTEM and ASPNET account. You can access the article through the indicated link. After successfully accepting the configuration, printing should be possible within the services.

http://support.microsoft.com/support/kb/articles/Q184/2/91.ASP

5.11.3. Debugging

On account of the systems architecture, debugging of web applications is not possible with the help of the DebWin2 debugger. Therefore, List & Label offers to alternatively create a log file. The following line activates this mode:

```
LL.Debug = LlDebug.Enabled | LlDebug.LogToFile;
```

Now, a file combit.log is placed in the Windows directory, where all calls to the List & Label API and all parameters are logged. If another file name is to be used, you can easily pass it to the component constructor:

```
ListLabel LL = new ListLabel(LlLanguage.Default, true, "c:\\debug.log");
```

Remark: Note that the user account "ASPNET" requires full control to the corresponding directory in order to create the log file. If no other path has been selected, full control of the Windows path is required for debugging the web application.

If you would like to start debugging later, just use the configuration file of the called application. You can, thus, provide a regular release version later, with the debug output.

```
<configuration>
<appSettings>
<add key="ListLabel DebugLogFilePath" value="c:\combit.log"/>
<add key="ListLabel EnableDebug" value="1"/>
```

```
</appSettings>
</configuration>
```

In this manner, you can activate debugging for web applications in machine.config/ web.config and for desktop applications in <exe>.config.

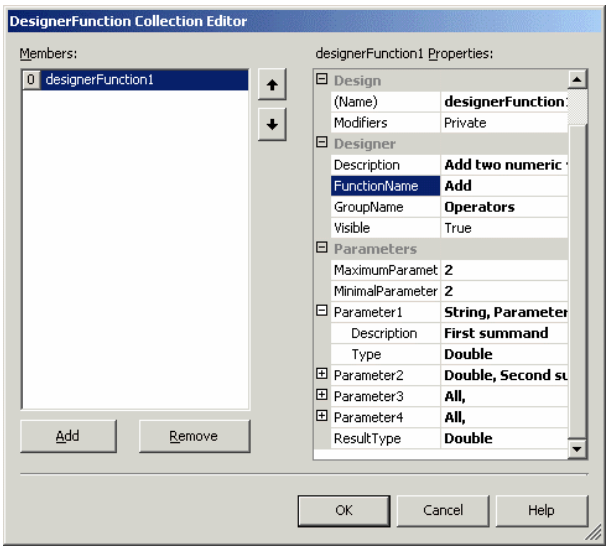
5.12. Enhancing the Designer

List & Label offers multiple options for enhancing the designer. Among others, the different events of the component, which, for instance, allow menu intervention and the offered functions, belong here. And there are even more options…

5.12.1. Adding designer functions

One of the important and powerful options of the designer is the formula assistant and the functions offered in it. With the help of the External$ function and the component's Evaluate event, a function can be handled separately by the program is offered to the user. Based on the .NET component, it is also possible to integrate completely customized functions into the designer.

To add a new function, open the properties window of the component at design time. Select the property DesignerFunctions and open the Editor using the "..." buttons. The picture shows the displayed Collection Editor.



In the dialog, you can add a new designer function via the "Add" button. A new component is automatically displayed and appears in the form tray. You can set the

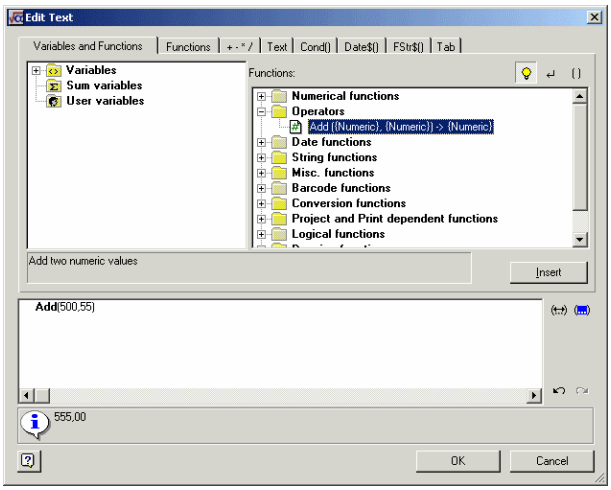
behavior of the new function in the designer via the properties window on the page to the right. The table shows the available properties.

Property		Data type	Description
FunctionName		string	The name of the designer function
Description		string	Additional description of the function for the formula assistant
GroupName		string	The group in the formula assistant in which the function is displayed
Visible		bool	Indicates whether or not the function is displayed in the assistant
MinimalParameters		int	The minimum number of parameters. Values between 0 and 4 are valid.
MaximumParameters		int	The maximum number of parameters. Here as well, values between 0 and 4 are valid. The value should be equal to or greater than the minimum number. A larger number results in optional parameters.
Parameter1 – 4		Designer Function Parameter	Each of the up to four parameters can be configured individually.
	Type	LIParamType	The data type of the parameter
	Description	string	Parameter description for the tooltip help in the designer
ResultType		LIParamType	The data type of the result

With the help of the properties, you can customize the new designer function. Finally, the event EvaluateFunction has to be handled in order to activate the function. You can gain access to the parameters indicated by the user through event arguments. Use the following lines in order to return the total of both parameters, for instance:

```
private void designerFunction1_EvaluateFunction(object sender,
 EvaluateFunctionEventArgs e)
{
   e.ResultValue = (int.Parse(e.Parameter1.ToString()) +
      int.Parse(e.Parameter2.ToString()));
}
```

The picture shows the application of an individual function Add in the formula assistant of the designer.



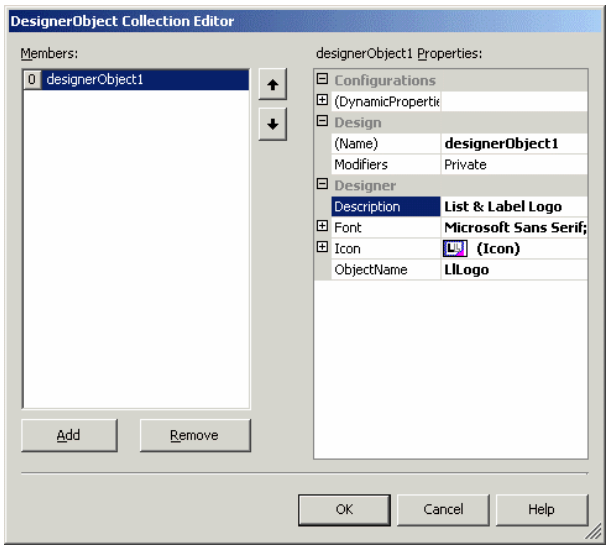
Two more events optionally allow you to further adapt the function. You can check the syntax with CheckFunctionSyntax. Here, you can check the data type of parameters and make sure that the parameters are, for instance, placed in a specific range. Via ParameterAutoComplete, it is possible to give different recommended values to the AutoComplete feature of the formula assistant. This, for instance, can be done as follows:

```
private void designerFunction1_ParameterAutoComplete(object sender,
   ParameterAutoCompleteEventArgs e)
{
   for(int i = 1; i<=20; i++)
      e.Values.Add(i.ToString());
}
```

5.12.2. Adding Designer objects

Similar to the enhancement of the designer functions, you can also define custom object types and register them with List & Label. The new objects are subsequently available to the user in the usual manner through the toolbar on the left and the menu.

Here, a special Collection Editor is also available. You can access it in the development environment with the "..." button of the property DesignerObjects of the previously placed List & Label .NET component. With the "Add" button, you can add a new object type.



Afterwards, you can set the properties of the new object. The table provides an overview.

Property	Data type	Description
ObjectName	String	The clear name of the object. This is used internally as an ID, which is, however, not shown to the user.
Description	String	The description is displayed in the designer. It may contain space characters, which should, however, not be more than 30 characters in length.
Icon	Icon	The object icon that is displayed in the toolbar and menu in the designer. It should concern a 16x16 Pixel size Icon with 16 colors.

The component offers three events. Firstly, the event CreateDesignerObject is triggered when the user creates a new object. If desired, you can show the initial dialog to the user. This can, for instance, be an assistant that makes it simple for the user to configure the new object. If you don't want to offer this, simply forego processing of the event.

The following lines show the display of a MessageBox as soon as the new object is placed in the workspace for the first time. You gain access to the designer window

through the event argument. The passed class supports the IWin32Window interface and can, as a result, be used for a MessageBox or another window as a parent.

```
private void designerObject1_CreateDesignerObject(object sender,
   CreateDesignerObjectEventArgs e)
{
   MessageBox.Show(e.DesignerWindow, "The new object is now displayed.");
}
```

The event EditDesignerObject is triggered, when the user double clicks on the newly added object or selects the item "Properties" from the context menu. Here as well, you can gain access to the designer window through event arguments, and a separate dialog can be displayed.

In the listing, you see the display of a file selection dialog. Here, you can choose a graphic to be displayed in the designer. The dialog is displayed by double-clicking on the object.

```
private void designerObject1_EditDesignerObject(object sender,
 EditDesignerObjectEventArgs e)
{
   DesignerObject desobj = (DesignerObject) sender;

   OpenFileDialog dialog = new OpenFileDialog();
   dialog.Filter = "JPEG files (*.jpg)|*.jpg|All files (*.*)|*.*";

   if(desobj.ObjectProperties.Contains("imagefile"))
      dialog.FileName = desobj.ObjectProperties["imagefile"].ToString();

   if(dialog.ShowDialog(e.DesignerWindow) == DialogResult.OK)
      {
      desobj.ObjectProperties["imagefile"] = dialog.FileName;
      e.HasChanged = true;
      }
}
```

When the user selects a file, the file name is stored in a dictionary. This dictionary (IDictionary) is offered by the component with the property ObjectProperties. It is important to store the data here and not in any other place, as it is (and should be) stored within the project file. For this purpose, the dictionary is serialized at the time of storing the project in the designer and de-serialized once again at the time of opening. You should be able to serialize all data stored in the dictionary. For this purpose, this data should at least be provided with the attribute serializable. The standard value types of the Common Language Runtime (string, int, etc.) are already serializable originally and can, thus, be used without further concern. A BinaryFormatter is used internally, the MIME coded result of which is stored in the project file.

List & Label should be informed if the user has made changes to the object or its properties within the EditDesignerObject event. This is necessary so that a save

query is displayed, if required, at the time of closing the designer. In case of a change, the property HasChanged of the event argument is to be simply set to true.

After the user has edited the object, you are asked by List & Label to draw the object. The event DrawDesignerObject is triggered for this purpose. A Graphics object and the rectangle of the object are passed through EventArguments. Now, you can draw in the work area with the known GDI + methods. While doing so, access to the underlying object properties is naturally possible and useful. The sample listing shows the presentation of graphic file selected above.

```
private void designerObject1_DrawDesignerObject(object sender,
   DrawDesignerObjectEventArgs e)
{
   DesignerObject desobj = (DesignerObject) sender;
   if(desobj.ObjectProperties.Contains("imagefile"))
      {
      string imagefile = desobj.ObjectProperties["imagefile"].ToString();
      e.Graphics.DrawImage(new Bitmap(imagefile), e.ClipRectangle);
      }
}
```

Remark: Due to a minor inconsistency, conversion of the font size should be carried out manually at the time of printing the text using the Graphics.DrawString, if necessary. The .NET framework calculates it internally as 1/10 inch. If millimeter (1/10) has been indicated as the metric unit (property Units), the size should be multiplied manually by 2.54:

```
e.Graphics.DrawString("Hello World", new Font("Arial", 20F * 2.54F),
   new SolidBrush(Color.Black), e.ClipRectangle);
```

5.13. Other differences

The following paragraphs describe several other differences between the List & Label .NET component and the known OCX and VCL controls.

5.13.1. Access to the API functions of List & Label

List & Label essentially consists of an ANSI DLL with an API. The manual describes the various available options, which are, to a large extent, covered by the .NET component's corresponding techniques. However, if you want further intervention and want to create a custom print loop, for instance, you require the complete range of List & Label APIs.

With the property Core of the component, you can gain access to the instance of a special API class. This offers the indicated functions as methods. The call is carried out without job handle and on the basis of the standard .NET data types. Apart from this, the majority of the calls correspond to the current description in the manual. Example:

```
LL.Core.LlPrintResetProjectState()
```

5.13.2. LlSetPrinterInPrinterFile

The LlSetPrinterInPrinterFile method is used to programmatically set the printer to be used for a project. For this purpose, a DEVMODE structure can be passed with the required information. Thanks to multiple overloads, it is also possible to assign a class instance of the PrinterSettings class from the .NET framework in case of the .NET component.

```
PrinterSettings settings = new PrinterSettings();
settings.PrinterName = PrinterSettings.InstalledPrinters[0];
LL.Core.LlSetPrinterInPrinterFile(LlProject.List, "c:\\test.lst", settings);
```

6. Programming Interface

6.1. Dynamic Link Libraries

6.1.1. Basics

A DLL (Dynamic Link Library) is a collection of routines in a file which is loaded on demand by the Windows kernel. Most windows-API-function-calls are contained in DLLs, for example KERNEL32.DLL, USER32.DLL, GDI32.DLL and many others (DLLs don't have to use the file extension "DLL"!). The language extensions for List & Label are also DLLs.

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 thus saving memory. Another advantage is that whenever an improved version of the DLL is available, the old one can be replaced. From then on all programs using the DLL automatically take advantage of the new DLL without having to be recompiled.

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

List & Label uses various DLLs for specific jobs.

6.1.2. Installation 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 program file 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 onto a path, on 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 Windows 98SE/ME and Windows 2000/XP.

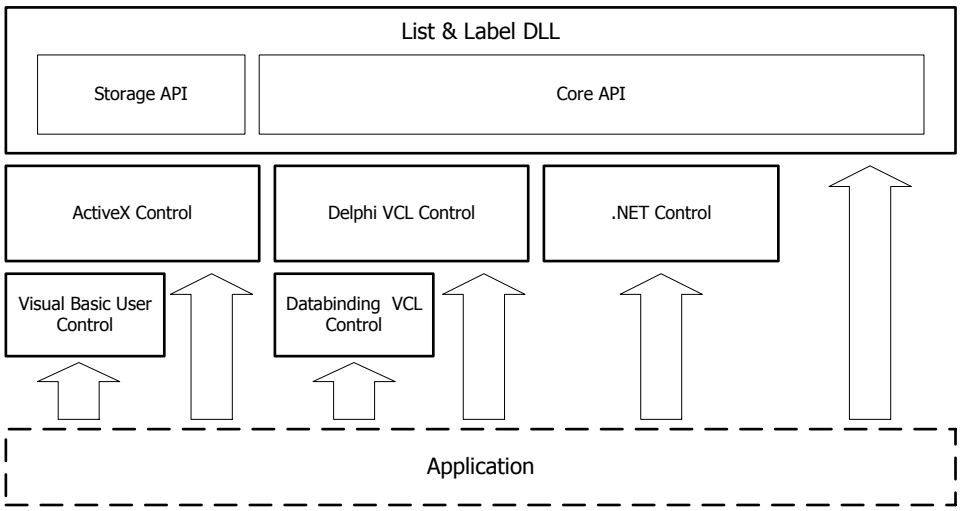
6.1.3. Linking of DLL routines

DLL routines can be called from various languages, like C, C++ , Pascal/Delphi, Visual Basic, VBA and others.

To ensure these calls do not cause a linker error the functions have to be declared to the linker (if the language has one, else the interpreter or runtime module).

The languages C, Pascal and Basic differ from each other. An extreme example are macro languages like Excel in which the declaration is relatively complicated.

Basically different components are available which are wrapper around the real DLL-functions. Therefore is the functional range substantially the same independent from the kind of used component.



6.1.4. Linking to C/C++ Programs with Import-Libraries

In your package you will find files called CMLL12.LIB and CULL12.LIB (for the Unicode DLL), the import-libraries. These files have to be declared to the linker. This enables the linker to connect the calls in the application to the DLL.

In order to pass the prototype declaration of the routines to the C-compiler the header file CMBTLL12.H has to be included after the "#include <windows.h>" statement as Windows variables are used in the declarations. Please note the '#ifdef' statements to choose the correct declarations for your Windows version. We recommend the declaration of STRICT to allow the compiler a more restrictive declaration check. Please check the information in the WIN 32 SDK.

You can also create the import library with the program IMPLIB which is usually enclosed in your SDK or possibly the compiler. Some manufacturers support this (Watcom/Sybase, Borland), others don't (Microsoft).

Also a dynamic loading of the DLL's is possible – you will find a description in the online knowledgebase under http://support.combit.net.

6.1.5. Linking to Delphi Programs

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

6.1.6. Linking to C++Builder Programs

Please refer to the online help for the VCL component's use.6.1.6.1.

6.1.7. Linking to Visual Basic-Programs

To use the OCX control add the control just as normal other OCX controls. If you want to additionally use the databinding control, add the file cmll12vb.ctl. Another control appears in the tool bar. Please refer to the OCX online help.

If you want to use direct DLL access and no OCX control, add the file cmll12.bas. Here you will find the necessary declarations.

6.1.8. Using the .NET assembly

Please refer to the documentation in the .NET directory of your List & Label installation and the corresponding chapter in this manual. An extensive online help describes in addition the characteristics of the component.

6.1.9. Using in 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 or OCX controls.

In case that your programming language is not contained you can create a declaration file on your own by using the chapter data types together with the according help of your programming language. In case of doubt you can contact our support.

6.1.10. Important Remarks to the function-parameters of DLLs

The return of strings from the DLL to the application is always defined by the return of a pointer to a memory area and as a further parameter an integer value for the length. The buffer size (in Bytes for the normal API and in characters for the Unicode API) 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.

Please note that with Delphi the routines require "\0"-terminated ANSIZ-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 to 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. Windows will reward you for this failure with GPFs.

In addition, it must be noted that with Visual Basic some functions cannot be supported; these are, however, not normally required. You can find out which functions these are in the corresponding chapters. If the parameter value is NULL © 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, it offers support of the native String data types.

6.1.11. Differences in data types of C/C++ , Delphi and VB

In the following table you can see the corresponding data types of the program development environments mentioned above.

C/C++	Delphi	Visual Basic
VOID	void	-
PVOID	pChar	String, As Any
BOOL	bool	Integer
CHAR8	char	Integer
CHAR16	WCHAR	Integer
UCHAR8	byte	Integer
TCHAR	TCHAR	Integer
INT8	shortint	Integer
UINT8	byte	Integer
INT16	smallint	Integer
UINT16	word	Integer
INT32	longint	Long
UINT32	longint	Long
INT	integer	Long
UINT	cardinal	Long
LONG	longint	Long
ULONG	longint	Long
DLGPROC	tFarProc	-
FARPROC	tFarProc	-
HWND	HWND	Long
HDC	HDC	Long
HCRS	hCursor	Long
HANDLE	tHandle	Long
HMENU	tHandle	Long
HFONT	tHandle	Long
PCHAR	pCHAR	String
TCHAR	TCHAR	-
PCHAR8	pCHAR	String
PCHAR16	pWCHAR	-
PTCHAR	pTCHAR	String/-
POINT	tPoint	POINT
RECT	tRect	RECT
COLORREF	tColorref	COLORREF
MSG	tMsg	MSG
FLOAT4	single	single
BSTR		String(ByVal)
BSTR*		String(ByRef)

6.1.12. Hexadecimal Numbers

Hexadecimal numbers are based on the number 16. Throughout this manual you will find that these numbers begin with '0x', the typical prefix for C. With Pascal this is replaced by '$'; with Basic by '&H'.

C:	0x1A2B3C4D
Pascal:	$1A2B3C4D
Basic:	&H1A2B3C4D

Occasionally, characters like '\xaa' can be found. That is, the C/C++ syntax for a character with the hexadecimal code AA, in decimal 170. With Pascal it is chr($AA); with Basic chr$(&HAA).

6.2. Hints on Parameters

- All strings are to be passed in ANSI-code and zero-terminated (possibly use the WIN-API-function OemToAnsi), in the following called ANSIZ.
- 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".
- With buffers the number of bytes corresponding to the passed length parameter must be reserved, otherwise a "GPF" will be unavoidable or the program will not operate correctly. LL_ERR_BUFFERTOOSMALL will be returned in these cases.
- The parameters, as far as possible, are checked for correctness. While developing a program it is therefore worthwhile switching on the debug mode (see LlSetDebug()) and checking the return values. Later you can switch off the parameter check using LL_OPTION_NOPARAMETERCHECK.

6.3. 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, signalizes an error which shows the reason with the corresponding error constants.

6.4. Variables/Fields and their Values

The hints below are valid for variables and for fields, for the latter "variable" is logically replaced with "field".

6.4.1. Characters Used in Variable Names

In variable names, the only characters that should be used are 'A' through 'Z', 'a' through 'z', '.', '_' and umlaut characters.

The names of different variables should be different, so please avoid giving variables the same name but different type (e.g. define once as *LL_NUMERIC* and once as *LL_TEXT*).

If the variable names consist of multibyte characters (for example Japanese names), please set *LL_OPTION_XLATVARNAMES* to FALSE.

Attention: You can decide whether variables are case sensitive or not (*LL_OPTION_VARSCASESENSITIVE*). Even if they are case sensitive, they must not differ only in the letter cases!

6.4.2. Characters Used in the Contents

All printable ANSI-codes including the control characters named below.

Please be careful if the old expression mode is being used and when the contents look like a field definition:

- if *LlDefineVariableExt('Test', '<Name>', LL_TEXT, NULL)* only '<Name>' would be replaced - if possible - by the contents of the corresponding field,
- with *LlDefineVariableExt('Test', '<Test>', LL_TEXT, NULL)* an endless loop is formed, see hints for *LlDefineVariable()*), or
- if the contents contain a requirement structure (only the old mode).

6.5. Rounding

Please read the following important notes to ensure that you do not run into inconsistencies to the data from your database. Sum variables are not rounded, so if you are using not only integers (i.e. invoices) we suggest you to use a rounding function, or (better) do not use multiplication or division for sum variables.

7. Functions and Operators in the designer

The designer functions can be found in the designer online help.

8. Usage in International Environments

Supporting different international environments is more or less complex, as you – if you have already done some research on that subject – might know. If you are not familiar with this topic, we suggest you to read special literature, for example: Nadine Kano: "Developing International Software" MS Press, ISBN 1-55615-840-8.

Most of the western countries have the same codepage (1252), which is the base for List & Label, so if you only plan to use this codepage and no far-eastern character set, you can skip this chapter.

Please keep in mind that the following chapter describes the List & Label functionality, not the user interface. You can edit and print a Japanese project in an English Windows (if the Japanese NLS has been installed)and vice versa – we are talking of formula evaluation and output here. The language of the user interface is independent from this.

8.1. Western Codepages (SBCS[1])

If you plan to use List & Label projects for SBCS codepages like 1257 (Baltic rim), 1251 (Cyrillic/Slavic), 1253 (Greek), 1254 (Latin 5: Turkish), 1250 (Latin 2), you should think about setting some options in advance. List & Label uses special characters that are usually not used:

- *LL_OPTION_TABREPRESENTATIONCODE / LL_CHAR_TAB* is the character used as tab stop in expressions and texts. Default is "÷" (code 247).
- *LL_OPTION_RETREPRESENTATIONCODE / LL_CHAR_NEWLINE*: is the character that represents a forced line break. Default is "¶" (code 182)
- *LL_OPTION_EXPRSEPREPRESENTATIONCODE / LL_CHAR_EXPRSEP* separates the lines of the multi-line function wizard, but is usually not visual anywhere. Default is "€" (code 164).

These codes might mean some meaningful text character in a codepage, so the respective representation codes might have to be changed. This has to be done before creation and editing of a project, of course.

For using List & Label with codepage 1253 (Greek), you must change LL_CHAR_TAB at least.

Some other representation codes have been given new values:

- *LL_OPTION_TEXTQUOTEREPRESENTATIONCODE / LL_CHAR_TEXTQUOTE*: Embraces texts in the project file. Will be never visible, and it must be a character that can never appear in a text. Since version 7 the default character has code 1. Old project files with the old value 168 can still be read.
- *LL_OPTION_PHANTOMSPACEREPRESENTATIONCODE / LL_CHAR_PHANTOMSPACE*: Default is now 2.

[1] SBCS = single-byte character set

- *LL_OPTION_LOCKNEXTCHARREPRESENTATIONCODE / LL_CHAR_LOCK*: Default is now the character code 3.

Codes 1 to 19 are reserved for List & Label.

8.2. Far East Code Pages (MBCS[2]/DBCS[3])

Far eastern languages need more than the 224 possible characters in an SBCS character set, so an extension was needed – the multi-byte character sets (MBCS) have been invented. Windows uses a special variant of them – the double byte character set (DBCS). The old ASCII characters (range below character code 128) are still one byte long, but most of the Far East characters use two bytes, one "lead byte" and one "trail byte". One of the results is that 10 characters of Japanese text need 10 to 20 bytes to be stored. This needs to be taken into account if a buffer will be passed to List & Label.

You must ensure that the representation codes do not lie in the range of the "lead bytes". Often you have to change all representation codes for *LL_CHAR_TAB*, *LL_CHAR_NEWLINE*, *LL_CHAR_EXPRSEP*:

Codepage	RET[4]	TAB[5]	EXPRSEP[6]
936 (simplified Chinese)	X	X	X
950 (trad. Chinese)	o.k.	o.k.	X
932 (Japanese)	X	X	X
949 (Korean Wansung)	X	X	X

(X=needs to be changed)

One additional point to be noted is that *LL_OPTION_XLATVARNAMES* should be set to FALSE for correct display.

In the DBCS version, String-functions like *left$()*, *mid$()*, *right$()* and *len()* work character based, in the SBCS version byte based.

8.3. Unicode

To have a simpler solution to the very complex DBCS string management, a character set has been created in which every character uses two bytes. This is enough to represent all important characters of all languages on earth simultaneously. The standard interpretation between codes and characters is called "Unicode" in Windows.

The disadvantage is that the development environment must be able to manage Unicode – not all can. And a different version of the List & Label DLL needs to be linked to.

---

2 MBCS = Multi-Byte Character Set

3 DBCS = Double Byte Character Set

4 recommended code: 20

5 recommended code: 21

6 recommended code: 22

It would be much more advantageous if the Windows 9x family would be able to support a Unicode application, but nearly all of the Unicode APIs in Windows 9x are functionless stubs.

Please note that the export modules do not fully support Unicode. You have to set the target codepage using LL_OPTION_CODEPAGE and – for the PDF export – the charset of the used fonts (ex. "Japanese"). A mixture of characters from different codepages is not possible ex. for HTML, RTF and XLS export. The image exporters and preview format offer full Unicode support.

The unicode version supports the RTF Control 2 and higher.

8.4. Implementation in List & Label

So List & Label and its core DLLs are offered in two flavors: one SBCS version (optionally extensible to support DBCS) and a Unicode version. The latter starts with "cu" in the file name, the former "cm". The DBCS version runs on Windows 9x and NT 4 and greater, while the Unicode version requires NT 4 or Windows 2000 (also NT).

All APIs having string parameters are implemented twice, as usual for Windows DLLs: one "A" version for DBCS parameters and one "W" version for Unicode. This is being hidden in the declaration file by macros, so usually you need not worry. But it is possible for a Unicode application to easily use the SBCS DLL, or for a SBCS application to use the Unicode DLL.

Of course that comfort does not come without cost: the "A" API of SBCS and "W" API of DBCS need to convert the parameters to the "correct" form, which means some time penalty. But if you use Windows NT or 2000, the Unicode DLL of List & Label calls the Unicode API of Windows, which is faster than the Windows "A" API.

8.5. Exchanging project files with different character sets

Exchanging project files does not create any problems when the representation codes of the systems are set to the same value. This may lead to the same visual representation of different codes (all codes < 32 are shown as bar or dot). In a formula, you can replace RET by "chr$(13)" and TAB by "chr$(9)" to minimize these problems.

More difficult are the chevrons («») used in RTF objects as braces for expressions. They are not usable in DBCS systems as they are letters, characters or lead bytes in those environments.

The DBCS version of List & Label thus has an automatism to guarantee the highest compatibility:

When running in an DBCS environment, the DBCS version defaults the formula markers to codes 4 and 6. If a new project is created, these values will be stored in the project file, so that they will be consistent every time the project is loaded. The

disadvantage is – again – a visual equality of opening and closing characters (formula start and end), but this is negligible. If an existing project is loaded, the values stored in it are used – if they are not in there (a pre-List & Label 7 project file) the old chevron values 0xab and 0xbb are being used.

As a result, you should make sure that the computer on which you are creating projects that might be used in DBCS environments with a DBCS character set is switched to a multibyte code page using *LL_OPTION_CODEPAGE* before a new project is created, thus storing chevrons as the compatible codes 4 and 6! For *LL_OPTION_CODEPAGE*, the corresponding NLS (National Language Support) for that code page must be installed on this computer.

8.6. Word Wrap

Word wrapping is automatically applied using the character set (script) of the text to be written. Supported word wrap rules are (in addition to the Western word wrapping) the symbol wrap (single character), Chinese (traditional and simplified), Japanese (kinsoku) and korean (Wansung, geumchik rule).

8.7. Localizing dates, currencies and numbers

Several of the functions which can be used in a List & Label formula can be localized to the current or a specific locale: *Date$()*, *LocCurr$()*, *LocCurrL$()*, *LocNumber$()*, *LocDate$()*, *LocTime$()* have an optional parameter which defines the locale to be used. If the parameter is omitted, the locale given by *LL_OPTION_LCID* (which defaults to the user locale) is used.

The *Locale$()* function can be used to query the system for all aspects of localization, for example for conditions.

In addition, the values for the thousands separator and the decimal point can be set as option string if they need to be different than the one set by *LL_OPTION_LCID* or the current locale's settings.

9. Callbacks and Notifications

Attention: this chapter is only interesting for applications that do not use OCX or VCL events. Normally it can be omitted.

9.1. Task

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).

9.1.1. Implementation in OCX or VCL

In these controls, the callbacks are implemented as events. Details can be found in the online help of the respective control type.

9.1.2. Implementation using the DLL

All this is possible if you implement one of the following code:

- 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. 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.

Warning: Callbacks are a very powerful method of handling functions. They are therefore easy to misuse and may be a stumbling stone for inexperienced windows-programmers.

9.2. User Objects

As List & Label cannot consider 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 are using the .NET component, user objects are very easy. Please refer to chapter 5.13.

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 analogous 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 done.

The same can be done for table fields so that the user has the capability of including an user object in his table:

```
LlDefineFieldExt(hJob, <Name>, <Content>, LL_DRAWING_USEROBJ, NULL);
```

For variables, but not for fields, there is also the possibility 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 can not be inserted in tables. Only non-editable user objects can be used as table field.)

If the user now presses the "edit"-button in the object's dialog 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.

Please note the references of article Q99109 in the MS Knowledge Base when you want to print DDBs (device dependent bitmaps) in callback objects: transform the DDB first of all to a DIB (device independent bitmap) if it is not compatible with the printer DC, otherwise an irregular behavior of the printer driver is pre-programmed.

9.3. Definition of a Callback Routine

A callback routine is defined like a normal Windows callback. For special questions like compiler switches please refer to your compiler manual.

The general form of a callback is in C

```
LRESULT CALLBACK _extern LLCallback(INT nMsg, LONG lParam, DWORD
lUserParam);
```

and in Delphi

```
function LLCallback(nMsg: integer; lParam: longint, lUserParam:
longint) :
            longint; external;
```

You can pass the routine pointer immediately:

```
LlSetNotificationCallback(hJob, LLCallback);
```

From now on your routine will be called by List & Label if it's necessary.

At the end of the program it is important that the callback is set to NULL, otherwise your system will raise a GPF.

```
LlSetNotificationCallback(hJob, NULL);
```

9.4. 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* to draw the background of a table field.

Depending on the function which your program has to (or is allowed to) carry out, 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, LONG lParam, LONG 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. If not 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.

9.5. Passing Data by Messages

Every message has three parameters: nMsg, wParam and lParam in the following definition of your message callback (it's 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()*. Should the default setting not suit your taste (per definition a unique value) another can be chosen with *LlSetNotificationMessage()*.

wParam again is our function *index* and *lParam* points to an intermediate structure of the type scLlCallback:

```
struct scLlCallback
{
   int    _nSize;
   LPARAM _lParam;
   LRESULT  _lResult;
   LPARAM _lUserParameter;
}
```

In this structure there the necessary *_lParam* (as parameter value) and *_lResult* (as return value) are stored.

```
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.

9.6. Two Device Contexts?

Yes... Unfortunately. List & Label of versions before 7 required it. But since List & Label is now 32 bit only, we don't require it any more. For compatibility reasons, we did not change the structures.

Both DCs are identical now.

Hints for Using GDI Objects

If you select a GDI object in this DC or undertake 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.

9.7. Overview

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 a *scLIUserObj* structure:

_nSize: Size of the structure, *sizeof(scLIUserObj)*

_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 ANSIZ-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 (*LIPrint()*, etc.))! Functions like *LIPrintGetCurrentPage()*, *LIPrintGetOption()* or *LIPrintEnableObject()* are allowed.

See: Hints for the usage 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 a *scLIEditUserObj* structure:

_nSize: Size of the structure, *sizeof(scLIEditUserObj)*

_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 *LIDefineVariableExt()*.

_lpPtr: *lpPtr* of the variable assigned to the object. This refers to the 5th parameter of the call *LIDefineVariableExt()*.

_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 being 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 usage 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 is adopted by List & Label.

_lpszParameter points to a ANSIZ 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 of 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: 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 en- or disable menu items formerly 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 a *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 *LIExprError()*. 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 (16 KBytes incl. termination for the return value, 128 Bytes incl. zero-termination for the error string).

LL_CMND_GETVIEWERBUTTONSTATE

Task:

Using this callback, List & Label asks the application about button states of the real data preview's toolbar buttons.

Activation:

Always activated

Parameters:

HIWORD(lParam) = Toolbutton 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 real data 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 one.

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_NTFY_EXPRERROR

Task:

Allows the host application to show if an expression error is found when the project is loaded.

Parameters:

lParam points to the error string that the wizard would display.

Return Value (_lResult):

ignored, always 0

LL_CMND_HOSTPRINTER

Task:

Allows the user to define one (or two) device contexts for the printer, which then are used by List & Label instead of its own printer management

Activation:

```
LlSetOption(hJob, LL_OPTION_USEHOSTPRINTER, 1)
```

Parameters:

lParam is a pointer to a struct of type scLIPrinter:

_nSize: sizeof of the structure, sizeof(scLIPrinter)

_bFirst: BOOL, indicates whether the printer for page 1 or the following pages is meant

_nCmd: subcommand, see below

_hWnd: handle of the parent window, needed for *LL_PRN_EDIT*

_hDC: DC-handle for the functions *LL_PRN_CREATE_DC*, *LL_PRN_DELETE_DC* and *LL_PRN_RESET_DC*

_nOrientation: printer paper orientation (*DMORIENT_HORIZONTAL, DMORIENT_LANDSCAPE*), used for *LL_PRN_SET_ORIENTATION* and *LL_PRN_GET_ORIENTATION*

_bPhysPage: BOOL, indication whether the physical or the printable area is to be used.

_nBufSize, _pszBuffer: points to memory where the requested texts can be stored: used for *LL_PRN_GET_DEVICENAME*, *LL_PRN_GET_DRIVERNAME* and *LL_PRN_GET_PORTNAME*

_nUniqueNumber: unique number for this printer instance. List & Label might request some printers several times, so this ID can be used to find out which printer is meant.

_nUniqueNumberCompare: the unique number of the printer to be compared with the current one (in case of a *LL_PRN_COMPARE_PRINTER* command, you need to return whether the printers in nUniqueNumber and nUniqueNumberCompare are the same).

_nPaperFormat: paper format (see DEVMODE.dmPaperSize)

_xPaperSize: paper size in mm/10 (horizontal)

_yPaperSize: paper size in mm/10 (vertical)

Return value:

depends on the subcommand in *_nCmd*:

Value	Meaning
LL_PRN_CREATE_DC	the printer DC shall be created and stored in *_hDC*. No change of the return value.
LL_PRN_DELETE_DC	the printer DC (in *_hDC*) can be destroyed. No change of the return value.
LL_PRN_SET_ORIENTATION	the paper orientation shall be set to the orientation given in *_nOrientation*. The printer DC is stored in *_hDC*. No change of the return value.
LL_PRN_GET_ORIENTATION	the current paper orientation should be stored in *_nOrientation* as *DMORIENT_PORTRAIT* or *DMORIENT_LANDSCAPE*. No change of the return value.
LL_PRN_EDIT	reserved (not yet used)
LL_PRN_GET_DEVICENAME	copy the device name of the printer into *_pszBuffer* (optional). No change of the return value.
LL_PRN_GET_DRIVERNAME	copy the driver name of the printer into *_pszBuffer* (optional). No change of the return value.
LL_PRN_GET_PORTNAME	copy the port name of the printer into *_pszBuffer* (optional). No change of the return value.

LL_PRN_RESET_DC	the printer DC shall be reset (see API function *ResetDC()*). The new value (usually the same in current Windows versions) should be stored into *_hDC*. No change of the return value.
LL_PRN_COMPARE_-PRINTER	List & Label needs this function to decide, whether the printer for the first page is the same as the one for the following pages. You must compare the printers (for example, *DEVMODE* structure members) with the IDs in *nUniqueNumber* and *nUniqueNumberCompare*.Return 1 for equal, 0 for different.
LL_PRN_GET_-PAPERFORMAT LL_PRN_SET_-PAPERFORMAT:	Shall return/set paper format (*_nPaperFormat*, *_xPaperSize* and *_yPaperSize*)

Hints:

This callback is rarely used and can be ignored for nearly all cases.

Example:

```
case LL_CMND_HOSTPRINTER:
{
   scLlPrinter* pPrn = scLlPrinter*)sc._lParam;
   if (pPrn->_nSize == sizeof(scLlPrinter))
   {
      switch (pPrn->_nCmd)
      {
         case LL_PRN_CREATE_DC:
         UtilDebug("LL_PRN_CREATE_DC(%d)\n", pPrn->_bFirst);
            pPrn->_hDC = CreateDC("PSCRIPT.DRV", "x.ps",
               "\\\\srv\\pr", NULL);
            break;
         case LL_PRN_DELETE_DC:
            UtilDebug("LL_PRN_DELETE_DC(%d)\n", pPrn->_bFirst);
            DeleteDC(pPrn->_hDC);
            break;
         case LL_PRN_SET_ORIENTATION:
            UtilDebug("LL_PRN_SET_ORIENTATION","(%d, %d)\n",
              pPrn->_bFirst, pPrn->_nOrientation);
            // not in this trivial example
            break;
         case LL_PRN_GET_ORIENTATION:
            UtilDebug("LL_PRN_GET_ORIENTATION","(%d)\n",
              pPrn->_bFirst);
            // not in this trivial example
            break;
         case LL_PRN_GET_DEVICENAME:
            UtilDebug("LL_PRN_GET_DEVICENAME","(%d)\n",
              pPrn->_bFirst);
            chk_strncpy(pPrn->_pszBuffer, "TestDevice",
```

```
                  pPrn->_nBufSize);
               break;
            case LL_PRN_GET_DRIVERNAME:
               UtilDebug("LL_PRN_GET_DRIVERNAME", "(%d)\n",
                  pPrn->_bFirst);
               chk_strncpy(pPrn->_pszBuffer, "TestDriver",
                  pPrn->_nBufSize);
               break;
            case LL_PRN_GET_PORTNAME:
               UtilDebug("LL_PRN_GET_PORTNAME(%d)\n", pPrn->_bFirst);
               chk_strncpy(pPrn->_pszBuffer, "TestPort",
                  pPrn->_nBufSize);
               break;
            case LL_PRN_RESET_DC:
               UtilDebug("LL_PRN_RESET_DC(%d)\n", pPrn->_bFirst);
               // not in this trivial example
                  break;
            case LL_PRN_COMPARE_PRINTER:
               // the same printer for first and other pages
               sc._lReply = 1;
               // We would need a vector of printers
               // (using _nUniqueNumber) for different printers
               break;
         }
      }
   }
}
break;
// chk_strncpy() is a "corrected" strncpy-function,
// which makes sure that the
// destination is always '\0'-terminated.
// UtilDebug is an example function for Debugging Output
```

LL_CMND_CHANGE_DCPROPERTIES_CREATE

Task:

The host application can modify the printer DC.

Parameters:

lParam pointer to a structure of type scLIPrinter. The following fields have meaningful values:

_nSize: sizeof of the structure, sizeof(scLIPrinter)

_hDC: printer DC

Return Value:

ignored, always 0

Hints:

This callback is rarely used and can be ignored for nearly all cases. It is called when List & Label has created a printer DC (after *CreateDC()*).

LL_CMND_CHANGE_DCPROPERTIES_DOC

Task:

The host application can modify the printer DC.

Parameters:

Meaning of the Parameter lParam:

Pointer to a structure of type scLIPrinter. The following fields have meaningful values:

_nSize: sizeof of the structure, sizeof(scLIPrinter)

_hDC: printer DC

Return Value:

ignored, always 0

Hints:

This callback is rarely used and can be ignored for nearly all cases. It is called when List & Label has called *StartDoc()*.

_nPaperFormat is the Job-ID you can use to get information about the spooler progress.

LL_CMND_CHANGE_DCPROPERTIES_PAGE

Task:

The host application can modify the printer DC.

Parameters:

Meaning of the Parameter lParam:

Pointer to a structure of type scLIPrinter. The following fields have meaningful values:

_nSize: sizeof of the structure, sizeof(scLIPrinter)

_hDC: printer DC

Return Value:

ignored, always 0

Hints:

This callback is rarely used and can be ignored for nearly all cases. It is called when List & Label has called *StartPage()*.

LL_CMND_MODIFYMENU

Task:

Allows the application to modify List & Label's menu.

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 10000.

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.

This function allows many modifications to objects.

Activation:

```
LlSetOption(hJob, LL_OPTION_CALLBACKMASK, <other Flags> |
LL_CB_OBJECT);
```

Parameters:

lParam points to a scLlObject structure:

_nSize: Size of the structure, sizeof(scLlObject)

_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:

Usage: 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:

Usage: you can draw an individual background and/or shade as only then the true size of the object is 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 interesting, for example, for printing labels 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 a scLlPage-structure:

_nSize: Size of structure, sizeof(scLlPage)

_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 a scLlProject structure:

_nSize: Size of the structure, sizeof(scLlProject)

_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 filename 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, when 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 first of all draw the background then it draws the background again (if allowed) with the field pattern defined in the designer. In this way a kind of cooperation is possible between the programmer and the user.

Parameters:

lParam points to a 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

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 undertake 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 first of all your program draw the background then it draws the background when required again with the field background defined in the designer.

In this way a kind of cooperation between the programmer and the user is possible.

Parameters:

lParam points to a 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_FIELD_GROUP	Group header
LL_TABLE_-FIELD_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.

_bZebra: TRUE, when the user chooses the zebra-mode in the designer.

Return Value:

0

Hints:

In this callback no List & Label function may be called which will produce output (*LlPrint()*, 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 don't 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_NTFY_VIEWERBTNCLICKED

Task:

Notification that a button has been pressed in the real data preview.

Activation:

Always activated

Parameters:

lParam: Toolbutton 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_INFO_METER

Task:

Notification that a (possibly) lengthy operation takes place.

Activation:

Always activated

Parameters:

lParam contains an address to a struct 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 worked upon

_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 supervise a print job after it is passed to the spooler.

Parameter:

lParam contains an address of a struct of type *scLIPrintJobInfo*:

_nSize: size of the structure

_hLIJob: 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:

This function cannot be used with Windows 98SE or ME. The detail depth depends on the print spooler. Please note that all machines in the transmission chain (i.e. print server and client) need to run NT based operating systems (NT/Windows 2000/Windows XP).

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_FAILS_FILTER

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_FAILS_FILTER:
   bFails = TRUE;
   break;
```

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.

10. 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.

10.1. Basics

10.1.1. 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. Using LlPrintGetProjectParameter() the evaluated value is returned.

LL_PARAMETERFLAG_VALUE

> The parameter is a fixed value. Using LlPrintGetProjectParameter() this value is returned.

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 aren't saved to the project file and thus remain with their default values.

Note: you may not pass multiple parameters with the same name but different types!

10.1.2. Printing the project

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 possibly (see above) are saved to the preview file and can be extracted from there using LlStgsysGetJobOptionStringEx() you may save your own information persistently in this way. In the preview file, the parameters are saved with the prefix "ProjectParameter" before the actual name.

10.1.3. 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 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.MAIL.To	GLOBAL, PUBLIC [LL_OPTIONSTR_MAILTO]
LL.MAIL.CC	GLOBAL, PUBLIC [LL_OPTIONSTR_MAILTO_CC]
LL.MAIL.BCC	GLOBAL, PUBLIC

	[LL_OPTIONSTR_MAILTO_BCC]
LL.MAIL.From	GLOBAL, PUBLIC
LL.MAIL.ReplyTo	GLOBAL, PUBLIC
LL.MAIL.Subject	GLOBAL, PUBLIC [LL_OPTIONSTR_MAILTO_SUBJECT]
LL.MAIL.Provider	User definable. Possible values: empty (Use internal MAPI) "SMAPI" (use CMMX01.DLL) "XMAPI" (use CMMX01.DLL) "SMTP" (use CMMX01.DLL)
LL.MAIL.SMTP.ServerTimeout	User definable
LL.MAIL.SMTP.ServerAddress	User definable
LL.MAIL.SMTP.ServerPort	User definable
LL.MAIL.SMTP.ServerUser	User definable
LL.MAIL.SMTP.ServerPassword	User definable
LL.MAIL.SMTP.LogonName	User definable
LL.MAIL.SMTP.ProxyAddress	User definable
LL.MAIL.SMTP.ProxyPort	User definable
LL.MAIL.SMTP.ProxyUser	User definable
LL.MAIL.SMTP.ProxyPassword	User definable
LL.MAIL.SMTP.ProxyType	User definable
LL.MAIL.SMTP.LogonDomain	User definable

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 beautify the address:

FIRSTNAME + " " + NAME + " <" + EMAIL + ">"

The preview control automatically adopts the values of LL.FAX.* and LL.MAIL.*. Additionally, the values are passed to the export modules – these also take care to use the user defined contents.

Please note a change in the behavior of the export modules until version List & Label 9 here: 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.

10.1.4. Automatic storage of form data

If you are using the form elements it is possible to realize an automatic storage after completion of the preview with the project parameters. For this purpose you can use the following project parameters:

SaveAs.Format	Desired export format, e.g. "XML"
SaveAs.Filename	Output file name, e.g. "test.xml"
SaveAs.ShowDialog	With it the save dialog can be enabled ("1") resp. disabled ("0")
SaveAs.NoSaveQuery	Disabled the request whether the file should be saved after completion or not.

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);
```

11. Description of the API Functions

LlAddCtlSupport

Syntax:

```
INT LlAddCtlSupport (HWND hWnd, UINT32 nFlags, LPCTSTR lpszInifile);
```

Task:

Allows you to use combit dialog styles in your own application.

Parameter:

hWnd: Window-Handle of the program

nFlags: Multiple flags can be used by adding these constants:

Value	Meaning
LL_CTL_ADDTOSYSMENU	to extend the system menu of the program with a choice of a dialog style.
LL_CTL_ALSOCHILDREN	to assure that the child windows automatically use the same dialog styles as their parent.
LL_CTL_-CONVERTCONTROLS	Ensures that correct drawing is possible even with VB ("Thunder") Controls.

lpszInifile: Name of a valid registry key. Suggestion: leave empty ("") to use the default.

Return Value:

Error code

Hints:

Enables e.g. the server application to be provided with other (combit) dialog styles.

Example:

```
LlAddCtlSupport(hWndMain, LL_CTL_ALSOCHILDREN |
LL_CTL_CONVERTCONTROLS, "");
```

See also:

LlSetDlgboxMode

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)

Return Value:

Error code

Hints:

This API function can be used to create the sketches on-the-fly, 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 for designing and printing. The table is available in the Designer and List & Label can request it at print time.

Parameter:

hJob: List & Label Job-Handle

pszTableID: This value is the unique name of the table. It is returned by the *LlPrintDbGetCurrentTable()* function at print time. If you are passing an empty string or NULL the table buffer will be deleted.

pszDisplayName: This value is the 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:

See the hints in chapter "4. Working with the Report Container".

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDbAddTable(hJob, "", NULL);
LlDbAddTable(hJob, "Orders", NULL);
LlDbAddTable(hJob, "OrderDetails", NULL);
<... etc ...>
LlJobClose(hJob);
```

See also:

LIDbAddTableSortOrder, LIDbAddTableRelation, LIPrintDbGetCurrentTable, LIPrintDbGetCurrentTableSortOrder, LIPrintDbGetCurrentTableRelation

LIDbAddTableRelation

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 LIDbAddTable. List & Label doesn't directly distinguish between different relation types. You simply pass a relation and its ID and can query the current relation later at print time using *LIPrintDbGetCurrentTableRelation()*.

Parameter:

hJob: List & Label Job-Handle

pszTableID: ID of the child table. Must be identical to the ID passed in LIDbAddTable.

pszParentTableID: ID of the parent table. Must be identical to the ID passed in LIDbAddTable.

pszRelationID: Unique ID of the table relation. It is returned by LIPrintDbGetCurrentTableRelation at print time.

pszRelationDisplayName: This value is the name of the table relation as displayed in the designer and it is not saved into 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 "4. Working with the Report Container". Before using the call the parent- and child table must be passed with LIDbAddTable.

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:

LIDbAddTable, LIDbAddTableSortOrder, LIPrintDbGetCurrentTable, LIPrintDbGet-CurentTableSortOrder, LIPrintDbGetCurrentTableRelation

LIDbAddTableSortOrder

Syntax:

```
INT LlDbAddTableSortOrder(HLLJOB hJob, LPCTSTR pszTableID,
  LPCTSTR pszSortOrderID, LPCTSTR pszSortOrderDisplayName);
```

Task:

Defines available sort orders for the tables added via LIDbAddTable. Each sorting has its unique ID that can be queried using LIPrintDbGetCurrentTableSortOrder 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 LIDbAddTable.

pszSortOrderID: Unique ID of the table sort order. It is returned by LIPrintDbGetCurrentTableSortOrder at print time.

pszRelationDisplayName: This value is the 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 "4. Working with the Report Container". Before using the call the table must be passed with LIDbAddTable.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDbAddTable(hJob, "Orders", NULL);
LlDbAddTableSortOrder(hJob, "Orders", "Name ASC", "Name [+]");
<... etc ...>
LlJobClose(hJob);
```

See also:

LIDbAddTable, LIDbAddTableRelation, LIPrintDbGetCurrentTable, LIPrint-DbGetCurrent-TableSortOrder, LIPrintDbGetCurrentTableRelation

LIDbSetMasterTable

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 table 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 LIDbAddTable.

Return Value:

Error code

Hints:

See the hints in chapter "4. Working with the Report Container". Before using the table must be passed with LIDbAddTable.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDbAddTable(hJob, "Orders", NULL);
LlDbSetMasterTable(hJob, "Orders");
<... etc ...>
LlJobClose(hJob);
```

See also:

LIDbAddTable, LIDbAddTableRelation, LIPrintDbGetCurrentTable, LIPrint-DbGetCurrent-TableSortOrder, LIPrintDbGetCurrentTableRelation

LlDebugOutput

Syntax:

```
void LlDebugOutput(INT nIndent, LPCTSTR pszText);
```

Task:

Prints the text to the debug window of DEBWIN2.EXE 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 to trace 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)=12
  @LlJobClose(1)
...done
```

See also:

LlSetDebug, Debwin2.exe

LlDefineChartFieldExt

Syntax:

```
INT LlDefineChartFieldExt(HLLJOB hJob, LPCTSTR lpszName,
  LPCTSTR lpszCont, INT32 lPara, LPVOID lpPara);
```

Task:

Defines a chart field and its contents. Chart fields are needed for ex. for charts in table columns. Refer to the chart chapter in this manual for more details.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to an ANSIZ string with the name of the chart field

lpszCont: Pointer to an ANSIZ string with the contents of the chart field

lPara: Field type (*LL_TEXT*, *LL_NUMERIC*, ...)

lpPara: for future extensions, must be NULL.

Return Value:

Error code

Hints:

Mind the general hints in the section "Variables and Fields in List & Label".

See also:

LlDefineChartFieldStart, LlEnumGetFirstChartField, LlPrintDeclareChartRow

LlDefineChartFieldStart

Syntax:

```
void LlDefineChartFieldStart(HLLJOB hJob);
```

Task:

Empties List & Label's internal chart field buffer to delete old chart field definitions.

Parameter:

hJob: List & Label job handle

Return Value:

Error code

Hints:

The hints for *LlDefineVariableStart()* also apply to this function.

If the function *LlPrintIsChartFieldUsed()* is used in your application, *LlDefineChartFieldStart()* may not be used after the call to *LlPrint[WithBox]Start()*, else the "used" flag is reset and *LlPrintIsChartFieldUsed()* always returns FALSE.

See also:

LlDefineChartFieldExt, LlEnumGetFirstChartField, LlPrintDeclareChartRow

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 an ANSIZ string with the name of the field

lpszCont: Pointer to an ANSIZ string with the contents of the field

Return Value:

Error code

Hints:

Mind 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 record 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 record 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, INT32 lPara, LPVOID lpPara);
```

Task:

Defines a list/table field and its contents.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to an ANSIZ string with the name of the field

lpszCont: Pointer to an ANSIZ string with the contents of the field

lPara: Field type (*LL_TEXT*, *LL_NUMERIC*, ...) if required combined by 'or' with (see below)

lpPara: for future extensions, must be NULL.

Return Value:

Error code

Hints:

Mind the general hints in the section "Variables and Fields in List & Label".

LlDefineField(...) is identical to *LlDefineFieldExt(..., LL_TEXT, NULL)*.

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 suited 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, INT32 lPara, LPVOID lpPara);
```

Task:

Defines a list field and its (handle) contents.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to an ANSIZ string with the name of the field

hContents: handle to an object of type HMETAFILE, HENHMETAFILE (32 bit), HICON or HBITMAP

lPara: *LL_DRAWING_HMETA*, *LL_DRAWING_HEMETA* (32 bit), *LL_DRAWING_HICON* or *LL_DRAWING_HBITMAP*

lpPara: for further extensions, must be NULL.

Return Value:

Error code

Hints:

Mind 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.

List & Label predefines the fields listed in *LlDefineField()*.

The metafile handle must be valid as long as it is needed, that is during the whole layout definition or until after *LlPrintFields()* respectively *LlPrint()* has finished.

After usage 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_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(,
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", "Smith");
LlDefineField(hJob, "forename", "George");
LlDefineFieldExt(hJob, "residence", "Cambridge", LL_TEXT, NULL);
LlDefineFieldExtHandle(hJob, "Chart", hMeta, LL_DRAWING_META, NULL);
<... etc ...>
LlJobClose(hJob);
DeleteObject(hMeta);
```

See also:

LlDefineFieldStart, LlDefineField, LlDefineFieldExt

LlDefineFieldStart

Syntax:

```
INT LlDefineFieldStart(HLLJOB hJob);
```

Task:

Empties List & Label's internal field buffer to delete old field definitions.

Parameter:

hJob: List & Label job handle

Return Value:

Error code

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()*, else the "used" flag is reset and *LlPrintIsFieldUsed()* returns always FALSE.

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. This can be done on the developer's machine with both the standard and professional edition. To deploy the designer to an end user's machine, the professional version is required.

Parameter:

hJob: List & Label job handle

hWnd: handle of the main window of the calling application

lpszTitle: Window title

nObjType: project type

Value	Meaning
LL_PROJECT_LABEL	for labels
LL_PROJECT_CARD	for cards
LL_PROJECT_LIST	for lists

if required combined with 'or' with:

Value	Meaning
LL_FIXEDNAME	to delete the menu items 'new' and 'load' (preferred: *LIDesignerProhibitAction()*)
LL_NOSAVEAS	to delete the menu item 'save as' (preferred: *LIDesignerProhibitAction()*)
LL_EXPRCONVERTQUIET	to quietly undertake a possibly necessary conversion into the new expression mode instead of informing the user with a dialog box
LL_NONAMEINTITLE	no filename of the current project in List & Label's main window

lpszObjName: file name of the desired object

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 minimizing the List & Label layout designer the calling program is automatically minimized, in the subsequent restoration List & Label is also restored.

Important: On an end user machine, this call will only succeed in the Professional edition of *List & Label*. It will fail in the standard edition.

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")
LlJobClose(hJob);
```

See also:

LlDesignerProhibitAction, LlSetOption, LlSetDlgboxMode, LlSetFileExtensions

LlDefineSortOrder

Syntax:

```
INT LlDefineSortOrder(HLLJOB hJob, LPCTSTR lpszID,
  LPCTSTR lpszText);
```

Task:

Defines a sorting order and its identification string (ID). You can define multiple sort orders.

Parameter:

hJob: List & Label job handle

lpszID: unique character string, required for *LlPrintGetSortOrder()*

lpszText: Text belonging to the sorting order. This text is displayed in the designer

Return Value:

Error code

Hints:

If you wish to give a project a certain sorting order then you must define this before calling the designer so that you or the user can select a sorting for the project.

This is advisable for tables, e.g., where the sorted field usually stands at the beginning; for a project where "company" is first, the sorting should be according to the company name, for another project where "lastname" is first a sorting according to lastname should be used.

These functions only operate on table projects and must have the flag LL_OPTION_ONLYONETABLE!

This only makes sense if your application evaluates the sort order set in the project and sorts the data accordingly before printing the data!

Example:

```
HLLJOB hJob;

hJob = LlJobOpen(0);
LlDefineSortOrderStart(hJob);
LlDefineSortOrder(hJob, "Name", "Name");
LlDefineSortOrder(hJob, "Company", "Company");
<... etc ...>
LlJobClose(hJob);
```

See also:

LlDefineSortOrderStart, LlPrintGetSortOrder

LlDefineSortOrderStart

Syntax:

```
INT LlDefineSortOrderStart(HLLJOB hJob);
```

Task:

Empties List & Label's internal sorting buffer to delete old definitions.

Parameter:

hJob: List & Label job handle

Return Value:

Error code

Hints:

If you do not offer at least one sorting, the corresponding menu item does not appear in the designer.

See *LlDefineSortOrder()*.

Example:

```
HLLJOB hJob;

hJob = LlJobOpen(0);
LlDefineSortOrderStart(hJob);
LlDefineSortOrder(hJob, "Name", "Name");
LlDefineSortOrder(hJob, "Company", "Company");
<... etc ...>
LlJobClose(hJob);
```

See also:

LlDefineSortOrder, LlPrintGetSortOrder

LlDefineSumVariable

Syntax:

```
INT LlDefineSumVariable(HLLJOB hJob, LPCTSTR lpszName,
  LPSTR lpszCont);
```

Task:

Defines a sum variable's contents.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to an ANSIZ string with the name of the variable

lpszCont: Pointer to an ANSIZ string with the contents of the variable

Return Value:

Error code

Hints:

The field contents are interpreted as numerical values.

Usage 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 an ANSIZ string with the name of the variable

lpszCont: Pointer to an ANSIZ string with the contents of the variable

Return Value:

Error code

Hints:

Mind 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 record 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, INT32 lPara, LPVOID lpPara);
```

Task:

Defines a variable and its contents.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to an ANSIZ string with the name of the variable

lpszCont: Pointer to an ANSIZ 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:

Mind 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 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, LONG lPara, LPVOID lpPara);
```

Task:

Defines a variable and its contents.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to an ANSIZ string with the name of the variable

hContents: handle to an object of type:HMETAFILE, HENHMETAFILE (32 bit),HICON or HBITMAP

lPara: *LL_DRAWING_HMETA*, *LL_DRAWING_HEMETA* (32 bit), *LL_DRAWING_HICON* or *LL_DRAWING_HBITMAP*

lpPara: for future extensions, must be NULL.

Return Value:

Error code

Hints:

Mind 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 whole layout definition or until after *LlPrintFields()* respectively *LlPrint()* return.

After usage 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:

```
INT LlDefineVariableStart(HLLJOB hJob);
```

Task:

Empties List & Label's internal variable buffer to delete old definitions.

Parameter:

hJob: List & Label job handle

Return Value:

Error code

Hints:

Does not necessarily have to be called. However, as with every *LlDefineVariable...()* the internal variable list is examined for a variable which is already available with the same name and type, this can be accelerated somewhat 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()*, else *LlPrintIsVariableUsed()* will always return FALSE.

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

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

Return Value:

Error code

Hints:

If this function is used then 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.

The function index can have the following values:

Value	Meaning
0	All function exclusions are deleted, the menu item list is re-set (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 would be added.
LL_SYSCOMMAND_MINIMIZE	the designer window cannot be minimized (iconized).
LL_SYSCOMMAND_MAXIMIZE	the designer window cannot be maximized.
other	Here the menu IDs of the deleted menus can be given. The IDs of the menu items in List & Label can be found in the file MENUID.TXT included in your package.

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")
LlJobClose(hJob);
```

See also:

LlDefineLayout

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 hide will be reset.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

LlDesignerProhibitFunction(hJob, "");
LlDesignerProhibitFunction(hJob, "CStr$");
...
LlJobClose(hJob);
```

See also:

-

LlEnumGetEntry

Syntax:

```
HLISTPOS LlEnumGetEntry(HLLJOB hJob, HLISTPOS hPos,
  LPTSTR pszNameBuf, UINT nNameBufsize, LPTSTR pszContBuf,
  UINT nContBufSize, HANDLE* pHandle, INT32* 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 a variable of type INT32, 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.

Example:

The following example traverses the list of variables and prints out 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, UINT32 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, UINT32 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, UINT32 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 more 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,
  UINT32 nFlags);
```

Task:

Returns the next field/variable (if any). The iteration starts with *LlEnumGetFirstVar()* or *LlEnumGetFirstField()* and is being 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 more 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, INT 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 copy

Return Value:

Error status

Hints:

The function must be called straight after *LlExprParse()* returned an error.

Example:

See LlExprParse

See also:

LlExprParse, LlExprEvaluate, LlExprType, LlExprFree

LlExprEvaluate

Syntax:

```
INT LlExprEvaluate(HLLJOB hJOB, LPVOID lpExpr, LPTSTR lpBuffer,
  INT 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 copy

Return Value:

Error status

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 given 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.

Example:

See LlExprParse

See also:

LlExprParse, LlExprType, LlExprError, LlExprFree

LlExprFree

Syntax:

```
void LlExprFree(HLLJOB hJob, LPVOID 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 storage losses the function must be called when a tree returned by *LlExprParse()* is no longer required.

Example:

See LlExprParse

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 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 to 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", "29.2.2000", LL_TEXT);
   LlExprEvaluate(hJob, lpExpr, lpszBuf, sizeof(lpszBuf));
   lDateTwo = atol(lpszBuf);
   // lDateTwo now has the Julian date
   // 29.2.2000 (yes, that's ok!)
   LlExprFree(hJob, lpExpr);
}
else
{
   // Error!
   LlExprError(hJob, lpszErrortext, sizeof(lpszErrortext));
}
```

See also:

LlExprEvaluate, LlExprType, LlExprError, LlExprFree

LlExprType

Syntax:

```
INT LlExprType(HLLJOB hJOB, LPVOID lpExpr);
```

Task:

Evaluates the result type of the expression.

Parameter:

hJob: List & Label job handle

lpExpr: the pointer returned from the corresponding *LIExprParse()*

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 LIExprParse

See also:

LIExprParse, LIExprEvaluate, LIExprError, LIExprFree

LlGetDlgboxMode

Syntax:

```
UINT LlGetDlgboxMode(void);
```

Task:

Returns the current setting of the dialog box-layout.

Return Value:

The dialog box mode:

Dialog box mode: *LL_DLGBOXMODE_SAA, LL_DLGBOXMODE_ALT1... LL_DLGBOXMODE_ALT9*

'or'ed with the extended flags

extended flags: *LL_DLGBOXMODE_3DBUTTONS*, *LL_DLGBOXMODE_NOBITMAPBUTTONS*

See also:

LlSetDlgboxMode, LlJobOpen, LlJobOpenLCID

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 an ANSIZ string with the name of the chart field

lpszBuffer: address of buffer for contents

nBufSize: maximum number of characters to copy

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 also:

LlDefineChartFieldStart, LlDefineChartFieldExt, LlGetFieldType

LlGetDefaultProjectParameter

Syntax:

```
INT LlGetDefaultProjectParameter(HLLJOB hLlJob,
  LPCTSTR pszParameter, LPTSTR pszBuffer, INT nBufSize,
  UINT32* pnFlags)
```

Task:

Returns the default value of a project parameter (see project parameters chapter)

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 UINT32 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 also:

LlSetDefaultProjectParameter, LlPrintSetProjectParameter, LlPrintGetProject-Parameter

LlGetFieldContents

Syntax:

```
INT LlGetFieldContents(HLLJOB hJob, LPCTSTR lpszName,
  LPTSTR lpszBuffer, INT nBufSize);
```

Task:

Returns the contents of the corresponding (chart) field.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to an ANSIZ string with the name of the field

lpszBuffer: address of buffer for contents

nBufSize: maximum number of characters to copy

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 also:

LlDefineFieldStart, LlDefineFieldExt, LlDefineFieldExtHandle, LlGetFieldType

LlGetFieldType

Syntax:

```
INT32 LlGetFieldType(HLLJOB hJob, LPCTSTR lpszName);
```

Task:

Returns the type of the corresponding field.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to an ANSIZ 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:

LIDefineFieldStart, LIDefineFieldExt, LIDefineFieldExtHandle, LIGetFieldContents

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.

This function may not be used when List & Label's VCL, VBX or OCX control is used.

Example:

```
HLLJOB    hJob;
UINT      wMsg;

LlSetDebug(TRUE);
hJob = LlJobOpen(0);
v = LlGetNotificationMessage(hJob);
...
LlJobClose(hJob);
```

See also:

LISetNotificationMessage, LISetNotificationCallback

LlGetOption

Syntax:

```
DWORD LlGetOption(HLLJOB hJob, INT nMode);
```

Task:

Requests diverse 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 at the description of *LlSetOption()*. In addition, there are some new or (in regards 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: Tests 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;
UINT32    nLanguage;

LlSetDebug(TRUE);
hJob = LlJobOpen(0);
// ....
nLanguage = LlGetOption(hJob, LL_OPTION_LANGUAGE);
// ....
LlJobClose(hJob);
```

See also:

LlSetOption

LlGetOptionString

Syntax:

```
DWORD LlGetOptionString(HLLJOB hJob, INT nMode, LPTSTR pszBuffer,
  UINT nBufSize);
```

Task:

Requests diverse 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()*.

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, LPCTSTR pszPrinter,
  UINT* pnSizePrn, DEVMODE* pDM, UINT* pnSizeDm);
```

Task:

Queries the printer configuration of 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: filename of the project.

nPrinter: index of the printer to be queried (0=first, 1=second)

pszPrinter: address of buffer for printer name. If this pointer is NULL and pnSizePrn is not NULL, the needed size of the buffer will be stored in *pnSizePrn.

pnSizePrn: address of variable with buffer size.

pDM: address of buffer for the DEVMODE structure. If this pointer is NULL and pnSizeDm non-NULL, the needed 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.

Example:

```
WCHAR     szFilenameW[MAX_PATH];
UINT      PrnSize = 0;
UINT      DEVMODESize = 0;

mbstowcs(szFilenameW,szFilenameA,MAX_PATH);

LlGetPrinterFromPrinterFileW(hJob,nPrjType,szFilenameW, nPrn,
   NULL,&nPrnSize,
   NULL,&nDEVMODESize);

WCHAR      szPrinter[MAX_PATH];
DEVMODEW*  pDEVMODE = (DEVMODEW*)(new BYTE[nDEVMODESize]);

ASSERT(nPrnSize <= sizeof(szPrinter)/sizeof(szPrinter[0]));
LlGetPrinterFromPrinterFileW(hJob,nPrjType,szFilenameW,nPrn,
   szPrinter,&nPrnSize,
   pDEVMODE,&nDEVMODESize);

...

delete[] (BYTE*)pDEVMODE;
```

See also:

LlSetPrinterInPrinterFile

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 an ANSIZ string with the name of the sum variable.

lpszBuffer: address of buffer for contents

nBufSize: maximum number of characters to copy

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 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 used within the given project file.

Parameter:

hJob: List & Label Job-Handle

lpszProjectName: Pointer to an ANSIZ string with the project name

lpszBuffer address of buffer for contents

nBufSize: maximum number of characters to copy

Return Value:

Error code (almost LL_ERR_UNKNOWN_FIELD or 0)

Hints:

Might raise an exception, if the information of the used variables, fields and chart fields is not present (i.e. an old style project is used). You usually don't need to care about used identifiers, the data binding automatically handles the calls for you. When working database independent, it is preferable to use the "IsUsed" property of a member of the variables collection and it's counterparts for fields and chart fields.

See also:

LlPrintIsVariableUsed, LlPrintIsChartFieldUsed, LlPrintIsFieldUsed

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 an ANSIZ string with the name of the sum variable.

lpszBuffer: address of buffer for contents

nBufSize: maximum number of characters to copy

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 by *LlGetVariableType()* or *LlGetFieldType()*.

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 an ANSIZ string with the name of the variable

lpszBuffer: address of buffer for contents

nBufSize: maximum number of characters to copy

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 also:

LIDefineVariableStart, LIDefineVariableExt, LIDefineVariableExtHandle, LIGetVariableType

LlGetVariableType

Syntax:

```
INT32 LlGetVariableType(HLLJOB hJob, LPCTSTR lpszName);
```

Task:

Returns the type of the corresponding variable.

Parameter:

hJob: List & Label job handle

lpszName: Pointer to an ANSIZ 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:

LIDefineVariableStart, LIDefineVariableExt, LIDefineVariableExtHandle, LIGetVariableContents

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
LL_VERSION_MINOR (2)	Returns the minor version number, for example 1, (1 means 001 as the subversion has three digits).

Return Value:

see parameter

Example:

```
int   v;
v = LlGetVersion(VERSION_MAJOR);
```

LlJobClose

Syntax:

```
void LlJobClose(HLLJOB hJob);
```

Task:

Release the internal variables, free 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) 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-_NOLLXPRELOAD,* no List & Label extensions (Export module, Chart object,...) 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 value less than 0 shows an error.

Hints:

For ease of maintenance, we suggest global settings to be put in one place, immediately after the *LlJobOpen()* call (dialog design, callback modes, expression mode, ...).

The C?LL12.DLL requires the language dependent components which are stored in a separate DLL, e.g. C?LL1200.LNG or C?LL1201.LNG. They are loaded depending on the language setting. See also chapter 1.

The passed value is application dependent so that a value set by your application does not collide with other applications; several applications can therefore use different language DLLs simultaneously.

If List & Label is no longer required then the job should be released with the function *LlJobClose()* to give the DLL a chance of releasing the internal variables of this job.

Many additional language kits are available. Ask our sales department for further information or have a look at our internet site at www.combit.net. The language IDs appended to the filename 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, LlSetDlgboxMode, LlDesignerProhibitAction, LlSetFileExtensions

LlJobOpenLCID

Syntax:

```
HLLJOB LlJobOpenLCID(UINT32 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, LlSetDlgboxMode, LlDesignerProhibitAction, LlSetFileExtensions

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 path if necessary)

lpszPath: valid path of the temporary files ending with a backslash "\". Needs only be defined if *LlPreviewSetTempPath()* has been called, else it should be NULL or empty.

Return Value:

Error status

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 real data preview window.

Parameter:

hJob: List & Label job handle

lpszObjName: valid file name with full pathname!

lpszPath: valid path for the temporary files ending with a backslash "\". Need only be given if *LlPreviewSetTempPath()* has been called for the preview print process, else it can be NULL or empty.

hWnd: Window handle of the calling program

Return Value:

Error status

Hints:

The real data preview is a window that can be started independent 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, UINT32 nOptions, LPVOID pOptions);
```

Task:

Starts the real data preview. Additional options can define the behavior.

Parameter:

hJob: List & Label job handle

lpszObjName: valid file name with full pathname!

lpszPath: valid path for the temporary 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 (that is 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 status

Hints:

The real data preview is a window that can be started independent 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). Useful especially for applications running in a network environment.

Parameter:

hJob: List & Label job handle

lpszPath: valid path with a concluding backslash "\"

Return Value:

Error status

Hints:

In this path, the preview file(s) will be stored. The filename 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 status

Hints:

Normal objects and the header of a table object (see option *LL_OPTION_DELAYTABLEHEADER*) 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 caused a page break have to 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 stays incomplete or may not be printed).

Parameter:

hJob: List & Label job handle

Return Value:

Error codes

Hints:

Is necessary to abort the print through the program 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 are still in the printer driver are 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",
   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 restoring the printer configuration file.

Parameter:

hJob: List & Label job handle

lpszFilename: file name of the printer configuration file

nFunction: action

Action	Meaning
LL_PRINTERCONFIG_SAVE	saves the printer configuration file of the currently opened project in a file with the name of lpszFilename.
LL_PRINTERCONFIG_-RESTORE	copies the previously saved configuration file (created with *LL_PRINTERCONFIG_SAVE*) back to the current project.

Return Value:

Error status (always 0)

Hints:

It is important that *LL_PRINTERCONFIG_RESTORE* is being called before(!) *LlPrint()*!

Example:

The following principle should be used for hand-made copies on a temporary printer, that is, a user can select to have a temporary change of the printer using the printer dialog box, and choose multiple copies. Usually the second and the 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, LlPrintStart, LlPrintWithBoxStart, LlSetPrinterInPrinterFile, LlGetPrinterFromPrinterFile, LlSetPrinterDefaultsDir

LlPrintDbGetRootTableCount

Syntax:

```
INT LlPrintDbGetRootTableCount(HLLJOB hJob);
```

Task:

Returns the number of tables at the root level. Needed for correctly displaying a progress bar.

Parameter:

hJob: List & Label Job-Handle

Return Value:

Number of tables

Hints:

See the hints in chapter "4. Working with the Report Container".

See also:

LlDbAddTable, LlDbAddTableRelation, LlDbAddTableSortOrder, LlPrintDbGetCurrentTable, LlPrintDbGetCurrentTableSortOrder, LlPrintDbGetCurrentTableRelation

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 "4. Working with the Report Container".

See also:

LlDbAddTable, LlDbAddTableRelation, LlDbAddTableSortOrder, LlPrintDbGetCurrent-TableSortOrder, LlPrintDbGetCurrentTableRelation

LlPrintDbGetCurrentTableRelation

Syntax:

```
INT LlPrintDbGetCurrentTableRelation(HLLJOB hJob,
  LPTSTR pszRelationID, UINT nRelationIDLength);
```

Task:

This function returns the current table relation to be printed.

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 "4. Working with the Report Container".

See also:

LlDbAddTable, LlDbAddTableRelation, LlDbAddTableSortOrder, LlPrintDbGetCurrent-Table, LlPrintDbGetCurrentTableSortOrder

LlPrintDbGetCurrentTableSortOrder

Syntax:

```
INT LlPrintDbGetCurrentTableSortOrder(HLLJOB hJob,
  LPTSTR pszSortOrderID, UINT nSortOrderIDLength);
```

Task:

This function returns the current table sort order to be printed.

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 "4. Working with the Report Container".

See also:

LlDbAddTable, LlDbAddTableRelation, LlDbAddTableSortOrder, LlPrintDbGetCurrent-Table, LlPrintDbGetCurrentTableRelation

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:

Informs whether the last data record printed did match the filter given by the user, i.e. if it was really printed.

Parameter:

hJob: 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

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 status (important!)

Hints:

The object name can be " (empty) to address all objects, otherwise it has to 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 if 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 do always use *LlPrintEnd()* if you have used *LlPrintStart()* or *LlPrintWithBoxStart()*, even if these commands were aborted with an error, otherwise resource and memory losses may result.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

if (LlPrintStart(hJob, LL_PROJECT_LABEL, "test", 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 (including path)

Return Value:

Error code

Hints:

Has to be called before *LlPrint(WithBox)Start()*. Allows printer selection without having to perform a printing process (as you have to 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 status or *LL_WRN_REPEAT_DATA*

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 that you have to start a new page for the entry.

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 codes

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.

Example:

```
HLLJOB hJob;
hJob = LlJobOpen(0);

if (LlPrintStart(hJob, LL_PROJECT_LIST, "test",
   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, INT 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:

Get 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 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:

For *LL_PROJECT_LIST* 1 is always returned.

Can be used to calculate the total number of output pages. See hints in the programming part of this manual.

See also:

LlPrintGetItemsPerTable, LlPrintGetRemainingItemsPerTable

LlPrintGetItemsPerTable

Syntax:

```
INT LlPrintGetItemsPerTable(HLLJOB hJob);
```

Task:

Returns the number of lines of the smallest list on a page.

Parameter:

hJob: List & Label job handle

Return Value:

<0: error code
>= 0: number of table lines

Hints:

Can be used to calculate the total number of output pages. This value need not be correct as the height of lines of various table lines may be different because of word wrap or different numbers of lines per field, multiple table lines etc..

Any appearance condition and expression formula used to calculate this is using the currently defined variables and fields.

Example:

See the program example earlier

See also:

LlPrintGetItemsPerPage, LlPrintGetRemainingItemsPerTable

LlPrintGetOption

Syntax:

```
INT LlPrintGetOption(HLLJOB hJob, INT nIndex);
```

Task:

Returns the various print options which are set by the user after calling *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 whether the currently selected copies count is supported by the printer (usually, this is 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* have to be set to 1 and the copies have to be done manually, if needed.

LL_PRNOPT_UNIT

This option returns the measurement units set in the system's settings (registry settings). Returned values are: *LL_UNITS_MM_DIV_10* for 1/10 mm, *LL_UNITS_INCH_DIV_100* for 1/100 inch and *LL_UNITS_INCH_DIV_1000* for 1/1000 inch.

LL_PRNOPT_PRINTORDER

Returns the print order of the (labels/file cards) of the project. Default: *LL_PRINTORDER_HORZ_LTRB* (horizontal, from left top to right bottom)

LL_PRNOPT_DEFPRINTERINSTALLED

Returns a flag 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 lpszBuffer, UINT nBufSize);
```

Task:

Returns the various print option string settings.

Parameter:

hJob: List & Label job handle

nIndex: see *LlPrintSetOptionString()*

lpszBuffer: address of buffer for the string

nBufSize: maximum number of characters to copy

Return Value:

error code

Hints:

See LlPrintSetOptionString

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 also:

LlPrintStart, LlPrintWithBoxStart

LlPrintGetProjectParameter

Syntax:

```
INT LlPrintGetProjectParameter(HLLJOB hLlJob, LPCTSTR pszParameter,
  BOOL bEvaluated, LPTSTR pszBuffer, INT nBufSize, UINT32* 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 type LL_PARAMETERFLAG_FORMULA, this flag decides if the parameter should be evaluated first.

nBufSize: size of the buffer (in TCHARs)

Return Value:

Error code or required buffer size

Hints:

This function can not 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 also:

LlSetDefaultProjectParameter, LlGetDefaultProjectParameter, LlPrintGetProjectParameter

LlPrintGetRemainingItemsPerTable

Syntax:

```
INT LlPrintGetRemainingItemsPerTable(HLLJOB hJob,
  LPCTSTR lpszField);
```

Task:

Returns the remaining number of lines in a table.

Parameter:

hJob: List & Label job handle

lpszField: determines the table (with the field given here) which is taken for the calculation.

Return Value:

<0: error code
>= 0: number of remaining table lines

Hints:

As calculation is performed using the data defined at the time the function is called, it is likely that the return value is not applicable if the number of lines varies due to word breaks, multiple table lines and such. Variable and field contents cannot be modified until the next data line is printed.

If lpszField is NULL or points to an empty string (""), the minimal value of all tables is used, otherwise the table containing the given field is used for the calculation. Alternatively, an object name with a prefix ":" like e.g. ':Table1' can be given.

Appearance conditions and formulas will be evaluated with the currently defined data.

See also:

LlPrintGetItemsPerPage,LlPrintGetItemsPerTable,
LlPrintGetRemainingSpacePerTable

LlPrintGetRemainingSpacePerTable

Syntax:

```
INT LlPrintGetRemainingSpacePerTable(HLLJOB hJob, LPCTSTR lpszField,
  INT nDimension);
```

Task:

Returns the amount of space remaining in the table.

Parameter:

hJob: List & Label Job-Handle

lpszField: determines the table (with the field given here) which is taken for the calculation.

nDimension: defines the „dimension" of the value.

nDimensions should take one of the following values:

Value	Description
LL_GRIPT_DIM_SCM	Return value is in SCM units (1/1000 mm or 1/25400 inch)
LL_GRIPT_DIM_PERC	Return value is the percentage compared to the table size on the current page

Return Value:

Remaining space or error code (negative)

Hints:

If lpszField is NULL or points to an empty string (""), the minimal value of all tables is used, otherwise the table containing the given field is used for the calculation. Alternatively, an object name with a prefix ":" like e.g. ':Table1' can be given.

LlPrintGetSortOrder

Syntax:

```
INT LlPrintGetSortOrder(HLLJOB hJob, LPTSTR lpszBuffer,
  INT nBufSize);
```

Task:

Asks for the identification string (ID) of the sort order that the user has chosen in the designer.

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()*.

When printing a project, use this function get the sort order chosen by the user and then pass the data sorted in the corresponding sorting to List & Label.

Example:

```
hJob = LlJobOpen(0);
if (LlPrintStart(hJob, LL_PROJECT_LABEL, "test",
   LL_PRINT_NORMAL) == 0)
{
   LlPrintGetSortOrder(hJob, sSortOrder, sizeof(sSortOrder);
   <... change data index ...>
   <... output ...>
   LlPrintEnd(hJob);
}
else
   MessageBox(NULL, "error", "List & Label", MB_OK);
LlJobClose(hJob);
```

See also:

LlDefineSortOrderStart, LlDefineSortOrder

LlPrintIsChartFieldUsed

Syntax:

```
INT LlPrintIsChartFieldUsed(HLLJOB hJob, LPCTSTR lpszFieldName);
```

Task:

Informs whether the given chart field from the loaded project 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

Hints:

This function can only be called after *LlPrintStart()* or *LlPrintWithBoxStart()*.

This function needs *LL_OPTION_NEWEXPRESSIONS* 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. So, don't use *LlDefineChartFieldStart()* after *LlPrint[WithBox]Start()*.

Example:

```
if (LlPrintIsChartFieldUsed(hJob, "Month") == 1)
   LlDefineChartFieldExt(hJob, "Month", <...>);
```

See also:

LlPrintStart, LlPrintWithBoxStart, LlPrintIsVariableUsed, LlPrintIsFieldUsed

LlPrintIsFieldUsed

Syntax:

```
INT LlPrintIsFieldUsed(HLLJOB hJob, LPCTSTR lpszFieldName);
```

Task:

Informs whether the given field from the loaded project is used in one of the expressions or conditions of the project.

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

Hints:

This function can only be called after *LlPrintStart()* or *LlPrintWithBoxStart()*.

This function needs *LL_OPTION_NEWEXPRESSIONS* 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. So, don't use *LlDefineFieldStart()* after *LlPrint[WithBox]Start()*.

You can also do a wildcard search instead of searching for a specific field. This is especially useful if you pass your fields ordered hierarchically, ex. all fields from the "Article" table use "Article." as prefix. Simply do a search for "Article.*" then to find out if 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:

Informs whether the given variable from the loaded project is used in one of the expressions or conditions of the project.

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

Hints:

This function can only be called after *LlPrintStart()* or *LlPrintWithBoxStart()*.

This function needs *LL_OPTION_NEWEXPRESSIONS* 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. So, don't' use *LlDefineVariableStart()* after *LlPrint[WithBox]Start()*.

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 there 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 corresponding function implemented. See programmer's hints in this manual.

The function *LlPrinterSetup(...)* offers the possibility of calling a print selection dialog without further settings.

See also:

LlPrinterSetup, LlPrintSetOption, LlPrintGetOption, LlPrintOptionsDialog

LlPrintResetObjectStates

Syntax:

```
INT LlPrintResetObjectStates(HLLJOB hJob);
```

Task:

Resets the print status of all objects.

Parameter:

hJob: List & Label Job handle

Return Value:

Error code

Hints:

Every object in List & Label has an object state (i.e.: waiting for print, printing, unfinished, finished) This state can be reset to force a reprint of already finished objects.

This function can be used for mail merge tasks.

See Also:

LlPrintResetProjectState

LlPrintResetProjectState

Syntax:

```
INT LlPrintResetProjectState(HLLJOB hJob);
```

Task:

Resets the print state of the whole project, so that printing starts as if it was a new project.

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).

Up to version 6, for a mail merge print the project had to be loaded for each letter (which took time). Now the project state can be reset during the print before each new letter. This also has the advantage that all letters are in the same preview file, which had to be done programmatically before version 7.

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>
```

See Also:

LlPrintResetObjectStates

LlPrintSelectOffsetEx

Syntax:

```
INT LlPrintSelectOffsetEx(HLLJOB hJob, HWND hWnd);
```

Task:

Opens a dialog where 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 by *LlPrintSetOption(hJob, 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, LPSTR 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 in percent

Return Value:

Error code

Hints:

To make the text multi-line, linefeeds ('\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",
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 pre-set the number of copies required.

Parameter:

hJob: List & Label job handle

nIndex: see below

nValue: sets the option corresponding to the nIndex

Return Value:

Error code

Hints:

Values for nIndex:

LL_PRNOPT_COPIES

is the number of copies to be pre-set 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_PAGE

is the 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_OFFSET

is the 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_FIRSTPAGE

is the first page of the page range that shall be printed. If "All" has been chosen, this is identical to *LL_PRNOPT_PAGE*.

LL_PRNOPT_LASTPAGE

is the page number of the last page to be printed.

Default: 32767

LL_PRNOPT_JOBPAGES

is the number of pages a print job should contain if you choose *LL_PRINT_MULTIJOB* in *LlPrint[WithBox]Start()*.

Default: 16

LL_PRNOPT_PRINTDLG_ONLYPRINTERCOPIES

The PrintOptionsDialog will only allow a copies value to be entered if the printer supports copies.

Careful: the printer copies might not be useful for labels, as they might need programmer's copies support instead of the printer's. See chapter "Copies".

Default: FALSE

LL_PRNOPT_PRINTDLG_DESTMASK

Outdated, please use *LL_OPTIONSTR_EXPORTS_ALLOWED*

It is a combination of possible output media:

Value	Meaning
LL_DESTINATION_PRN	Printer
LL_DESTINATION_PRV	Preview
LL_DESTINATION_FILE	File

These will be defaulted by the nPrintOptions-parameter of the *LlPrint[WithBox]Start()* API call the following way:

nPrintOptions	Default value
LL_PRINT_NORMAL	*LL_DESTINATION_PRN*
LL_PRINT_PREVIEW	*LL_DESTINATION_PRV*
LL_PRINT_FILE	*LL_DESTINATION_FILE*
LL_PRINT_USERSELECT	all media flags combined

To allow exports, you need to use *LL_OPTIONSTR_EXPORT* and *LL_OPTIONSTR_EXPORTS_ALLOWED* instead of this option. Please refer to chapter 2.6.3. Export Modules for details.

Example

```
LlPrintStart(hJob,...,LL_PRINT_USERSELECT,...);

// allow only printer and preview (USERSELECT sets all bits)
LlPrintSetOption(hJob,LL_PRNOPT_PRINTDLG_DESTMASK,
      LL_DESTINATION_PRN|LL_DESTINATION_PRV);
```

```
// Default should be preview!
LlPrintSetOption(hJob,LL_PRNOPT_PRINTDLG_DEST,LL_DESTINATION_PRV);
// printer dialog allows user to change
LlPrintOptionsDialog(hJob,....);
// so get the final media:
bPreview = LlPrintGetOption(hJob, LL_PRNOPT_PRINTDLG_DEST) ==
LL_DESTINATION_PRV;
// ...print job....
// finished
LlPrintEnd(hJob,0);

if (bPreview)
{
   LlPreviewDisplay(hJob,...);
   LlPreviewDeleteFiles(hJob,...);
}
```

LL_PRNOPT_PRINTDLG_DEST

Outdated, please use *LL_OPTIONSTR_EXPORT*.

It is the output medium flag and defines both the default output medium and - after the print dialog - the chosen medium.

If more than one medium has been set in *xxx_DESTMASK*, this flag should be used - after the first *LlPrint()*, but before *LlPrintEnd()* - to check whether preview print has been selected.

nPrintOptions	Default value
LL_PRINT_NORMAL	*LL_DESTINATION_PRN*
LL_PRINT_PREVIEW	*LL_DESTINATION_PRV*
LL_PRINT_FILE	*LL_DESTINATION_FILE*
LL_PRINT_USERSELECT	all media flags combined

To allow exports, you need to use *LL_OPTIONSTR_EXPORT* and *LL_OPTIONSTR_EXPORTS_ALLOWED* instead of this option. Please refer to chapter 2.6.3. Export Modules for details.

See example above.

LL_PRNOPT_JOBID

Queried after *LlPrint()*, it will return the spooler job ID of the print. When there are two different printers or the print issues multiple jobs, a call to *LlPrint()* may return a different value than the previous ones, so we suggest to collect them after each *LlPrint()*.

Using this ID and some spooler functions of the Windows API, you can check for the actual completion of a print request.

LL_PRNOPT_UNITS

Returns the same value as *LlGetOption(..., LL_OPTION_UNITS).*

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

Returns the default export destination (for example „RTF", „HTML", „PDF", etc.) to be used (or shown in the print dialog)

LL_PRNOPTSTR_PRINTDST_FILENAME

is 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, UINT32 nFlags)
```

Task:

Changes the value of a project parameter (see project parameter chapter)

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 can not be called before LlPrint[WithBox]Start()!

See also:

LlSetDefaultProjectParameter, LlGetDefaultProjectParameter, LlPrintGetProjectParameter

LlPrintStart

Syntax:

```
INT LlPrintStart(HLLJOB hJob, UINT nObjType, LPCTSTR lpszObjName,
  INT nPrintOptions, INT nV1Dummy);
```

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

nPrintOptions: print options

Value	Meaning
LL_PRINT_NORMAL	output on printer
LL_PRINT_PREVIEW	output on preview
LL_PRINT_FILE	output to file
LL_PRINT_EXPORT	output to an export module, that can be defined with *LlPrintSetOptionString(LL_PRNOPTSTR_EXPORT)*
LL_PRINT_USERSELECT	output defaults to printer, but user can select in printer dialog

optionally combined with *LL_PRINT_MULTIPLE_JOBS*: output in several smaller print jobs (see below) with network spooler print.

nV1Dummy: Dummy variable so that applications for List & Label V 1.x also work with the new call syntax

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 and the print can already begin. The number of the page after which the job should split can be set with *LlPrintSetOption()*.

No abort dialog box is started - this is left to the programmer. But for the sake of the user we suggest this to be done!

Example:

```
HLLJOB hJob;

hJob = LlJobOpen(0);

if (LlPrintStart(hJob, LL_PROJECT_LABEL, "test",
   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 redraw of the abort box that is 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 possibly needed 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:

Informs 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 presently 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

nPrintOptions:

Value	Meaning
LL_PRINT_NORMAL	output on printer
LL_PRINT_PREVIEW	output on 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)*
LL_PRINT_USERSELECT	output defaults to printer, but user can select in printer dialog

optionally combined with *LL_PRINT_MULTIPLE_JOBS*: output in several smaller print jobs (see below) with network spooler print. These options take influence on *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

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 if required (see below) also an abort button.

Example:

```
HLLJOB    hJob;
hJob = LlJobOpen(0);

if (LlPrintWithBoxStart(hJob, LL_PROJECT_LABEL, "test",
  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

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 (using the Unicode-DLL) 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:

LlRTFCreateObject

LlRTFDisplay

Syntax:

```
INT LlRTFDisplay(HLLJOB hJob, HLLRTFOBJ hRTF, HDC hDC,
  RECT* pRC, BOOL bRestart, UINT* 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 shall 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. Else it will be continued after the place 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 being 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.

Return Value:

Error code

See also:

LlRTFCreateObject

LlRTFEditorInvokeAction

Syntax:

```
INT LlRTFEditorInvokeAction(HLLJOB hJob, HLLRTFOBJ hRTF,
  INT nControlID);
```

Task:

Allows activating an action in the RTF control by code. This is important for inplace 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 activate. 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 disable. 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,
  LPCTSTR lpszBuffer,UINT nBufferSize);
```

Task:

Returns the text of a 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 copy

Return Value:

Error code

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. Needed to determine the required buffer size for the text.

Parameter:

hJob: List & Label job handle

hRTF: RTF object handle

nFlags: each one option 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:

LIRTFCreateObject, LIRTFGetText

LlRTFSetText

Syntax:

```
INT LlRTFSetText(HLLJOB hJob, HLLRTFOBJ hRTF, LPCTSTR lpszText);
```

Task:

Sets the text into the RTF control. The format of the text (plain or RTF) is being auto-detected.

Parameter:

hJob: List & Label job handle

hRTF: RTF object handle

lpszText: New contents

Return Value:

Error code

See also:

LIRTFCreateObject

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
0	all projects are displayed

combined with *LL_FILE_ALSONEW* if a filename for a new (not yet existing) project can be entered.

pszBuffer, nBufSize: Buffer for the new filename. Must be initialized with a file name or an empty string.

pReserved: reserved, set to NULL or empty ("").

Return Value:

Error status

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 a 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.

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_EXPLORER,
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, else 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 windows directory).

Hints:

Use the program DEBWIN2.EXE included in your package to show the debug output in a separate window.

If the 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. A '@' 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 the debugging mode calls other functions of a DLL (even itself) which is also in the debugging mode.

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, UINT32 nFlags)
```

Task:

Sets the default value of a project parameter (see project parameter chapter)

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 project parameters chapter 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

LlSetDlgboxMode

Syntax:

```
void LlSetDlgboxMode(WORD nMode);
```

Task:

To set the layout parameter of the dialog boxes.

Parameter:

nMode: desired dialog box mode (see below)

Hints:

The dialog box mode is a number of flags combined with 'or' from the dialog box mode and the extended flags:

Value	Meaning
LL_DLGBOXMODE_SAA	Standard-Windows-style
LL_DLGBOXMODE_ALT1	...
...	...
LL_DLGBOXMODE_ALT8	Win95 Button style
LL_DLGBOXMODE_ALT9	Office 97 Button style
LL_DLGBOXMODE_ALT10	Office 97 Button style with Gray Buttons

extended flags:

Value	Meaning
LL_DLGBOXMODE_3DBUTTONS	Button-Texts have 3D style
LL_DLGBOXMODE_TOOLTIPS98	Tooltips in Win98 style
LL_DLGBOXMODE_NOBITMAPS	Buttons without bitmaps

This function should be called directly after *LlJobOpen()* or *LlJobOpenLCID()*.

The values set here are only valid for your application so that List & Label can use different dialog box modes in different applications.

Example:

```
LlSetDlgboxMode(LL_DLGBOXMODE_ALT9);
   // Set "flat" button style
```

See also:

LlGetDlgboxMode, LlJobOpen, LlJobOpenLCID

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 (3 extensions * 3 project types) are different!

Please call this function before *LlDefineLayout()* and before the functions *LlPrint...Start()*, best of all directly after *LlJobOpen()* or *LlJobOpenLCID()*.

You can get and set these extensions by *LlSetOptionString()* also.

Example:

```
HLLJOB hJob;
int    v;
```

```
hJob = LlJobOpen(0);
v = LlSetFileExtensions(hJob, LL_PROJECT_LIST, "llx", "lly", "llz");
// ....
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 else

Hints:

The callback function has higher priority than the message; if it is defined no message is sent but rather the callback function is called.

This function cannot be used if the 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 the chapter on callback objects.

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 else

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 if a general callback handler was installed with *LlSetNotificationCallback*. If so, it will be called. If not, List & Label checks if a message for the current event was defined using *LlSetNotificationMessage.* If so, the message will be sent. This function may not be used with the .NET component as this component uses the API for its own functionality already,

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 the chapter on callback objects.

Example:

```
LPARAM STDCALL MyCallback(UINT nFunction, LPARAM lParam)
//....

hJob 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 pre-setting 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 message's parameters can be found in the chapter about callback objects.

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, DWORD 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()*, best of all 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 list projects, only fields will be offered (Default).

This option only affects list projects.

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_AUTOMULTIPAGE

TRUE: List & Label is allowed to use its page break algorithm. This means you need some modification in your code if you worked with older versions of List & Label than version 6 (see appendix). (Default).

FALSE: no automatic page wraps are possible.

LL_OPTION_CALCSUMVARSONINVISIBLELINES

This sets the default value for the designer option whether sum variables should also be calculated if data lines are being suppressed or not. The value selected in the designer will then be saved in and loaded from the project file.

Default: FALSE

LL_OPTION_CALLBACKMASK

Any combination of the following values is possible as a value:

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 about 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 "unnatural" APIs: the "W" API of the SBCS/DBCS DLL and the "A" API of the Unicode DLL. It is also used to convert a Unicode project file for a SBCS/DBCS DLL and vice versa.

This setting is globally used, so it's valid for all List & Label jobs in one task, and the hJob parameter is being ignored.

Of course, you need the code page installed on your system (NLS[7] for that code page has to be installed).

Default: *CP_ACP*.

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

Used in conjunction with *LL_OPTION_STORAGESYSTEM*.

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).

[7] NLS = National Language Support

FALSE: contents stay unchanged (Default).

LL_OPTION_DEFDEFFONT

Allows you to set the handle of the font used as default for the project's default font. The handle needs 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: triggered by the first table line

FALSE: at *LlPrint()*. Of course, if fields are used in the header line, they must be defined at the *LlPrint()* call. (Default)

LL_OPTION_ENABLE_STANDALONE_DATACOLLECTING_OBJECTS

Compatibility option only. After switching to TRUE, charts and crosstabs will be available as standalone objects even in multi table mode. This implies the risk of user induced Design errors. We recommend leaving the option switched off. Older projects containing charts as standalone objects will still print fine and can also be Designed. However, the user will not be able to insert new standalone objects.

TRUE: Charts and crosstabs are available as standalone objects even in multi table mode

FALSE: Charts and crosstabs can only be inserted into the report container in multi table mode

Default: FALSE

LL_OPTION_EMFRESOLUTION

To display pages in the preview that are larger than 32.76 cm, a workaround has to be used to be able to display the EMF preview file by changing the EMF resolution.

The value passed to this option defines the behavior:

0: List & Label computes the optimal value (Default)

other value: divisor of millimeter, for example 100 is 1/100 mm resolution.

See LLStgsysGetJobOptionValue(*LS_OPTION_EMFRESOLUTION*).

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 in 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_EXTENDEDEVALUATION

FALSE: old expression mode. Note: This mode is no longer supported!

TRUE: a "mixed" mode: combines the simplicity of the old expression mode with the features of the new expression mode

Please note that setting this option overrides the settings of *LL_OPTION_NEWEXPRESSIONS*! Switching *LL_OPTION_EXTENDEDEVALUATION* to FALSE automatically switches *LL_OPTION_NEWEXPRESSIONS* to FALSE. As this mode is no longer supported, don't set this option to FALSE.

LL_OPTION_FONTQUALITY

(See also LL_OPTION_FONTPRECISION below)

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.lfQuality field when a font instance is being created.

The allowed 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 allowed values are referenced in the MSDN documentation.

Default: OUT_STRING_PRECIS.

LL_OPTION_FORCEFONTCHARSET

Selects whether all fonts in the system are being offered in font selection combo boxes or whether they must support the charset of the default LCID (or the font set by *LL_OPTION_LCID*). See also *LL_OPTION_-SCALABLEFONTSONLY*.

Default: FALSE

LL_OPTION_HELPAVAILABLE

TRUE: display help-buttons (Default)

FALSE: do not display help-buttons

LL_OPTION_IMMEDIATELASTPAGE

Usually the *LastPage()* flag will be set after all objects have been printed (up to a table in case of a report project). If you set this option to *TRUE*, a non-finished object will immediately force *LastPage()* to be true, and it will reset all its appended objects.

Default: *FALSE*

LL_OPTION_INTERCHARSPACING

TRUE: the space between the characters for block justified text will vary.

FALSE: only the width of blanks between words will be varied (Default)

LL_OPTION_INCLUDEFONTDESCENT

TRUE: the logfont member LOGFONT.lfDescent is regarded when calculating the line distances. This leads to a wider line space but prevents extreme font descents from being cut off.

FALSE: compatible mode (Default)

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 shipped with two different RTF controls (version numbers 1, 2 and 3). These differ in the features:

advantages of version 2:

- line distances can be varied
- distances before and after a paragraph can be varied

disadvantage of version 2:

- always white background
- still no block adjustment...

Using this option, you can prevent List & Label from loading version 2 (set option to 0x100) or allow it (set option to 0x200) if it is present in the system.

LL_OPTION_METRIC

TRUE: List & Label designer is set to metric system

FALSE: List & Label designer is set to inch

Default value depends on the system's setting.

See *LL_OPTION_UNITS*

LL_OPTION_MULTIPLETABLELINES

TRUE: Tables can have multiple line definitions that can be activated by separate conditions. This can be used to create sub reports by switching between different table lines by the application (Default).

FALSE: A table line is only one definition

LL_OPTION_NEWEXPRESSIONS

FALSE: old expression mode. Note: This mode is no longer supported!

TRUE: new expression mode (Default)

When the new expression mode is chosen the files are translated to this automatically if they have been created or saved in the old mode. If *LIDefineLayout()* has the flag *LL_EXPRCONVERTQUIET* not set then the user will be informed that the file will be converted.

Please note that setting this option overrides the settings of *LL_OPTION_EXTENDEDEVALUATION*!

LL_OPTION_NOFAXVARS

TRUE: the variables for fax are visible in the Designer (Default).

FALSE: the variables for fax are not visible in the Designer.

LL_OPTION_NOFILEVERSIONUPDATEWARNING

This option defines the behavior of the designer while 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, telling 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_NOFOOTERPAGEWRAP

Footer lines on a table which is finished will wrap to the next page(s) if necessary. When this option is set to TRUE, a footer which will not fit below the table will not be wrapped but starts on the next page, so a non-final footer line would be printed on this page instead.

Default: FALSE

LL_OPTION_NOMAILVARS

TRUE: the variables for email are visible in the Designer (Default).

FALSE: 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.

FALSE: List & Label does the check and returns *LL_ERR_NO_TABLEOBJECT* if the project contains no table. (Default)

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_NOTIFICATIONMESSAGEHWND

To set the window that is to receive notification messages (callbacks) when no callback procedure is being explicitly set.

Usually events are sent to the first non-child window starting with the window handle given in *LIDefineLayout()* or *LIPrintWithBoxStart()*. With this call you can modify the destination.

Default: NULL (default behavior)

LL_OPTION_OFNDIALOG_EXPLORER

TRUE: the file selection (open/save) dialogs have the explorer style.

FALSE: the "old" dialog style will be used.

LL_OPTION_OFNDIALOG_NOPLACESBAR

TRUE: suppresses the "places" bar in Windows 2000 file open dialogs.

FALSE: the full file dialog is displayed in Windows 2000 (Default).

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_PROJECTBACKUP

TRUE: during editing in the Designer a backup file is generated (Default).

FALSE: no backup file is generated.

LL_OPTION_ONLYONETABLE

TRUE: only one table is allowed, no second list object can be inserted.

FALSE: an arbitrary amount of list objects may be inserted (Default).

If your code does not support multiple tables, you should set this option to TRUE.

LL_OPTION_PRVZOOM_LEFT, LL_OPTION_PRVZOOM_TOP, LL_OPTION_PRVZOOM_WIDTH, LL_OPTION_PRVZOOM_HEIGHT

Real data preview: the rectangle coordinates of the preview window in percent of the screen. If not set (set to -1), the positions of the window when it has been closed last time 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

Real data preview: initial zoom factor in percent (Default: 100)

LL_OPTION_REALTIME

TRUE: Time() and *Now()* will always use the current time

FALSE: The time will be fixed once at project load time (Default)

LL_OPTION_RESOLUTIONCOMPATIBLETO9X

TRUE: List & Label barcodes are printed in a higher resolution than other objects. This can lead to display problems of barcodes on Windows 9x machines when displaying preview files created with Windows NT/200/XP. This option restricts the resolution to a displayable value.

FALSE: compatible mode (Default)

LL_OPTION_RETREPRESENTATIONCODE

Character code of the character that represents a 'newline' 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_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 can look unexpectedly.

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) into the project file and the preview file. This can be important in 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: they 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

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_STORAGESYSTEM

Until version 4, List & Label stored the preview data in multiple files (Basic information in <PROJECT>.000, printer definition in <PROJECT>.~~~ and the metafiles in <PROJECT>.<nnn>. Starting with List & Label 5.0 we introduced a new kind of data file, a single-file storage called <PROJECT>.LL. Advantages:

can be sent easily (Inter-/Intranet)

32 bit: the data can be compressed (see *LL_OPTION_COMPRESSSTORAGE*), thus reduces amount of space used on HD or transfer time

less risk of name conflicts with projects of the same name (for example, article list and article report etc).

Possible values:

LL_STG_COMPAT4: List & Label uses the old multi-file storage.

LL_STG_STORAGE: List & Label uses the new one-file storage (Default).

Please read the chapter on preview files for more details.

LL_OPTION_SUPERVISOR

TRUE: all menu options are allowed and even locked objects are not locked. This mode enables parts which are not accessible to the user to be used without much additional programming.

FALSE: limitations are valid (Default)

LL_OPTION_SUPPORTPAGEBREAK

TRUE: the page break checkbox of the text object is displayed and can be checked by the user.

FALSE: the page break checkbox is hidden (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_TABSTOPS

TRUE: List & Label replaces tabs (code 0x09) in a text by spaces. (Default)

FALSE: tabs will be expanded by spaces for a column width of 8 characters. This makes no sense however for proportional fonts.

LL_OPTION_UISTYLE

Sets the style of the designer interface.

LL_OPTION_UISTYLE_STANDARD: Office 97® Look & Feel

LL_OPTION_UISTYLE_OFFICEXP: Office XP®/Visual Studio .NET® Look & Feel

LL_OPTION_UISTYLE_OFFICE2003: Office 2003® Look & Feel

LL_OPTION_UNITS

LL_UNITS_MM_DIV_10: List & Label designer is set to metric system.

LL_UNITS_INCH_DIV_100: List & Label designer is set to inch, resolution is inch/100.

LL_UNITS_INCH_DIV_1000: List & Label designer is set to inch, resolution is inch/1000.

Default value depends on the system's setting. If the system is set to inch, inch/1000 will be the default resolution.

See *LL_OPTION_METRIC*

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 resizing the barcode object will only allow sizes in the standard size range.

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 more work. See *LL_CMND_HOSTPRINTER*.

FALSE: List & Label manages the printer device

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 options results in a bit lower speed.

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. That 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()*, best of all directly after *LlJobOpen()/LlJobOpenLCID()*. If an option needs to be set at a different moment, it is being stated in that option's description.

LL_OPTIONSTR_CARD_PRJDESCR

Use this parameter to set the description of the corresponding project type for the file dialogs.

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_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 loaded by *LL_OPTIONSTR_LLXPATHLIST*), for example "PRN;PRV;FILE;HTML;RTF"

The following IDs are predefined:

Value	Meaning
PRN	Printing to printer
PRV	Printing to Preview (file)
FILE	Printing to printer file

LL_OPTIONSTR_EXPORTS_ALLOWED

This property can be used to restrict the output list presented to the user in the *LlPrintOptionsDialog[Title]()* dialog.

Pass a semicolon separated list of allowed export IDs, see *LL_OPTIONSTR_-EXPORTS_AVAILABLE*

Example:

```
LlPrintStart(hJob,...,LL_PRINT_USERSELECT,...);

// allow only printer and preview (USERSELECT sets all bits)
LlSetOption(hJob, LL_OPTIONSTR_EXPORTS_ALLOWED, "PRN;PRV");
// Default should be preview!
LlPrintSetOption(hJob, LL_PRNOPT_EXPORT, "PRV");

// printer dialog allows user to change
LlPrintOptionsDialog(hJob,....);
// so get the final media:
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_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 path names of the files.

This list can be very large, so please allocate enough buffer and check the return value of *LlSetOption()* on the error value (*LL_ERR_BUFFERTOOSMALL*).

LL_OPTIONSTR_HELPFILENAME

You can use this function to force help filename, 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.

When the project is being sent by fax, these expressions will be evaluated and used directly 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 on the project in a special format to be used by other fax drivers. For details, please look into the manual of the fax software.

If these options are not set, and the user did not enter any expressions in the fax dialog, the „MS FAX" export will not be available.

LL_OPTIONSTR_LABEL_PRJDESCR

Use this parameter to set the description of the corresponding project type for the file dialogs.

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 „perslic.txt" and „redist.txt" in the List & Label installation directory.

LL_OPTIONSTR_LIST_PRJDESCR

Use this parameter to set the description of the corresponding project type for the file dialogs.

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, of the extension modules you want to use in your application.

By default, i.e. whenever opening a job or setting this option, the following extension modules are loaded automatically:

CMLL12OB.LLX
CMLL12PW.LLX
CMLL12HT.LLX
CMLL12EX.LLX
CMLL12OC.LLX

These files are loaded from the DLL's path

You can use Wildcards ("?", "*") to load multiple modules at once.

To suppress the loading of a default extension, pass its file name preceded by a "^", ex. "^CMLL12OB.LLX". To suppress all default extensions, pass "^*" as first "filename".

Use "^*;c:\windows\system\cmll12bc.llx" e.g. to only load the optional barcode extension module.

When this parameter is used for *LlGetOptionString()* you will get a list of available extension modules. (for example "CMLL12OB.LLX;CMLL12HT.LLX").

If the 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 for a NULL value at print time. Default value: empty("").

LL_OPTIONSTR_PREVIEWFILENAME

Can be used to preset the name for 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 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 give a translation table with the old and one or more of the replacement printers. You can do this by calling this function more than once or by issuing multiple definitions for the distinct 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 being used for the encryption. The password itself is not being stored in the project, so do not forget it!

In the designer you can store encrypted projects in unencrypted format 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 bits encryption. The password is not (!) absolutely secure as it is passed by an API. The hurdle for stealing project files is quite high however.

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 filename.

LL_OPTIONSTR_SHORTDATEFORMAT

The string used to convert a date in a string in:

```
date$(<Datum>, »%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 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. Within the Designer, the provided alias will be displayed instead of the field/variable name. When saving the file, only the original names will be stored. In this way, by supplying suitable alias names, the project can be localized. The option string needs to be set for each name that should be localized in the form "<alias>=<original name>", ex.

```
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 more easily. 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, to use for example user specific printer settings in a network environment.

Parameter:

hJob: List & Label job handle

pszDir: Pathname 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,
  DEVMODE* 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

nPrinter: printer index (0 for printer for page 1, 1 for printer for pages > 1, -1 for both at once)

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 which will be used for printing. If the printer configuration file does not exist, it will be created.

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.

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 Windows when the project is used the next time.

Parameter:

hJob: List & Label job handle

nObjType: *LL_PROJECT_LABEL*, *LL_PROJECT_CARD* or *LL_PROJECT_LIST*

lpszObjName: the file name of the project

Return Value:

Error code

Example:

```
HLLJOB    hJob;

hJob = LlJobOpen(0);

LlSetPrinterToDefault(hJob, LL_PROJECT_LIST, "test.lst");
if (LlPrintStart(hJob, LL_PROJECT_LIST, "test",
   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 real data preview.

Parameter:

hJob: List & Label job handle

nMenuID: ID of the button you want to remove. The IDs of the menu items in List & Label can be found in the file MENUID.TXT of 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	PDF 417 / Maxicode module

pszExtensionName: name of the extension ("HTML", "RTF", "PDF417", ...)

pszKey: the 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 of the respective module.

See also:

LIXSetParameter

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	PDF 417 / Maxicode module

pszExtensionName: name of the extension ("HTML", "RTF", "PDF417", ...)

pszKey: the name of the parameter

pszValue: the 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 of the respective module.

Using this function, you can pre-set some options of a module, for example the path of the output file, the print intensity of a barcode etc.

See also:

LIXGetParameter

12. Management of the print- and preview files

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 loss of quality.

All data is being stored in one file. Using the optional compression that you can switch on using

```
LlSetOption(hJob, LL_OPTION_COMPRESSSTORAGE, 1)
```

the file's size can be reduced about 3 to 10 times! Compression slows down the print process, but comes handy for example if you want to present data on the internet for download or preview using our OCX.

The file has the extension ".LL". We do not give away any information about its inner structure, and we recommend not relying on any detail you may find out! This is intentional as we have an own API to access the data contained in it, thus you have no advantage by peeking into the file, and we want to keep the freedom of changing the format whenever needed without having conflicts with existing software.

12.1. The Preview API

You don't have to care about the details of the preview files - the API functions *LlStgsysxxx()* do that for you.

All of these functions are exported by the C?LS12.DLL. This DLL, which you will usually distribute with external viewers, is as small as possible. If you want to use this DLL via an import library you need to link to the C?LS12.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.

Of course, the current storage 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 place marker %d (e.g. „page %d"). That's important e.g. for converting JPEG because without the place marker only one page will be the result.

pszFormat: Target format. Valid values are:

"TIFF" (also "PICTURE_MULTITIFF")
"JPEG" (also "PICTURE_JPEG")
"EMF"
"TTY"
"PDF"

With this parameter you have the possibility to declare a semicolon separated list with further export options. You will find the accepted values in the chapter "The Export Modules". An example could be the parameters "PDF;PDF.EncryptionFile=1". Additionally you can choose page ranges. An example for this could be the usage of

"PDF;Export.PageIndexRange.Min=2;Export.PageIndex.Max=3".

With this only the pages 2 and 3 converted to PDF.

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: Errorcode
= 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

Hints:

This function is a simple wrapper around *DeleteEnhMetaFile()*.

Example:

See LlStgsysGetPageMetafile

See also:

LlStgsysGetPageMetafile, LlStgsysGetPageMetafile16

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 in which to print (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 being 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 in which to print. 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 be 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:

LIStgsysPrint, LIStgsysStoragePrint

LlStgsysGetAPIVersion

Syntax:

```
int LlStgsysGetAPIVersion(HLLSTG hStg);
```

Task:

Returns the version of the Stgsys API.

Parameter:

hStg: The handle returned by *LIStgsysStorageOpen()*

Return value:

The version number of the Stgsys API in List & Label C?LL12.DLL and C?LS12.DLL

Hints:

The current version is 1 (as of 2006)

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:

LIStgsysGetFileVersion

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 *LIStgsysStorageOpen()* this path will also be included.

Parameter:

hStg: The handle returned by *LIStgsysStorageOpen()*

nJob: Job Index: 1: first Job,... (1..*LlStgsysGetJobCount()*)

nFile: Page number

Value	Meaning
-1	Management File
0	Printer configuration file
>0	Page Metafile of that page (1.. *LlStgsysGetPageCount()*)

lpszBuffer: initialized buffer for the filename

nBufSize: size of the buffer

Return value:

Error code

Hints:

Only previews of type *LL_STG_STORAGE* can have multiple jobs (by using *LlStgsysAppend()*).

The nFile Parameter distinguishes the type of file to return the name for.

In 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	Version number (currently 1, as of 5/2000)
Bits 8..15	Type: *LL_STG_COMPAT4* or *LL_STG_STORAGE*

Hints:

This call is also very important to find out about properties of the storage file and to be able to take care of possible differences.

See also:

LIStgsysGetAPIVersion

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 (1 if *LL_STG_COMPAT4*)
<0: Error code

Hints:

Only previews of type *LL_STG_STORAGE* can have multiple jobs (by using *LlStgsysAppend()*).

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 the project parameter chapter.

See also:

LlStgsysSetJobOptionStringEx

LlStgsysGetJobOptionValue

Syntax:

```
INT LlStgsysGetJobOptionValue(HLLSTG hStg, INT nOption);
```

Task:

Returns some numerical parameters of 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 want 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_HAS16BITPAGES

Returns whether the file contains 16 bit pages (1) or not (0). Important when working with older LL files of List & Label.

LS_OPTION_BOXTYPE

Returns the style of the meter box that has been used at the time of the preview print. (and which should be used during printing again). 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 (*LL_UNITS_MM_DIV_10* or *LL_UNITS_INCH_DIV_100*)

LS_OPTION_PRINTERCOUNT

Number of printers used:
1: printer for page 1 and the following pages are the same,
2: different printers have been chosen.

LS_OPTION_EMFRESOLUTION

Contains the value of the EMF resolution (standard is 100 for 1/100 mm), see *LlSetOption(LL_OPTION_EMFRESOLUTION)*.

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()* and *LlStgsysGetPageMetafile16()*)

See also:

LlStgsysGetPageMetafile, LlStgsysGetPageMetafile16

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 asked 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 a 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:

If the storage has been created with a 16 bit program, the function returns NULL. In this case, the application can request the handle via *LlStgsysGetPageMetafile16()*.

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:

LlStgsysGetPageMetafile16, 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_PRTNAME

Returns the name of the original printer (for example "PSCRIPT")

LS_OPTION_PRTDEVICE

Returns the device of the original printer (for example "HP LaserJet 4L")

LS_OPTION_PRTPORT

Returns the port of the original printer (for example "\\prntsrv\laser25")

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".

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 of it (0).

LS_OPTION_PRN_PIXELSOFFSET_X

Returns the horizontal offset of the printable area in respect to the paper edge (of the original printer).

LS_OPTION_PRN_PIXELSOFFSET_Y

Returns the vertical offset of the printable area in respect 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).

Return value:

>=0: Value
<0: Error code

Hints:

"Printer" or "original printer" means the printer that has been chosen during creation time of the preview file.

These values are invaluable if you want to create an own preview and print management, especially if the destination printers are different from the original.

To get the correct value, set the job with *LIStgsysSetJob()* before calling this API function.

See also:

LlStgsysGetPageOptionString

LlStgsysGetPagePrinter

Syntax:

```
INT LlStgsysGetPagePrinter(HLLSTG hStg, INT nPageIndex,
  LPTSTR pszDeviceName, UINT nDeviceNameSize, HGLOBAL* phDevmode);
```

Task:

Returns the printer and its settings that would be used for that 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 shall be stored. If NULL, the DEVMODE structure is not being 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

LlStgsysPrint

Syntax:

```
HLLSTG LlStgsysStoragePrint(HLLSTG hStg, LPCTSTR pszPrinterName1,
  LPCTSTR pszPrinterName2, INT nStartPageIndex, INT nEndPageIndex,
  INT nCopies, UINT nFlags, LPCSTR 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 print

nEndPageIndex: Index of the last page to print

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

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 *LlPrintStart()*) is being 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 chosen during creation). If no printer with the given device name is present, the default printer is chosen.

You need to set the job (*LlStgsysSetJob()*) before you call this function.

See Also:

LlStgsysPrint, LlStgsysSetJob

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 (if it was not opened READ-ONLY). Don't 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 or such into the storage file.

pszBuffer: Address of a 0-terminated string

Return value:

Error code

Hints:

Of course, the storage may not be opened with bReadOnly = TRUE!

Example:

```
LlStgsysSetJob(hStg, 1);
LlStgsysSetPageOption(hStg, 1, LS_OPTION_USER, "Letters A-B");
```

See also:

LlStgsysGetPageOptionValue

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. 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:

LlStgsysStorageOpen, LlStgsysConvert

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 care about 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 care about 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 others LlStgsys functions, 0 means error

Hints:

If you use a path for temporary data, this will be used as directory of the preview files, else the path of the project file name will be used. This convention is compatible with the calls to *LlPrint(<Project file>)* and *LlPreviewSetTempPath(<Temporary Path>).*

Note that the functions *LlStgsysAppend()* and *LlStgsysSetPageOptionString()* need the file to be opened with bReadOnly = FALSE!

bJobTranslation = TRUE is handy if you don't want to care about 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:

```
HLLSTG LlStgsysStoragePrint(LPCTSTR lpszFilename,
  LPCTSTR pszTempPath, LPCTSTR pszPrinterName1,
  LPCTSTR pszPrinterName2, INT nStartPageIndex,
  INT nEndPageIndex, INT nCopies, UINT nFlags,
  LPCSTR 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 care about 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 print

nEndPageIndex: Index of the last page to print

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 copies by the printer feature, if possible
LS_PRINTFLAG_METER	to show a meter dialog
LS_PRINTFLAG_-ABORTABLEMETER	to show a meter dialog which has a "Cancel" button

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 *LlPrintStart()*) 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 chosen during creation). If no printer with the given device name is present, the default printer is chosen.

See also:

LlStgsysPrint

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 CMMX01.DLL is used for sending export results by mail.

The settings will be saved in the registry at "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 here. The values will then be set automatically.

nFlags: One or a combination of the following flags:

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 specific – 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 has to be called after use of the email functions or during termination of 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: 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/html	Mail body text (HTML).
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:

-

13. The Export Modules

In addition to the output to preview file List & Label offers some other output formats.

These output formats can be created by some special export modules used by List & Label having the file extension .LLX (= List & Label Extension). This List & Label extension interface is designed to allow multiple output formats in one single extension file.

The standard output formats provided with List & Label are HTML, MHTML, RTF, PDF, XML, JPEG, EMF and BMP.

13.1. The Principle of the Modules

The export output formats ("HTML", "RTF", "PDF", "MHTML", "XML", "PICTURE_JPEG", "PICTURE_EMF", "PICTURE_BMP", "PICTURE_TIFF", "PICTURE_MULTITIFF", "XLS", "TTY", "TXT") can be used besides the standard output media printer ("PRN"), preview ("PRV") and file ("FILE"). The actual export to one of the new formats can be done analogous to the normal printing.

The output formats which are shown to the end user or which are used directly can be specified by your program.

13.2. The Programming Interface of the export modules

13.2.1. Global (De)activation of the Export Modules

List & Label by default tries to load the export extension module cmll12ex.llx from the main DLLs path. Thus, all export formats are 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. "^cmll12ex.llx". To load the module from a different path the same option may be used.

If you want to load the export modules from a different directory you also use this option. For example you can use "c:\programs\<your application>\cmll12ex.lx, to load the export modules from your application directory.

13.2.2. 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 semicolon. This list also contains the standard output formats "PRN", "PRV" and "FILE". By setting the option *LL_OPTIONSTR_EXPORTS_ALLOWED* the available export formats can be restricted. This setting concerns the available output formats in the dialog *LlPrintOptionsDialog()*. Please note, that the print destination parameter in *LlPrint[WithBox]Start()* takes influence on the export media as well. Therefore you should 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(...);
```

13.2.3. Selecting/Querying the output format

The selection/query of the output format can be done by a parameter of the methods *LlPrint[WithBox]Start()*. The following list shows the different values of this parameter:

Value	Meaning
LL_PRINT_NORMAL	Output format "Printer" will be default.
LL_PRINT_PREVIEW	Output format "Preview" will be default.
LL_PRINT_FILE	Output format "File" will be default.
LL_PRINT_EXPORT	An export module will be set as default output format. After that 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 be also the default output format in *Ll-PrintOptionsDialog().*

Example of how to set the output format to RTF:

```
...
LlPrintWithBoxStart(..., LL_PRINT_EXPORT, ...);
LlPrintSetOptionString(hJob, LL_PRNOPTSTR_EXPORT, "RTF");
LlPrintOptionsDialog(...);
```

If you want to prohibit the selection of the output format to the end user, you could use the option *LL_OPTIONSTR_EXPORTS_ALLOWED* to disable the other formats. Just specify the output format you want 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 pay regard to this fact determining the default output format directly or disabling this configuration opportunity in the designer. If not, your end user could be confused when selecting e.g. "RTF" in the designer, but then finds "HTML" as a default format for printing.

Example of how to pay regard to a possibly selected export media (if no selection was 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 possibility in the designer:

```
LlSetOption(hJob, LL_OPTION_SUPPORTS_PRNOPTSTR_EXPORT, TRUE);
//...
LlDefineLayout(...);
```

To determine which output format was selected by the end user in the *LlPrintOptionsDialog()* use this option.

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
}
```

13.2.4. 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.

A part 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.

13.2.5. Export Without User Interaction

An export without user interaction can be done very easily using the methods already mentioned .

Example:

If you want 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 it! (The meaning of the HTML-export specific options can be found in the following chapters)

13.2.6. Query 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 a 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-file.

The files created by the export will not be deleted automatically and should be deleted by your application.

13.3. HTML export module

The HTML export module creates (with some restrictions) HTML 4.01 code.

13.3.1. How the HTML export module works

The export module collects all List & Label objects of the page currently printed and places them in a big 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 if 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 the absolute positions is, that the result of the export is a more precise representation of the original layout (in the designer). The representation with percentage positions has the advantage that it is normally better printable than the other representations. This results from the fact that browsers can resize this kind of representation.

Because every 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 mostly be aligned to the same edges. This results in a less complex an faster loadable layout-grid.

The HTML 4.01 standard doesn't 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 realized in filling the cell (in the layout-grid) of the next object over the rectangle.

Besides, there are other restrictions set by the target format. The most important ones are listed in the next chapter.

13.3.2. Restrictions

- Overlapping objects (except rectangles) are not supported
- Rectangles can't 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 'Draw Separators Through' in table objects is not supported.
- The left offset in the first column of a table line will be ignored.
- The chart object is exported as picture and thus cannot appear transparently.
- The transformation of RTF text to HTML code is made by a RTF parser. This parser interprets the basic RTF formatting. Extended RTF functionality like embedded objects will be ignored.
- Rotated RTF text is not supported
- Horizontal and vertical lines are exported as images, all other lines are ignored
- Gradient fills are not supported
- Rotated text is not supported
- Custom drawings in callbacks are not supported
- Table frames of neighbor cells are not drawn overlapping each other but discrete. This can double the frame width and needs to be taken into account at design time already
- TotalPages$() may not be used in rotated text objects

If the HTML object is exported as HTML text, the part of the stream between the <body> and </body> tags will be embedded. This by concept leads to the following restrictions:

- Cascading Style Sheets are not all supported.
- Page formatting such as background color, margins etc. get lost.
- HTML does not allow scaling. Thus, the exported result may differ from the layout in the designer, especially when the HTML object holds 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 might get lost.

13.3.3. The 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..100*, with *100* representing the highest quality (least compression). Default: 100

Picture.BitsPerPixel: Specifies the color depth of the generated graphic. The value of 256 colors is normally enough for HTML. Please note, that values like 24 bit or higher can result in very large graphic files.

Value	Meaning
1:	Monochrome
4:	16 Colors
8:	256 Colors
24:	24bit True Color
32:	32bit True Color
Default:	8

Verbosity.Rectangle: Configures how rectangle objects should be exported.

Value	Meaning
0:	Ignore object
1:	Object as JPEG (and additionally 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 (ex. 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 not the hint on exporting restrictions.
Default:	2

HTML.Charset: Specifies the character set that will be used in the created HTML page. This value specifies how the browser interprets the characters in the page. If no charset was specified, this value will be determined automatically based on the project's codepage. In the following table you can see a list of codepages and their corresponding charset.

Value	Meaning
1250:	Charset will be set to ISO-8859-2 ("Latin-2").
1251:	Charset will be set to ISO-8859-5 ("Cyrillic").
1252:	Charset will be set to ISO-8859-1 ("Latin-1").
1253:	Charset will be set to ISO-8859-7 ("Greek").
1254:	Charset will be set to ISO-8859-9 ("Latin-5, Turk").
1255:	Charset will be set to ISO-8859-8 ("Hebraic").
1256:	Charset will be set to ISO-8859-6 ("Arabian").
1257:	Charset will be set to ISO-8859-4 ("Latin-4").
932:	Charset will be set to ISO-2022-JP ("Japanese").
936:	Charset will be set to GB2312 ("Chinese").
Default:	The value of the default-charset in the current LNG-File (current language).

HTML.Title: Specifies the title of the generated HTML-Document. Default: Title being used with *LIPrintWithBoxStart()*.

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 (ex. 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: Filename of the first HTML page. Default: "index.htm". You may also use printf format strings like "%08d" in the filename (ex. "Export Page %d.htm"). In this case, the files for the pages will be named by replacing the placeholder by the correctly formatted page number. Otherwise you'll 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 to export 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 if 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 check box in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default:	1

HTML.Form.Header: Defines a certain HTML-form-tag like "<form method="POST" action=...". If this option was specified all object names are analyzed for special tags used for the form extension (see chapter 'HTML-form creation'). Default: Empty, no form creation

HTML.Form.Footer: This option defines a certain HTML-form-tag showing the end of the form. Default: "</form>"

13.3.4. Hyperlinks

Hyperlinks can be embedded in text, table and RTF objects directly in the designer using the *Hyperlink()* function. In this way, dynamical hyperlinks and formulas can be realized.

Another way to create 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:http://www.combit.net" this would create a picture hyperlink when being exported to HTML.

13.3.5. HTML-form creation

The HTML module supports a special mode to create HTML forms. The module converts objects with certain names to input controls. This mode can be activated using the option *HTML.Form.Header*.

Example:

```
LlXSetParameter(hJob, LL_LLX_EXTENSIONTYPE_EXPORT, "HTML",
"HTML.Form.Header",
          "<form method="POST" action=http://www.xyz.de/x.cgi>");
```

The specified HTML-code will also be exported, and all object names will then be investigated for following values:

text objects

Value	Meaning
@EDIT:<Value>	Creates a single line edit control with <Value> as control name in the form. The contents of the first paragraph of the object will be the default text of the edit field.

@MULTIEDIT:<Value>	Creates a multi line edit control with <Value> as control name in the form. The contents of the first paragraph of the object will be the default text of the edit field.
@LISTBOX:<Value>	Creates a listbox control with <Value> as control name in the form. The contents of the paragraphs of the object will be the entries in the listbox.
@COMBOBOX:<Value>	Creates a combobox control with <Value> as control name in the form. The contents of the paragraphs of the object will be the entries in the list.
@RADIOBUTTON:<Group-name>,<Value>	Creates a radiobutton control. If the contents of the first paragraph contain 'T', 'X' or '1', the control will be checked.
@CHECKBOX:<Group-name>,<Value>	Creates a checkbox control. If the contents of the first paragraph contain 'T', 'X' or '1', the control will be checked.
@BUTTON:<Value>	Creates a button control. The contents of the first paragraph will be the button text.
@SUBMIT:<Value>	Creates a Submit-button control. The contents of the first paragraph will be the button text.
@RESET:<Value>	Creates a Reset-button control. The contents of the first paragraph will be the button text.

picture objects

Value	Meaning
@RADIOBUTTON:<Group-name>,<Value>	Creates a radiobutton control.
@CHECKBOX:<Group-name>,<Value>	Creates a checkbox control.
@IMAGE:<Value>	Creates an image control.

rectangle objects

Value	Meaning
@RADIOBUTTON:<Group-name>,<Value>	Creates a radiobutton control.
@CHECKBOX:<Group-name>,<Value>	Creates a checkbox control.

These form objects will be exported with the designed coordinates. Normal objects can have other coordinates based on their displayed data e.g. as text objects that were not filled completely. Some input controls cannot be placed very precisely on a certain position. Their size depends on the data they currently display. Therefore an 1:1 representation of List & Label to HTML is not possible.

Note: only entries which object size is large enough in order to be printed by List & Label can be exported. This means, that e.g.: if you have a text object that will become a combobox in the exported form, it must be large enough for all paragraphs to be printed.

13.4. MHTML export

The MHTML (Multi Mime HTML) export module works analogous to the HTML 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 per mail, as the recipient can open the file directly and doesn't need any access to further (external) picture files.

13.4.1. The programming Interface

All options of the HTML exporter are supported, pass "MHTML" as module name. The option Export.AllInOneFile will be ignored, as this format always results in one file only.

13.5. XML export

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.

13.5.1. The 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..100*, with *100* representing the highest quality (least compression). Default: 100

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: Specifies the color depth of the generated graphic. The value of 256 colors is normally enough for XML. Please note, that values like 24 bit or higher can result in very large graphic files.

Value	Meaning
1:	Monochrome
4:	16 Colors
8:	256 Colors
24:	24bit True Color
32:	32bit True Color
Default:	8

Verbosity.Rectangle: Configures how rectangle objects should be exported.

Value	Meaning
0:	Ignore object
1:	Complete object information and 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 (ex. chart object) should be exported.

Value	Meaning
0	Ignore object
1	Complete object information and object as JPEG
Default:	1

XML.Charset: Specifies the character set that will be used in the created XML pages. This value specifies how the browser interprets the characters in the page. If no charset was specified, this value will be determined automatically based on the project's codepage. In the following table you can see a list of codepages and their corresponding charset.

Value	Meaning
1250:	Charset will be set to ISO-8859-2 ("Latin-2").
1251:	Charset will be set to ISO-8859-5 ("Cyrillic").
1252:	Charset will be set to ISO-8859-1 ("Latin-1").
1253:	Charset will be set to ISO-8859-7 ("Greek").
1254:	Charset will be set to ISO-8859-9 ("Latin-5, Turk").
1255:	Charset will be set to ISO-8859-8 ("Hebraic").
1256:	Charset will be set to ISO-8859-6 ("Arabian").
1257:	Charset will be set to ISO-8859-4 ("Latin-4").
932:	Charset will be set to ISO-2022-JP ("Japanese").
936:	Charset will be set to GB2312 ("Chinese").
Default:	The value of the default-charset in the current LNG-File (current language).

XML.Title: Specifies the title of the generated XML-Document. Default: Title being used with *LlPrintWithBoxStart()*.

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: Filename of the first XML page. Default: "index.xml". You may also use printf format strings like "%08d" in the filename (ex. "Export Page %d.xml"). In this case, the files for the pages will be named by replacing the placeholder by the correctly formatted page number. Otherwise you'll 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 to export 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 if 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 check box 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

13.6. RTF export module

The RTF export module creates documents in the Rich Text Format based on the Microsoft specification 1.5/1.7. The exported files were mainly optimized for Word 2000 and Lotus Word Pro. Please note that the rendering of the exported files can differ with different word processors.

Besides others, the following hints and restrictions should be considered:

- The max. color depth is 24bit
- 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 be placed automatically on the next page.
- The option 'Draw Separators Through' 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 less 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
- Frames thinner than ½ pt (about 0.4 mm in List & Label) will not be displayed correctly.
- Frames with 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

13.6.1. The 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: Specifies the color depth of the generated graphic. A value of 256 colors is normally enough for RTF. Please note, that values like 24 Bit or higher can result in very large graphic files.

Value	Meaning
1:	Monochrome
4:	16 Colors
8:	256 Colors
24:	24bit True Color
Default:	8

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 (ex Word 97)
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 (ex Word 97)
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.LLXObject: Configures how LLX objects (ex. 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: Filename 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 to export without user interaction.

Value	Meaning
0:	With user interaction (dialogs)
1:	No dialogs or message boxes will be displayed (only if *Export.File* was specified).
Default:	0

Export.ShowResult: Specifies if 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 check box in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default:	1

13.7. PDF export module

The PDF export module creates documents in the Portable Document Format. This format can be viewed platform independent with the free Adobe Acrobat Reader®.

Besides others, the following hints and restrictions should be considered:

- Currently (2006) not all Unicode/Multibyte codepages are supported. Japanese and Chinese characters are normally exported as long as the corresponding Adobe Font Packages are provided . In the Designer it is important to set the charset of the text to the right value, i.e. there is no automatic unicode character/charset converting. Under certain circumstances right-to-left charsets in RTF objects are not accurately exported.
- Rotated boldface/italic true type fonts may have a slightly different width
- Fonts with a font width <> 0 are not supported
- Bitmap fonts can not be used/embedded
- Line ends are displayed with round caps
- Not all EMF records can be displayed accurately – if you are using complex EMFs you should pass them as bitmaps

13.7.1. 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. Default: title passed to *LIPrintWithBoxStart()*.

PDF.Subject: Specifies the subject of the generated PDF document. Default: empty.

PDF.Keywords: Specifies the keywords of the generated PDF document. Default: empty.

PDF.Encryption.EncryptFile: If this parameter is set, the result file will be encrypted. Different other options are enabled then which are described below.

Value	Meaning
0	Don't encrypt file
1	Encrypt file
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.Encryption.Level: Sets the encryption strength. Only effective if PDF.Encryption.EncryptFile is set to "1".

Value	Meaning
0	40 bit encryption
1	128 bit encryption (needs Acrobat Reader version 5 and higher)

Value	Meaning
Default	0

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 to always choosing a suitable password explicitly.

PDF.UserPassword: The user password for the encrypted file. This password is needed to access the encrypted file. If no password is given, the access is possible without password – however possibly limited (see above).

PDF.FontMode: Sets the TrueType embedding mode.

Value	Meaning
0	No embedding. The TrueType fonts on the target machine will be used if possible, or the font mapper will choose substitutes.
1	All TrueType fonts are embedded
2	Only symbol fonts are embedded
3	No embedding. All fonts will be replaced by PostScript standard fonts.
4	Subset embedding. Only the characters from the selected codepage are embedded.
5	Subset embedding. Only the used characters are embedded.
Default	5

PDF.ExcludedFonts: Allows passing a semicolon separated list of fonts that will never be embedded, regardless of the PDF-FontMode setting. Default: "Arial".

PDF.CompressStreamMethod: Sets the compression method for the PDF stream

Value	Meaning
0	No compression
1	Flate compression
2	Run-Length compression
3	FastFlate compression
Default	1

PDF.JPEGQuality: Sets the compression quality for the embedded images. Higher quality settings result in larger files, of course.

Value	Meaning
0	Minimum file size
1	25% quality
2	50% quality
3	75% quality
4	100% quality
5	No JPEG compression
Default	3

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: Filename of the document.

Export.Quiet: Use this option to configure the possibility to export 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 if 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 check box in the dialog.

Value	Meaning
0	Checkbox will be hidden
1	Checkbox will be available
Default:	1

13.8. Excel Export

The Excel export module creates Microsoft Excel® documents. The creation is independent of an 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.

The following restrictions 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 print out compared to List & Label. This effect can be minimized by setting a zoom for the print out (see XLS.PrintingZoom option).
- RTF texts are embedded as pictures. As this increases the file size remarkably, we recommend to use normal text wherever possible.
- Tabs will be replaced by blanks.
- The option 'Draw Separators Through' 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.
- Lines through text objects will be interrupted.
- Texts partially overlapping filled rectangles are filled completely.
- Overlapping text and picture objects are ignored.
- Lines which 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.
- 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

13.8.1. 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: Specifies the color depth of the generated graphic. A value of 256 colors is normally enough for XLS. Please note, that values like 24 Bit or higher can result in very large graphic files.

Value	Meaning
1:	Monochrome
4:	16 Colors
8:	256 Colors
24:	24bit True Color
Default:	8

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.

1	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 (ex. 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 the increased text height in Excel. Default: 89

XLS.PrintingZoom: Scaling factor for the print out of the project. Necessary in order to compensate the inability to place any objects in the non printable area. Default: 88(=88% zoom)

XLS.Title: Specifies the title of the generated XLS document. Default: title passed to *LlPrintWithBoxStart()*.

XLS.IgnoreGroupLines: Allows ignoring group header and footer lines 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 ignoring header and footer lines 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
Default:	1

XLS.IgnoreLinewrapForDataOnlyExport: Allows ignoring line wraps. 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 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 (Project > Options)
2	Only columns which actually contain a numeric values (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.
Default	1

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 for 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").

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: Filename of the document.

Export.Quiet: Use this option to configure the possibility to export 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 if 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 check box 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

13.9. Text Export

The text export module has two different modes (see Export.OnlyTableData). The "data only" mode exports data from table objects to a text format. Optionally the separator and framing character can be 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 a pure data conversion export.

Alternatively a text file can be created that resembles – as far as the format allows – the layout of the project. Please make sure to choose a font size that is large enough in the Designer. If the lines of text in your project can not be assigned to different lines in the text file, lines may be overwritten resulting in a loss of data in the result file.

13.9.1. 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 *LIXSetParameter(..."TXT"...)* and read by calling *LIXGetParameter(..."TXT"...).*

The verbosity settings only affect the "data only" mode.

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. If this option is empty, a file selection dialog will always be displayed.

Export.File: Filename of the document.

Export.Quiet: Use this option to configure the possibility to export 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 if 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 to hide the respective check box 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:	1

Export.AllInOneFile: Configures the export result format. Only effective in the layout preserving mode.

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

TXT.FrameChar: Specifies the framing character for the columns. Only effective in the data only mode.

Value	Meaning
NONE	No framing
"	" as framing character
‘	‘ as framing characters

TXT.SeparatorChar: Specifies the separator character. Only effective in the data only mode.

Value	Meaning
NONE	No separator
TAB	Tab as separator
BLANK	Blank as separator
,	, as separator
;	; as separator

TXT.IgnoreGroupLines: Allows ignoring group header and footer lines in the resulting text file. Only effective in the data only mode.

Value	Meaning
0	Group lines are exported
1	Group lines are ignored
Default:	1

TXT.IgnoreHeaderFooterLines: Allows ignoring header and footer lines in the resulting text file. Only effective in the data only mode.

Value	Meaning
0	Header and footer lines are exported
1	Header and footer lines are ignored
Default:	1

TXT.Charset: Specifies the character set of the result file. If Export.OnlyTableData is set to "0", the target codepage needs to be passed additionally (ex. 932 for Japanese) using LL_OPTION_CODEPAGE.

Value	Meaning
ANSI	Ansi character set
ASCII	Ascii character set
UNICODE	Unicode character set
Default:	ANSI

13.10. TTY Export

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.

13.10.1. 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 *LlXSetParameter(..."TTY"...)* and read by calling *LlXGetParameter(..."TTY"...).*

Export.Path: Path where the exported PRN file should be saved. If this option is empty, a file selection dialog will always be displayed.

Export.File: Filename of the PRN file.

Export.Quiet: Use this option to configure the possibility to export 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.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

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.

13.11. Picture Export

This module creates a graphics file (JPEG, BMP, EMF, TIFF, Multi-TIFF) for every printed page. The filenames of the created graphics will be enumerated. If the filename contains the format identifier "%d", this identifier will be replaced by the page number.

13.11.1. 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" or "PICTURE_MULTITIFF", 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..100*, with *100* representing the highest quality (worst compression). Default: 100

Picture.BitsPerPixel: Specifies the color depth of the generated graphic. A value of 256 colors is normally enough. Please note, that values like 24 Bit or higher can result in very large graphic files.

Value	Meaning
1:	Monochrome
4:	16 colors
8:	256 colors
24:	24bit True Color
32:	32bit True Color
Default:	8

Export.File: Filename 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'll get a simple page numbering for the result files.

Export.Path: Path where the exported files should be saved. If this option is empty, a file selection dialog will always be displayed.

Export.Quiet: Use this option to configure the possibility to export 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 if 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 to hide the respective check box 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.

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
Default:	None

TIFF.CompressionQuality: Specifies the compression quality for the TIFF export. Default: 75

13.12. MS Fax Export

You can send List & Label documents directly as fax using the fax service of Windows XP and 2000. If you connect a fax modem to such an operating system, the fax driver will be automatically installed.

Additional information is needed for an automatic fax sending (that is, no dialog for the fax destination, cover page etc. shall be displayed). 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 did not enter any expressions in the fax parameters dialog, the export to MS FAX will not be available.
This module has no programming interface.

Various accepted fax applications could be used from List & Label with the corresponding print(/fax) driver. If the fax application support the passing of the fax number by the document in most cases the number input dialog could be suppressed. For using 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 to the placed fax number. Other fax applications offers similar possibilities – we recommend taking a look at the documentation of your fax application.

13.13. Digitally Sign Export Results

By accessing the products digiSeal® office of secrypt GmbH or OPENLiMiT® SignCubes Software with license keys provided by the e•siqia Holding AG 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 require a card reader and a signature card with a certificate stored on it. Details on hard- and software requirements can be found in the signature provider's documentation.
digiSeal® office requires the digiSealAPI.dll. Windows NT or higher is mandatory for this package.
OPENLiMiT® SignCubes Software requires an interface DLL that is provided by the e•siqia Holding AG.

13.13.1. 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.

13.13.2. Programming Interface

The signature process can be influenced by a lot of 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 choosing the software to use if more than one of the supported products are installed.

Value	Meaning
0	Default, no explicit choice of signature software
1	Sign using secrypt digiSeal® office
2	Sign using OPENLiMiT® SignCubes Software
Default	0

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

13.14. Send Export Results via email

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.

13.14.1. Setting mail parameters by code

Like the other export options, some of the parameters for email dispatch 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 on project parameters for further details. A couple of other options can be set or read using *LlXSetParameter(..."<exporter name>"...) / LlXGetParameter(..."<exporter name>"...).* Note that exporter name may be left empty to address all export modules.

Export.SendAsMail: Activates the sending of export files via email. This option corresponds to the checkbox "Send exported files by email" for the end user.

Value	Meaning
1:	The exported files are sent via the specified provider (see below)
Default:	0

Export.SendAsMailAvailable: Enables to hide the respective check box 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 beside MSMAPI need the CMMX01.DLL.

Value	Meaning
SMAPI	Simple MAPI
XMAPI	Extended MAPI
SMTP	SMTP
MSMAPI	System MAPI (using the default MAPI client)
Default	The default value depends on the system's or application's settings (see below)

If the DLL can not be found, the mail will be sent using system MAPI (MSMAPI).

The provider is selected by either setting it explicitly using this option or letting the user choose in the *LsMailConfigurationDialog()*.

List & Label tries first to retrieve the application specific mail settings from the registry – if those can not be found, the global mail settings will be applied. These can be set using the control panel applet (CMMX01.CPL) or *LsMailConfigurationDialog()*.

Export.Mail.Body: Mail body text.

Export.Mail.AttachmentList: Additional attachments (besides the export results) as tab separated list ("\t", ASCII code 9).

Export.Mail.ShowDialog: Chooses to send 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 given)
1:	The standard "Send mail" dialog is displayed. The values passed are preset there.
Default:	0

Export.Mail.SendResultAs: Allows sending the result of an HTML export directly as HTML mail text.

Value	Meaning
text/html	If SMTP 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

The SMTP provider offers a set of additional options. Those generally don't need to be set explicitly but should be set in the *LsMailConfigurationDialog()*.

Export.Mail.SMTP.ServerTimeout: Socket timeout, in milliseconds

Export.Mail.SMTP.ServerAddress: SMTP server IP address or URL

Export.Mail.SMTP.ServerPort: SMTP server port

Export.Mail.SMTP.User: SMTP server user name (if necessary)

Export.Mail.SMTP.Password: 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

Export.Mail.SMTP.ProxyUser: Proxy user name (only Socks5)

Export.Mail.SMTP.ProxyPassword: Proxy password (only Socks5)

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 (combined from "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.

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 simple:

```
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 double quotes, as the passed parameter needs to be evaluated as formula:

```
LlSetDefaultProjectParameter(hJob,"LL.Mail.To", "'abc@xyz.de'", 0);
```

14. The Viewer-OCX-Control

14.1. Overview

The control CMLL12V.OCX can be used to view, export or print List & Label preview files in your own environment.

It can for example be inserted in your own application or in an internet page.

When printing, the projects will be fit into the page, automatically regarding the "physical page" flag and the page orientation to create the best printout result possible.

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

14.2. Registration

The control can be registered in the usual way, for example with the command "REGSVR32 CMLL12V.OCX", by the programming environment or by your setup program. It will not be usable without registration. See REDIST.TXT for more information.

14.3. Properties

AsyncDownload [in, out] BOOL: 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 effects 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 an 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 LLVIEW12.EXE and its menu items). So it's no problem at all to define 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.

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)).

14.4. 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: Zoom 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).

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.

SendTo: Starts "Send"

SaveAs: Starts "Save As"

GetOptionString([in] BSTR sOption) BSTR: Returns mail settings. See the chapter on project parameters for more information.

SetOptionString([in] BSTR sOption, [in] BSTR sValue) BSTR: Sets mail settings. See the chapter on project parameters for more information.

14.5. Events

BtnPress

Syntax:

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

Task:

Tells you that a 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:

-

14.6. 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.

14.7. Needed files

CMLL12V.OCX needs:

- CMLS12.DLL
- CMPR12.DLL
- CMLL12XL.DLL for direct PDF export functionality
- CMDW12.DLL for direct picture export functionality
- MFC 4.2 (MFC42.DLL, MSVCRT.DLL, MSVCRT20.DLL). Are often present on the user's computer. Keep in mind that the MFC42.DLL must be registered!

- WININET.DLL, URLMON.DLL
  Are only needed if Intra-/Internet URLs are used. If the computer supports internet download (for example if IE is already installed), usually these files are already on the system.
  Keep in mind that the URLMON.DLL must be registered!

14.8. CAB-Files packaging

A CAB file is shipped with List & Label.

14.9. 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. Take a look at l11vdemo.htm.

15. The Standalone Viewer Application LLVIEW

15.1. Overview

LLVIEW12.EXE is a powerful application to view and print 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 files with that extension, the viewer is being started.

15.2. Command Line Parameters

LLVIEW12 <file name>

Loads the file.

No URL can be given here (a result of MFC code deficiencies).

LLVIEW12 /p <file name>

Prints the file (with a printer dialog).

LLVIEW12 /pt <file name> <printer name>

Prints the file using the given printer. If the printer name contains spaces, enclose it in quotation marks.

15.3. Registration

The viewer registers itself when you start it. A setup program should call it with the "/regserver" option to prevent the viewer dialog to be opened.

Unregister using "/unregserver".

15.4. Needed files

LLVIEW12.EXE needs CMLS12.DLL and CMPR12.DLL. Depending on the required direct export functionality you also need:

- CMLL12XL.DLL if you want direct PDF export functionality
- CMDW12.DLL if you want direct picture export functionality

16. Error Codes

16.1. General error codes

Following are the return codes. They start with *LL_ERR_*, for example *BAD_JOBHANDLE* means *LL_ERR_BAD_JOBHANDLE*. The values in brackets are the decimal numbers that are written in the debug output.

Value	Meaning
BAD_JOBHANDLE (-1)	A function is called with a job handle not generated with *LlJobOpen()*.
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()*.
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.
PRINTING (-12)	An error occurred during print. Most frequent cause: print spooler is full
EXPORTING (-13)	An error occurred during print. (e.g. no access rights on the destination path, export file already existing and write protected,…)
NEEDS_VB (-14)	This DLL-version requires Visual Basic.
BAD_PRINTER (-15)	*LlPrintOptionsDialogTitle()*: no printer available.
NO_PREVIEWMODE (-16)	Preview functions: no preview-mode is set.
NO_PREVIEWFILES (-17)	*LlPreviewDisplay()*: no preview-files found.

PARAMETER (-18)	NULL pointer as a parameter is not allowed, possibly also other parameter errors. Please use the debug mode to determine the error.
BAD_EXPRESSION (-19)	New expression mode: an expression in *LIExprEvaluate()* 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 Debwin2. 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 with this object name exists.
NO_TABLEOBJECT (-28)	*LIPrint[WithBox]Start()*: No table in the list mode available.
NO_OBJECT (-29)	*LIPrint[WithBox]Start()*: the project has no objects, and empty pages can be printed in another way!
NO_TEXTOBJECT (-30)	*LIPrintGetTextCharsPrinted()*: No text object in this project.
UNKNOWN (-31)	*LIPrintIsVariableUsed()*, *LIPrintIsFieldUsed()*: the given variable does not exist. The project contains no information about used variables and fields because it was not yet 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 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()* was not defined.
UNKNOWNFIELD (-36)	The field given with *LIGetFieldType()* or *LIGetFieldContents()* was not defined.
UNKNOWNSORTORDER (-37)	The sorting order given by the ID for the grouping functions was not defined.
NOPRINTERCFG (-38)	*LIPrintCopyPrinterConfiguration()*: file not found or wrong format
SAVEPRINTERCFG (-39)	*LIPrintCopyPrinterConfiguration()*: File write error (network rights allow writing/creation, disk full?)
RESERVED (-40)	function not implemented yet
NOVALIDPAGES (-41)	16 bit StgAPI tries to read a 32 bit file
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 existent in the project.
TOO_MANY_CONCURRENT_PRINTJOBS	Only in server/webserver applications. The count of current users is beyond the number of licensed users.
BAD_WEBSERVER_LICENSE	Only in server/webserver applications. The webserver license file (*.wsl) is invalid or damaged.
NO_WEBSERVER_LICENSE	Only in server/webserver applications. The webserver license file (*.wsl) could not be found.
DRAWINGNOTFOUND(-53)	A required drawing file could not be found, See also LL_OPTION_ERR_ON_FILENOTFOUND

ERR_NOUSERINTERACTION (-54)	A call would require a user interaction, however the application is running on a web server
ERR_BADDATABASESTRUCTURE (-55)	The database structure at design and runtime does not match
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 CMLL11@@.LNG is not available.
NO_MEMORY (-102)	Not enough free memory.
EXCEPTION (-104)	A GPF happened inside a List & Label call. List & Label might be unstable.
LL_ERR_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.
LL_WRN_TABLECHANGE (-996))	In a hierarchical layout the table name is changing. See also chapter "4. Working with the Report Container".
LL_WRN_-PRINTFINISHED(-997))	Return value of LIRTFDisplay(): no more data to print
LL_WRN_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.

16.2. Additional Error Codes of the Storage-API

Following are the return codes of the StgAPI functions. They also start with *LL_ERR_*. The values in brackets are the decimal numbers that are written in the debug output.

Value	Meaning
STG_NOSTORAGE (-1000)	The file is not a List & Label preview file
STG_BADVERSION (-1001)	The version number of the preview file is incompatible
STG_READ (-1002)	Error reading preview file
STG_WRITE (-1003)	Error writing preview file
STG_UNKNOWNSYSTEM (-1004)	Unknown preview file format for the storage system
STG_BADHANDLE (-1005)	Bad parameter (invalid metafile handle)

STG_ENDOFLIST (-1006)	*LIStgsysGetFilename()*: page not found
STG_BADJOB (-1007)	*LIStgsysxxx()*: job not found
STG_ACCESSDENIED (-1008)	Storage has been opened with ReadOnly flag and cannot permit write access
STG_BADSTORAGE (-1009)	internal error in preview file, or empty file
STG_CANNOTGETMETAFILE (-1010)	*LIStgsysDrawPage()*: metafile could not be created (possibly defective preview file)
WRN_STG_UNFAXED_PAGES (-1100)	*LIStgsysPrint() and LIStgsysStoragePrint()* (only while printing to MS FAX): Some pages did not include a phone number and could not be faxed.

17. DEBWIN2.EXE

17.1. Basics

DEBWIN2 is a tool for manifold debug functions.

If the debug mode is switched on with *LlSetDebug()* List & Label outputs status information on the DEBWIN2 window.

As soon as your application is started it will start sending debug messages to Debwin2.

Besides the error codes (see chapter "16. Error Codes") you often get more information that helps to trace the reasons for unexpected program behavior. A typical debug output looks like this:

```
CMLL12  : 09:05:01.266 00000ee4  [lldemo32.exe] LlSelectFileDlgTitleEx
(1,0x00010a3a,'(NULL)',0x00008001,0x0012F86C,129,0x00000000)
CMLL12  : 09:05:19.726 00000ee4  ->'c:\templates\article.lbl'=0
```

You see the called module (CMLL12), timing information, the current thread ID, the caller (lldemo32.exe), 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 a load of such output lines. If you don't get the output, normally

- you haven't activated the debug mode with *LlSetDebug()* or
- you haven't activated logging in DEBWIN2

DEBWIN2 has a line ring buffer in which you can skip forwards and backwards as you choose. This is possible with the usual cursor keys or the scroll bars.

Further information can be obtained from the file DEBWIN.TXT.

17.2. Support

We do not offer support for this additional tool.
It is purely a debugging tool!

18. Appendix: Barcode Formats

The barcode formats can be found in the online help, topic "barcode formats".

19. Appendix: Configuration Files

List & Label stores its information in the following files:

- Interface configuration (window size, default-colors, default-font, ...): in the registry under HKEY_CURRENT_USER/Software/combit/CMBTLL/<application name>.
- Label and list definitions: in a file with the desired file name and the default extension '.lbl' for labels, '.lst' for lists and '.crd' for file cards. The files are text-oriented and can be changed with any text editor (NOTEPAD or similar) if necessary. The strict syntax has to be complied with. For a project file written by the SBCS/DBCS version of List & Label, using the API-call GetPrivateProfileString ("Description", "Text", ..., <file name>) the descriptive text of the project files can be obtained. Please read the definition of the format in appendix 2.
- The project file written by the Unicode version of List & Label is in the Unicode format and has the Unicode Magic as first WORD.
- The file extensions can differ from the settings when using the function *LlSetFileExtensions()* or *LlSetOptionString()*.
- Printer configuration: in a file with the label or list name and the extension '.lbp' for labels, '.lsp' for lists and '.crp' for file cards. The information is stored in binary and should not be changed in any case. The file extensions can differ from these defaults when using the function *LlSetFileExtensions()* or *LlSetOptionString()*. The directory of this file can be set by *LlSetPrinterDefaultsDir()*. The printers themselves can be switched by *LlSetPrinterInPrinterFile()*.
- Also stored in this file are the parameters for the export modules.
- The preview files for the file selection box have the extensions '.lbv', '.lsv' and '.crv'. These are windows-usual bitmaps. The file extensions can differ from these defaults when using the function *LlSetFileExtensions()*.

The preview files for the preview are described in the chapter 'Management of the print- and preview files'.

20. Appendix: File Formats

Information on the format of the project files of List & Label can be downloaded from our online knowledgebase http://support.combit.net.

21. Appendix: What's new in Version 5.0

21.1. Summary

21.1.1. General

- Module names changed from LX to L5
- Preview print to one file which can be sent. This is now default!
- The preview management file has been renamed (from ".000" to ".LL")
- STGAPI: functions to manage this preview file
- SendTo mail function in preview
- Optional compression of the preview file
- small changes in the format of the metafiles (no SetViewportOrg() any more).
- 32-bit-Preview-ActiveX/OCX/COM/OLE-Control.
- Independent, small, license-free 32-bit-Preview-Viewer, uses the OCX mentioned above.
- LL_CMND_HOSTPRINTER
- New Barcodes MSI-Plessey, CODE93, CODE11.
- New graphics formats JPEG, PCD.
- New options for LISetOption() and LIGetOption().
- Combination of preview jobs into one file possible
- Initial preview zoom factor and window size
- New expression mode is default

21.1.2. Designer

- RTF Control (32 Bit only)
- New dialog style for Office 97 look-alike Buttons
- Group footers
- Optional page break on first group header line
- Table object extended to 5 head, 20 footer, group and 30 data lines
- Wizard mode for new label and file-card projects
- Import of projects possible

21.2. New Functions

21.2.1. DLL-API

- Stgsys-Functions to manage the preview file(s)
- LIPrintCopyPrinterConfiguration() to manage the printer configuration files, for example for temporary printer selection
- LIPreviewDisplayEx

21.3. The Expression Mode

The old expression mode is no longer supported in List & Label.

22. Appendix: What's new in Version 6.0

22.1. Summary

22.1.1. General

- The names of the modules (EXE, DLL, OCX, VBX, LNG) changed according to the version number. The same applies to the additional DLLs, please refer to the chapter "DLLs Required for the Program" at the beginning of this manual.
- The OCX Control has been extended:
- encapsulates nearly all functions of the List & Label DLL
  - contains a routine which implements the print loop for most kinds of print jobs
  - the function naming convention is now compatible to List & Label and the VCL control
- The VCL Control has been extended:
  - encapsulates all functions of the List & Label DLL
  -contains a routine which implements the print loop for most kinds of print jobs
  - the function naming convention is now compatible to List & Label and the OCX control
- Automatic page wrap for tables and text objects, even for objects linked to tables
- The RTF Object can wrap to multiple pages (32bit)
- Hierarchical variable and field lists ("Address.Name", "Address.Street", ...)
- RTF texts also in tables
- New variable type *LL_RTF* to select directly in the RTF object's property dialog
- Date variables and fields can contain the time (as fraction of the day in the fractional part)
- Date variables and fields can be passed without offset calculation from Delphi, Basic and C
- Extended object link options (negative size fit, link to end) simplify design and programmatic approach for multi-page projects
- New output option: Print to file
- New or modified expression functions: *Continued(), Now(), AddDays(), AddHours(), AddMinutes(), AddSeconds(), Hour(), Minute(), Second(), RGB(), Min(), Max(),* extended parameters for *Date$()*, extended parameters for *FStr$()*, extended parameters for *WoY()*
- Some internal variables/fields have been renamed:
  - CountPageData -> LL.CountDataThisPage
  - CountAllData -> LL.CountData
  - CountPagePrintedData -> LL.CountPrintedDataThisPage
  - CountAllPrintedRows -> LL.CountPrintedData
  - FCountPageData -> LL.FCountDataThisPage
  - FCountAllData -> LL.FCountData
  - FCountPagePrintedData -> LL.FCountPrintedDataThisPage
  - FCountAllPrintedRows -> LL.FCountPrintedData

  (for compatibility, the old names are still recognized, yet slower)

- Print Dialog
  - improved design
  - optional: User can choose the type of output media (Printer/File/Preview)
- Barcodes can be printed on pages larger than A4/letter
- Preview can be used for pages larger than A4/letter using a workaround to Win95/98 limitation (32bit) (see *LL_OPTION_EMFRESOLUTION*)
- Optional history of changes in project file: user and computer name (32 bit), date and time information of creation and last change
- Optional history of changes in preview file: user, computer name, date and time information of creation (32 bit)
- Barcode and drawing object contents can be calculated by an expression

22.1.2. Workspace

- Optional: new "Explorer"-Style in FileOpen/SaveAs dialogs
- Several improved dialogs
- Help button has new design
- New controls for color and pattern selection
- New dialog style for Office 98 "sliding" tooltips
- Extended expression wizard:
  - multi-line
  - 32 bit: Drag&Drop from variable list and variable window
  - automatic insertion with an operator if needed
  - functions to manage bracket levels to support development of complex formulas
- Hierarchical variable list in separate window for Drag&Drop (32 bit)
- onto workspace: creation of new objects
  - onto objects: insertion of new paragraph, table field, change of contents (depends on object type)
  - on text object property dialog: insertion of a paragraph
  - on table object dialog: insertion of a field
  - on expression wizard: insertion of the variable in the formula
- The right mouse button menu has been expanded:
  - editing of a single table field
  - editing of a single text paragraph
  - editing of a formula in a barcode or drawing object
- Modification of the cell width of single or multiple table fields using the mouse
- Insertion of fields into a table per Drag&Drop
- Barcodes (EAN8, EAN13xxx, DPIdent, DPLeit, GermanParcel, Postnet, FIM) can restrict the resizing to their respective allowed sizes.
- New functions in the preview:
  - Save a single page (32 bit, Storage only)
  - Save the whole preview (32 bit, Storage only)
  - Print a single page to a selectable printer
  - Print the whole preview to a selectable printer

- Enhanced LLVIEW7 (the independent viewer for preview files)
  - Thumbnail-style page preview
  - dual language with auto-detection of the language, can also be selected manually

22.2. New Functions and Options

22.2.1. DLL-API

- *LlSetPrinterInPrinterFile()*
- *LlPrintSetOptionString()*
- *LlPrintGetOptionString()*
- *LlDebugOutput()*
- New flags for *LlPrintXXXStart()*: *LL_PRINT_FILE*, *LL_PRINT_USERSELECT*
- *LlSelectFileDlgTitleEx()* for *LlSelectFileDlgTitle()* and *LlSelectFileDlg()*: parameters changed
- *LlPrintSelectOffsetEx()* for *LlPrintSelectOffset()*: parameters changed
- New options:
  LL_OPTION_USEBARCODESIZES
  LL_OPTION_MAXRTFVERSION
  LL_OPTION_AUTOMULTIPAGE
  LL_OPTION_EMFRESOLUTION
  LL_OPTION_SETCREATIONINFO
  LL_OPTION_XLATVARNAMES
  LL_OPTION_SPACEOPTIMIZATION
  LL_OPTION_FORCEMBCS
  LL_OPTION_VARSCASESENSITIVE
  LL_OPTION_DELAYTABLEHEADER
  LL_OPTION_OFNDIALOG_EXPLORER
  LL_OPTION_LANGUAGE
  LL_OPTION_PHANTOMSPACEREPRESENTATIONCODE
  LL_OPTION_LOCKNEXTCHARREPRESENTATIONCODE
  LL_OPTION_EXPRSEPREPRESENTATIONCODE
  LL_OPTION_DEFPRINTERINSTALLED
  LL_OPTION_PRINTDLG_DESTMASK
  LL_OPTION_PRINTDLG_DEST
  LL_OPTION_PRINTDLG_ONLYPRINTERCOPIES
- Modified options:
  LL_OPTION_SUPPORTPAGEBREAK defaults to TRUE
  LL_OPTION_WIZ_FILENEW: can be set by the user in the designer, thus the option does not do anything any more.
- New variable/field types and subtypes:
  LL_RTF
  LL_DATE_DELPHI1
  LL_DATE_DELPHI

LL_DATE_MS
LL_DATE_OLE
LL_DATE_VFOXPRO

22.2.2. STGAPI

- New options for *LIStgsysGetJobOptionValue()*: *LL_STGSYS_OPTION_ISSTORAGE, LL_STGSYS_OPTION_EMFRESOLUTION*
- New options for *LIStgsysGetPageOptionString()*: *LL_STGSYS_OPTION_CREATION*, *LL_STGSYS_OPTION_CREATIONAPP*, *LL_STGSYS_OPTION_CREATIONDLL*, *LL_STGSYS_OPTION_CREATIONUSER*

22.3. Concept Changes

The following functions are not supported any more (they might be included in the declaration file, but with no real functionality): *LIPrintCheckLineFit(), LIPrintBeginGroup(), LIPrintEndGroup(), LIVB...(), LIPrintIntermediateTitle(), LIPrintHSeparator().*

The print logic has changed: Since version 6 List & Label handles the complete page breaks by its own. Therefore we cannot guarantee that all constructions that might have been built with List & Label 5 will work correctly in any case. Especially the following tasks should be solved in another way.

- Use of *Page()* in changing conditions for groupings to generate an intermediate title. Here problems might occur, when a data line causes a page break.
- line height of 0 pt for statistics printout (take a look to the new designer option since version 7).

Other items to take care:

- *LIPrintGetRemainingItemsPerTable()* writes the data currently available in a buffer, so after calling this functions data cannot be changed until the line has been printed.
- You may have to adjust paragraph or line spacing.

23. Appendix: What's new in Version 7.0

23.1. Summary

23.1.1. General

- List & Label 7.0 is sold in two flavors: the standard SBCS version for Western codepages, which can optionally be enhanced to handle far east languages and a Unicode version (please refer to the chapter about international use). All of them are capable of automatic word break according to the font's script language.
- Export to RTF, HTML, DIB und JPG
- more integrated Barcodes:
  - 4 new MSI-Barcodes
  - MSI-Barcodes now also handle the full 0-9,A-F character range
- optional module with the 2D Barcodes PDF417 and Maxicode
- enhanced VCL component for Delphi:
  - integrated data binding with Master-Detail-Mode
  - Property- and component editor (Designers can be called from within the IDE)
- enhanced OCX Control:
  - Intellisense (if supported from the IDE)
  - List & Label-constants as ENUMs
- additional VB-Control
  - with source code
  - integrated data binding
- Text objects can be bottom aligned
- Pictures can be embedded in the project file
- The RTF object supports embedded objects and pictures
- Extended to support 50 table header, data, footer and group header and footer lines
- Virtually unlimited number of sum and user variables which can have names defined in the designer
- New or modified formula functions: *LocCurr$()*, *LocCurrL$()*, *LocNumber$()*, *LocDate$()*, *LocTime$()*, *Locale$()*. New parameter for *Date$()* and *FStr$().*
- Print options dialog
  - visually enhanced
  - enhanced selection of output medium, default output medium can be preselected in the designer
- File format of the label templates has changed
- A project file that is loaded in the designer is locked against write access.
- While printing, the paper size is set to the project's page size if the printer is not set correctly (if the printer in the P-file is different or there is no P-file).

23.1.2. User Interface

- Color marks to better distinguish the different types of table lines, transparent on Win2000/98 systems, when MSIMG32.DLL is present.

- Layer, variables and preview window are hidden when the designer is not active
- Enhanced selection frame to size or move a frame
- Enhanced formula wizard with three function groupings (function type, type of return value and type of first parameter) to help finding a function.
- Text objects:
  - Font setting is faster
  - Bottom aligned option
  - Line distance in a paragraph can be selected
- new LLVIEW7
  - independent of the viewer OCX, easier registration
  - more languages

23.2. New Functions/Options

23.2.1. DLL-API

- New functions:
  LlJobOpenLCID()
  LlPrintResetObjectStates()
  LlPrintResetProjectState()
  LlXSetParameter()
  LlXGetParameter ()
- Deleted functions:
  LlPrintSelectFont() (Font entries in structures are now handles)
  LlSetDecimalChar() (implemented as OPTION)
- New options:
  LL_OPTION_LCID
  LL_OPTION_CODEPAGE
  LL_OPTION_DEFDEFFONT
  LL_OPTION_NOTIFICATIONMESSAGEHWND
  LL_OPTION_IMMEDIATELASTPAGE
  LL_OPTION_COMPRESSRTF
  LL_OPTION_ALLOW_LLX_EXPORTERS
  LL_OPTION_SUPPORTS_PRNOPTSTR_EXPORT
  LL_OPTION_FORCEFONTCHARSET
  LL_OPTION_CALCSUMVARSONINVISIBLELINES
  LL_OPTION_NOFOOTERPAGEWRAP
  LL_OPTIONSTR_EXPORTS_AVAILABLE
  LL_OPTIONSTR_EXPORTS_ALLOWED
  LL_OPTIONSTR_DEFDEFFONT
  LL_OPTIONSTR_EXPORTFILELIST
  LL_OPTIONSTR_LLXPATHLIST
  LL_OPTIONSTR_DECIMAL
  LL_OPTIONSTR_THOUSAND
  LL_OPTIONSTR_CURRENCY
  LL_OPTIONSTR_SHORTDATEFORMAT

LL_PRNOPT_JOBID
LL_PRNOPTSTR_EXPORT
LL_PRNOPTSTR_EXPORTDESCR
LL_PRNOPT_PRINTJOBNAME

- Modified options:
  LL_OPTION_MULTIPLETABLELINES Defaults to TRUE
  LL_OPTION_TEXTQUOTEREPRESENTATIONCODE Defaults to 1
  LL_OPTION_PHANTOMSPACEREPRESENTATIONCODE Defaults to 2
  LL_OPTION_LOCKNEXTCHARREPRESENTATIONCODE Defaults to 3
- Deleted options:
  LL_OPTION_FORCEMBCS (now set by ..._*CODEPAGE*)
- New callbacks:
  LL_CMND_CHANGE_DCPROPERTIES_CREATE
  LL_CMND_CHANGE_DCPROPERTIES_DOC
  LL_CMND_CHANGE_DCPROPERTIES_PAGE
  LL_NTFY_EXPRERROR
- New variable and field types:
  LL_DATE_DMY
  LL_DATE_YMD
  LL_DATE_MDY
  LL_DATE_YYYYMMDD
  LL_BARCODE_MSI_10_CD
  LL_BARCODE_MSI_10_10
  LL_BARCODE_MSI_11_10
  LL_BARCODE_PDF417 (optional module needed)
  LL_BARCODE_MAXICODE (optional module needed)
  LL_BARCODE_MAXICODE_UPS (optional module needed)
- New hostprinter commands:
  LL_PRN_GET_PAPERFORMAT
  LL_PRN_SET_PAPERFORMAT
- Structures modified:
  scLlPrinter

23.2.2. STGAPI

- New functions:
  LlStgsysPrint()
  LlStgsysPrintStorage()

24. Appendix: What's new in Version 8.0

The old expression mode is no longer supported.

24.1. Summary

24.1.1. General

- new object types: HTML, Chart
- new export formats: PDF, XML, MHTML
- new variable types LL_HTML , LL_NUMERIC_LOCALIZED and LL_DATE_LOCALIZED
- the if() and cond() functions have been optimized
- new internal designer variable: "LL.OutputDevice". Contains the medium the project is being printed to.
- new designer function *Hyperlink$()*
- *Empty()* function optionally applies implicit *ATrim$()*
- the names of table line definitions can be freely changed
- each group header may trigger a page break
- the text object can have a background color
- optional variation of character spacing in block justified text
- optimized preview file format, decreased file size
- automatic backup copy when saving files in the designer
- new compatibility option enabling Win NT generated files containing barcodes to be read on Win9x/ME machines.
- better resolution for inch system.
- changed loading behavior of List & Label extension modules, see LL_OPTIONSTR_-LLXPATHLIST
- support for new Windows 2000 file dialog styles
- project sketch files can be colored, default is monochrome. See LL_OPTION_-SKETCH_COLORDEPTH
- possible table line definitions increased to 100 per type
- the project file description in the file dialogs can be changed (LL_OPTIONSTR_-xxx_PRJDESCR)
- optionally: Japanese postal barcode (CM32L8BJ.LLX)
- optionally: Euro currency translation functions (CM32L8EU.LLX)
- optionally: new bitmap effect functions (Mask, Rotation, Black/White, emboss) (CM32L8BM.LLX)
- the CM32CR8.dll is no longer required for print-only projects.
- the new coordinate system LL_INCH_DIV_1000 might cause your user objects to be able to deal with it. *LL_OPTION_UNITS* returns the current resolution, *LL_OPTION_METRICS* only returns whether the system is metric or not. Thus, please take a look at your usage of these options. The maximum paper size in the inch/1000 resolution is restricted to approx. 83 cm on Win9x/ME systems.

24.1.2. Designer

- the designer interface has been redesigned for even more user friendliness. In addition to the preview window, new designer modes allow for an editable preview.
- the zoom modes of workspace and preview window are changeable independently of each other.
- the formula wizard has been equipped with many new features as auto completion, tool tips, parameter hint and syntax highlighting (not for MBCS/Unicode Version)

24.2. New Functions/Options

24.2.1. DLL-API

- New functions:
  LlDesignerProhibitFunction()
  LlSetNotificationCallbackExt()
  LlDefineChartFieldStart()
  LlDefineChartFieldExt()
  LlPrintDeclareChartRow()
  LlPrintGetChartObjectCount()
  LlPrintIsChartFieldUsed()
  LlGetChartFieldContents()
  LlEnumGetFirstChartField()

- New options:
  LL_OPTION_INTERCHARSPACING
  LL_OPTION_ALLOWMENUMANAGER
  LL_OPTION_INCLUDEFONTDESCENT
  LL_OPTION_RESOLUTIONCOMPATIBLETO9X
  LL_OPTION_OFNDIALOG_NOPLACESBAR
  LL_OPTION_SKETCH_COLORDEPTH
  LL_OPTION_SKIPRETURNATENDOFRTF
  LL_OPTIONSTR_MAILTO
  LL_OPTIONSTR_MAILTO_CC
  LL_OPTIONSTR_MAILTO_BCC
  LL_OPTIONSTR_MAILTO_SUBJECT
  LL_OPTIONSTR_SAVEAS_PATH
  LL_OPTIONSTR_LABEL_PRJDESCR
  LL_OPTIONSTR_CARD_PRJDESCR
  LL_OPTIONSTR_LIST_PRJDESCR
  LL_OPTIONSTR_LLFILEDESCR

- Changed options:

LL_OPTIONSTR_LLXPATHLIST new logic
LL_OPTION_MAXRTFVERSION = 0 prevents List & Label from loading RTF controls
LL_OPTION_UNITS: new 1/1000 inch unit

- New variable and field types:
  LL_DATE_LOCALIZED
  LL_NUMERIC_LOCALIZED
  LL_HTML

25. Appendix: What's new in Version 9.0

25.1. Summary

25.1.1. General

- optional encryption of project files
- changes in licensing (license structure itself, and setting the license by an API call)
- new object type: OLE Container
- supports page ranges and copies when printing from the preview
- thumbnail view in the preview
- new export format: Excel
- new export format: Text
- new export format: TIFF/Multi-TIFF
- group header lines can be forced to print directly below the header line
- group footer lines can force a page wrap below them
- nearly all object properties allow expressions
- new internal LL variables, for example to allow object placement depending on the paper size or printable area
- new tool windows for object list and object properties
- standalone RTF Editor (new group of API calls)
- rotation of RTF objects (Windows NT/2000/XP)
- RTF objects support text overflow in other RTF objects
- support of MS FAX as export medium (Windows 2000/XP)
- reduced size of the preview files up to 60%
- enhanced .NET component, including user-definable functions and objects
- new event for print and design methods of the OCX and VCL. Print options can be set there, for example to cause an export without user interaction.

25.1.2. User Interface

- Office XP Look & Feel
- Dockable tool windows and toolbars, Tabstrips for multiple docked windows
- help lines in the designer with configurable snap width
- table object: line definitions can now easily be exchanged and compressed
- property window

25.2. New Functions/Options

25.2.1. DLL-API

- New functions:

LIRTF...() (API calls for standalone RTF object)
LIPrintGetRemainingSpace()
LIGetPrinterFromPrinterFile()

- New options:
  LL_OPTION_UISTYLE
  LL_OPTION_NOFILEVERSIONUPDATEWARNING
  LL_OPTION_FONTPRECISION
  LL_OPTION_FONTQUALITY
  LL_OPTIONSTR_FAX...
  LL_OPTIONSTR_PROJECTPASSWORD
  LL_OPTIONSTR_LICENSINGINFO

26. Appendix: What's new in Version 10.0

26.1. Summary

26.1.1. General

- New designer object: form control (edit control, combobox, checkbox or button)
- Totally renewed PDF export (font embedding, faster, less restrictions)
- Preview control: direct export to PDF and other formats
- Preview control: send form data via HTTP or email in different formats
- Preview control: thumbnail bar switchable
- Emails can be sent via SMTP or MAPI, parameters can be set by code
- New preview controls (OCX/VCL/.NET)
- Stand alone RTF inplace object, insertable into own projects
- New export type: TTY; sends text directly to dot-matrix printers
- Extended text export with layout
- Optional frame for text object
- Frames in tables can be assigned per cell
- New barcode formats (Data Matrix, PZN, Code 39 CRC, PIX, Royal Mail)
- Callback for print job supervision
- New designer functions "RegExMatch$()", "Previous()"
- Project includes
- Stgsys-API only in LS-DLL
- New naming convention for DLL names

26.1.2. User interface

- Optional Office 2003 Look & Feel
- More sizeable dialogs
- Line objects can be inserted horizontal and vertical by using the Shift-Key
- RTF-Editor supports zoom (requires at least RTF Version 3)

26.2. New Functions/Options

26.2.1. DLL-API

- New functions:
  CMLL10.DLL:
  LIRTFEditorProhibitAction()
  LIRTFEditorInvokeAction()
  LISetDefaultProjectParameter()
  LIGetDefaultProjectParameter()
  LIPrintSetProjectParameter()
  LIPrintGetProjectParameter()

CMLS10.DLL:
LsSetDebug()
LsMailConfigurationDialog()
LIStgsysConvert()
LIStgsysGetJobOptionStringEx()
LIStgsysSetJobOptionStringEx()
LIStgsysGetPageOptionStringEx()
LIStgsysSetPageOptionStringEx()
LIStgsysStorageConvert()

- New options:
  LL_OPTION_ESC_CLOSES_PREVIEW
  LL_OPTIONSTR_PREVIEWFILENAME

- New callback:
  LL_INFO_PRINTJOBSUPERVISION

26.3. Changed Functions/Options

26.3.1. DLL-API

- The entire Stgsys-API now resides in the CMLS10.DLL and was removed from the CMLL10.DLL
- The constants LL_STGSYS_... have been renamed to LS_... (ex.: LL_STGSYS_OPTION_CREATION -> LS_OPTION_CREATION)
- LIRTFEditObject has an additional parameter that switches the modal mode
- The recommended way to send export results by mail is using the project parameter API. Find details in the chapter "13.14. Send Export Results via email".

27. Appendix: What's new in Version 11.0

27.1. Summary

27.1.1. General

- API support for relational data structures and hierarchical reports
- Designer: Aggregate functions
- NULL-value support
- .NET: completely rewritten data binding
- .NET: extensive new online help
- VCL: completely rewritten data binding
- Support for page x/y without 2-pass technique
- Improved Excel-Export
- Improved RTF-Export
- Conditional pagewrap before group header / after group footer lines
- Rotatable table columns
- New print option dialog options
- New barcode: Aztec

27.1.2. User interface

- New tool window "Table structure"
- Visibility of the table column separators in the rulers for an easier adjustment
- Advanced line definition dialog: it is possible to hide single line definitions
- New toolbar controls in the preview
- Redesigned treeview icons
- Property dialogs for frames and formattings

27.2. New Functions/Options

27.2.1. DLL-API

- New functions:
  CMLL11.DLL:
  LIDbAddTable()
  LIDbAddTableRelation()
  LIDbAddTableSortOrder()
  LIDbSetMasterTable()
  LIPrintDbGetCurrentTable()
  LIPrintDbGetCurrentRelation()
  LIPrintDbGetCurrentSortOrder()
  LPrintIDbGetRootTableCount()
  LIGetUsedIdentifiers()

  CMLS11.DLL:

LsMaiJobOpen()
LsMailJobClose()
LsMailSetOptionString()
LsMailGetOptionString()
LsMailSendFile()

- New options:
  LL_OPTION_CALC_SUMVARS_ON_PARTIAL_LINES
  LL_OPTION_PROJECTBACKUP
  LL_OPTION_ERR_ON_FILENOTFOUND
  LL_OPTION_NOFAXVARS
  LL_OPTION_NOMAILVARS
  LL_OPTIONSTR_EXPORTS_ALLOWED_IN_PREVIEW
  LL_OPTIONSTR_HELPFILENAME
  LL_OPTIONSTR_NULLVALUE

27.3. Changed Functions/Options

27.3.1. .NET-Component

- The functions LlPrintFields() and LlPrintFieldsEnd() return an "int" in place of a "bool". This change was necessary to support hierarchical table structures. If you simply want to port your projects, then follow the description in the next chapter.

28. Appendix: What's new in Version 12

28.1. Summary

28.1.1. General

- Crosstabs
- Easier coding for charts (also in table columns) using the *LlDb...()*-APIs
- Support for digital signatures
- .NET/VCL/OCX: convenient support for Dictionaries
- .NET: Databinding support for generic list classes
- HTML-Export: frames using CSS
- XLS-Export: optional new worksheet for each page
- New ActiveX controls for function and object Designer extensions
- Java: JNI-DLL

28.1.2. User interface

- New tool window "report structure" supporting tables, charts and crosstabs
- New property "Datasource" for charts (also in table columns) and crosstabs
- Completely reworked icons
- Anti-aliasing and increased color depth for preview sketch files
- Support for fixed barcode bar width
- Support for variable barcode bar ratio
- New barcode SSCC/NVE
- New tool window for project properties
- New property "Minimum Page Count" for project
- Support for gradient fills
- Appearance condition for text paragraphs and table columns
- Improved page wrap for HTML object

28.2. New Functions/Options

28.2.1. DLL-API

- New options:
  LL_OPTION_ENABLE_STANDALONE_DATACOLLECTING_OBJECTS
  LL_OPTIONSTR_VARALIAS
- New callback:
  LL_CMND_SAVEFILENAME

28.3. Changed Functions/Options

28.4. .NET-Component

- the *LlDbAddTable()*, *LlDbAddTableSortOrder()* and *LlDbAddTableRelation()* APIs of the core object don't have a "DisplayName" parameter anymore. Use the new dictionary object, ex. LL.Dictionary.Tables.Add("Orders", "Bestellungen").

28.5. Upgrading to List & Label 12

28.5.1. General

The file format between List & Label Version 11 and 12 has changed. Converting projects is normally no problem with the new version. By default a warning message is shown if the user is converting an old project into the new format. You can suppress this message with LL_OPTION_NOFILEVERSIONUPGRADEWARNING. Pre-existing projects will be converted in the background during print time.

As with any software update, we highly recommend to check all project and template files after updating. Improvements can sometimes mean that things are done slightly different which might have an unexpected impact on your projects.

28.5.2. Converting VB Projects

You can convert pre-existing Visual Basic-Projects as follows:

List & Label Projects Using the OCX:

- Load the Visual-Basic project (*.vbp resp. *.mak) in a text editor. Replace the line
  ```
  Object={2213E283-16BC-101D-AFD4-040224009C0B}#11.0#0; CMLL11O.OCX
  ```
  with the following
  ```
  Object={2213E283-16BC-101D-AFD4-040224009C0C}#12.0#0; CMLL12O.OCX
  ```
  and the line
  ```
  Module= CMLL11; CMLL11.BAS
  ```
  with the line
  ```
  Module=CMLL12; CMLL12.BAS
  ```
- After saving your changes load the form (*.frm) in a text editor, which contains the List & Label-OCX. Replace the line
  ```
  Object = "{2213E183-16BC-101D-AFD4-040224009C0B}#11.0#0";
  "CMLL11O.OCX"
  ```
  with the following
  ```
  Object = "{2213E183-16BC-101D-AFD4-040224009C0C}#12.0#0";
  "CMLL112.OCX"
  ```
- In case that you want to convert older List & Label versions to List & Label 12 change the according entries analogously.
- Now you can load your projects in Visual Basic. The source code must be significant adapted according to the original version.
- Since VB 5 the .BAS declaration file is not necessary, because the List & Label constants are contained in the OCX-control.

28.5.3. Converting Delphi-Projects

See the hints in the Delphi online help.

28.5.4. Converting .NET-Projects

Normally it is enough to replace the reference to the listlabel11.dll resp. listlabel11Unicode.dll by a reference to the listlabel12.dll resp. listlabel12unicode.dll. Additionally you should replace the old components in your tool box by the new ones.

29. Small changes, great enhancements

29.1. User's choice of the print medium

The following does only apply to your own printing loop, not to the one that is inside the .NET, OCX or VCL control (this one already implements this feature):

If you supply *LL_PRINT_USERSELECT* or *LL_PRINT_EXPORT* to the print type of *LlPrint[WithBox]Start()*, the user can select the print medium (printer, file, preview or export). You save one menu option as you do not have to offer "Print to Printer" and "Print to Preview" etc, and it looks nice. Before ending the print job (*LlPrintEnd()*) you have to query the selected device by calling *LlPrintGetOptionString(LL_PRNOPTSTR_EXPORT)*: "PRV" (preview), "PRN" (printer) and "FILE" (file) or any exporter ID string will tell the output media the project has been sent to.

29.2. Speed hints

Do not use special characters in variable or field identifiers. Then switch off the check with *LlSetOption(hJob,LL_OPTION_XLATVARNAMES,0)* and time-consuming operations will be suppressed.

If your application is finished with testing, you can disable the parameter check of List & Label using *LlSetOption(hJob,LL_OPTION_NOPARAMETERCHECK,1)*. This also reduces some - hopefully unnecessary - overhead.

If you use *LlPrintIs[Chart]FieldUsed()* and *LlPrintIsVariableUsed()* before setting variable contents, you can save time depending on the complexity of your database requests...

29.3. List & Label within NT-Services

Whenever *List & Label* is run within a Service, all message boxes are suppressed and the default answer is given. This is being logged to the COMBIT.LOG file in your windows directory, enabling you to trace what's happening.

As DEBWIN2 cannot debug services, the debug output may be directed to LOG-File. Set the flag *LL_DEBUG_CMBTLL_LOGTOFILE* using *LlSetDebug()* to do so.

Please note the license agreement when running List & Label in a service.

29.4. Web reporting

The usage of List & Label with ASP.NET is described in detail in chapter "5.11. Using List & Label for Web reporting". Of course, also other programming languages can be used for web reporting. In general, the following prerequisites must be met:

- The server has to run a Windows OS, List & Label can only be used on windows platforms. Of course the clients are not restricted to this.
- The user account that runs the web application needs to have access to a printer driver. The printer doesn't need to be physically available, the driver itself is

sufficient. Also, the user must have read access to the path where the *List & Label*-DLLs are stored.

- The actual print process is a quiet export (see chapter "13.2.5. Export Without User Interaction") and redirects the client to the resulting export file.

Please read the license agreement before using *List & Label* on a web server.

29.5. More tricks

Many more hints and tricks can be found in the online Knowledge Base at http://support.combit.net.

30. Support Concept for combit Development Tools

You have purchased a combit professional development tool. You have the possibility to obtain direct support from the manufacturer of this product in Germany.

You will find our support concept in the information material that came with your product or at http://support.combit.net.

31. Index

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Z