



report generator system

NCReport Definitive Guide

Norbert Szabo

Version 3.6, October 2026 - NCReport 2.35

Table of Contents

Dedication	1
Preface	2
I: Basics	3
1. Introduction to Reporting Tools	4
2. About NCReport in general	5
2.1. A short history	5
2.2. What is NCReport?	5
2.3. Features	5
2.4. Why NCReport?	6
2.5. How NCReport works	6
2.6. Working schema	6
2.7. Report template file	7
3. Installing NCReport	8
3.1. Requirements	8
3.2. NCReport has been tested with:	8
3.3. Install binary package on Linux	8
3.4. Install (commercial) source package under Linux	9
3.5. Install binary package on Windows	9
3.6. Install (commercial) source package under Windows	9
3.7. Contents of the installation directory	10
3.8. Acknowledgements	10
4. Getting started	11
4.1. Creating a basic report	11
4.2. Beginning with a new report	11
4.3. Testing the report in the Designer	21
4.4. Variables and Groups	22
4.5. Integrating NCReport into a Qt application	26
5. Report Items	30
5.1. Static Text Item	30
5.2. HTML (Rich) Text	30
5.3. Data Field	31
5.4. Line	31
5.5. Rectangle	31
5.6. Ellipse	31
5.7. Image	31
5.8. Barcode	31
5.9. Table View / Model	32
5.10. Cross Table	32
5.11. Custom Graphics Content	32

5.12. Graphics View Content	32
5.13. Chart	32
6. Parameters	33
6.1. Parameter Syntax	33
6.2. Testing Parameters	33
6.3. API code: Passing parameters to the report	34
II: Designer	35
7. Getting Started with NCRReport Designer	36
7.1. Launching Designer	36
7.2. The User Interface	36
7.3. NCRReport Designer Main Window	37
7.4. Geometry editor	38
7.5. Data Source Tree	39
7.6. Field Expression Builder	40
8. Designing a Report	42
8.1. Connecting to a database from Designer	42
8.2. Beginning a new report	44
8.3. Report sections	45
8.4. Setting up page and report options	47
8.5. Adding data sources	53
8.6. Assigning a data source to the Detail	56
8.7. Adding report items	57
8.8. Adding a total variable field	66
8.9. Other items	67
8.10. Adjustment and formatting	67
8.11. Running the report	68
9. Designer Reference	70
9.1. Editing Tools	70
9.2. Grid and Snap to Grid	73
9.3. Zooming	74
9.4. XML Source of the Report	74
9.5. Opening and Saving Reports in a Database	74
9.6. Layout Direction	76
9.7. The Preferences Dialog	77
9.8. The Test Data Editor	80
9.9. User Interface Languages	81
9.10. Command Line Arguments	82
III: Advanced Features	83
10. Data Sources	84
10.1. Data source types	84
10.2. Common data source properties	84

10.3. Location types	85
10.4. Opening roles	86
10.5. Referring to data source columns	86
10.6. SQL data source	87
10.7. Text data source	89
10.8. XML data source	89
10.9. String list data source	90
10.10. Item model data source	91
10.11. Custom data sources	91
10.12. Data sources in the Designer	92
11. Data/Script Expressions	95
11.1. Using references in expressions	95
11.2. References in templates	96
11.3. Reference examples	96
11.4. Testing Field Expression	96
11.5. Print-When Expressions	97
11.6. Testing a print-when expression	98
11.7. Print only when expression is true condition	98
11.8. Templates in Fields and Texts	98
11.9. Script expressions in special locations	98
11.10. Data Source Functions	99
11.11. Data source (meta) functions	99
11.12. Data source column (value) functions	101
12. Variables and Totals	103
12.1. How variables work	103
12.2. Page totals and running balances	107
12.3. Group totals and nested group totals	108
12.4. Modifying variables	109
12.5. System variables	111
13. Script Editor	113
13.1. Script ID	113
13.2. Script Definition	113
13.3. Available Buttons	113
14. Data Formatting	115
14.1. Text formats	115
14.2. Numeric formats	116
14.3. Date Formats	117
14.4. Template arg() string	120
14.5. Boolean values	120
15. Text and Item Display Options	121
15.1. Rich Text, Plain Text and Markdown in Text Items	121

15.2. Markdown in Text Items	122
15.3. Hiding Repeated Values	123
15.4. Breaking Rich Text Across Pages	124
15.5. Rotating Items	125
16. Images	128
16.1. Adding an Image Item	128
16.2. Image Sources	129
16.3. Image Formats	130
16.4. Scaling and Transformation	131
16.5. Alignment	132
16.6. HTML Options	132
16.7. Output Support	132
16.8. XML Examples	132
17. Barcodes	134
17.1. How Barcodes Are Rendered	134
17.2. Adding a Barcode Item	134
17.3. Setting the Symbology	135
17.4. Barcode Value: Static or Dynamic	137
17.5. Barcode Options	137
17.6. XML Examples	138
18. Charts and Graphics	140
18.1. Online Chart Images	140
18.2. Chart Item (Highcharts Integration)	141
18.3. Graph (Custom) Item	142
18.4. Graphics View Item	144
19. Zones	146
19.1. Zone settings	147
19.2. Zones example	147
20. Dynamic data driven size and position	148
20.1. Dynamic position and size settings	148
20.2. Examples	148
21. Dynamic data driven shape style	150
21.1. Dynamic style settings	150
21.2. Example	150
22. Page Breaks	151
22.1. Detail page break condition	151
22.2. Detail starts on new page	151
22.3. Group page break options	151
22.4. Report item page break	151
22.5. Section options related to paging	152
22.6. Batch and sub-report page breaks	152

23. Text Document printout mode	153
23.1. Steps of usage	153
23.2. How the document is built	153
23.3. Text Document printout report example	153
24. Label and Multi-Column Reports	154
24.1. Report Types	154
24.2. Label Settings	154
24.3. Designing the Label	155
24.4. How Labels Are Generated	156
24.5. Planning a Label Sheet	156
24.6. Multi-Column Reports	157
25. Data Relation System	159
25.1. Defining a parent data source	159
25.2. Defining child data sources	160
25.3. Setting up the detail section	161
25.4. Designing the report	161
25.5. Changes in version 2.13	162
26. Double pass mode	164
26.1. Setting double pass mode	164
26.2. Example using the <code>pagecount</code> variable	164
27. Internationalization	165
27.1. Adding languages	165
27.2. Adding translations of Fields or Labels	165
27.3. Setting up the current language	166
27.4. Setting up the language	166
27.5. Setting up the current language from command line	166
28. Sub-Report iteration	167
28.1. Sub-Report data source	167
28.2. Reference to master data source	167
28.3. Sub-Report options	167
29. Table View Rendering	169
29.1. Adding TableView item	169
29.2. Table View Dialog	169
29.3. Setting the object references	170
29.4. Example	170
29.5. Printing Item Model Based Table without QTableView	171
29.6. Custom Cell Content	172
29.7. Handle progress signal of table rendering	173
30. Cross-Tab Tables	174
30.1. Table Structure	174
30.2. Using Cross-Tab Tables in the Designer	175

31. Conditional Formatting	180
31.1. Dynamic Style Tag Symbols	180
31.2. Editing Style Code in Designer	180
31.3. Default Style	181
31.4. Examples	181
32. Report Outputs	182
32.1. Output types	182
32.2. Common output options	182
32.3. Running the report with an output object	183
32.4. Preview output	184
32.5. Preview dialogs: Qt print preview and modern preview	184
32.6. Printer output	185
32.7. PDF output	185
32.8. SVG output	186
32.9. Image output	186
32.10. Markup language outputs: HTML and Excel	187
32.11. Text output	188
32.12. Custom outputs	188
32.13. Selecting the output in the Designer	189
32.14. Selecting the output in the command line tool	189
33. General Text Output	190
33.1. Text template manager tags	190
33.2. Text template tags	190
33.3. Examples	191
33.4. Running a report to text output	191
34. Batch Report Mode	192
34.1. How batch mode works	192
34.2. Batch mode API	192
34.3. Example	193
34.4. Running batch reports in the Designer	194
35. Special Detail Sections	195
35.1. Sub (Detail) Sections	195
35.2. Adding a Sub-Section	195
35.3. Odd / Even Pages	196
35.4. Repeated detail by constant or dynamic value	197
36. User Functions	198
36.1. The NCReportUserFunction Class	198
36.2. When the Function Is Called	199
36.3. Assigning a Function to an Item	199
36.4. Implementing and Registering a User Function	199
36.5. Script-Based Functions	200

IV: Command Line Tool	202
37. Command line client	203
37.1. Running the command line executable	203
37.2. Command line options.	203
V: Using the NCReport API	206
38. Using the NCReport API	207
38.1. Project file settings	207
38.2. Initializing the NCReport class	207
38.3. Include directives	207
38.4. Creating the NCReport class	208
38.5. Connecting to an SQL database	208
38.6. Setting the report's source	209
38.7. Adding parameters	209
38.8. Running the report	209
38.9. Running the report in one step	209
38.10. Running the report in customized mode	210
38.11. Initializing the report's output	210
38.12. Running the report	211
38.13. Previewing the report	211
38.14. Deleting the report object	212
38.15. Using other data sources	212
38.16. Custom data sources	215
38.17. Custom items in NCReport	219
38.18. Batch report mode	221
38.19. Monitoring report progress	221
38.20. Customization hooks	225
38.21. Other runtime settings	228
VI: Specification	231
39. Specification	232
39.1. Root element	232
39.2. Page options	233
39.3. Data sources	235
39.4. Report Parameters	240
39.5. Variables	240
39.6. System Variables	242
39.7. Scripts	242
39.8. Expressions	243
39.9. References in expressions	243
39.10. Using a script expression in a field:	244
39.11. Report sections	244
39.12. Report items	249

Appendix A: Glossary	262
Index	265

Dedication

To all NCReport users,

This work is dedicated to you, the tireless explorers of the digital realm. Your curiosity drives innovation, your perseverance overcomes countless challenges, and your creativity brings life to technology in ways unimaginable. Whether you are troubleshooting, coding, designing, or simply navigating the complexities of the virtual world, your efforts shape the future of our digital landscape.

Thank you for your resilience, your passion, and your relentless pursuit of excellence. This is for you.

With friendly greetings,

Norbert Szabo

Preface

This book is designed to be the clear, concise, standard reference for the NCReport reporting software. It serves as the official documentation for NCReport. We hope to answer, definitively, all the questions you might have about the elements, features, and entities in NCReport. It is a comprehensive user manual and definitive guide to the NCReport reporting system. It also contains installation instructions, tutorials, and information about the contents of the distribution. In particular, we cover the following subjects:

- The general nature of NCReport. We quickly bring you up to speed on how the pieces fit together.
- How to create NCReport reports. Where should you start and what should you do?
- Understanding all of the report elements. Each element is extensively documented, including its intended semantics and the purpose of all its attributes. An example of proper usage is given for every element.
- How to run NCReport reports. After you've created one, what do you do with it?
- How to integrate the NCReport library into a Qt application.
- How to run reports with the command line client.

Getting this documentation. This book is not available in printed form, but you can print it yourself if you prefer to read it on paper. You can download the electronic (PDF) version from our web site: <https://ncreport.org/download>

Getting the examples. All of the examples are available on our web site. The sample reports used in this book are also included in the `reports` directory of the NCReport distribution. The most up-to-date information about this documentation is available at <https://ncreport.org/documentation>

Request for comments. Please help us improve future editions of this book by reporting any errors, inaccuracies, bugs, misleading or confusing statements, and plain old typos that you find. An online errata list is maintained at <https://support.grassbits.com/>. Email your bug reports and comments to support@grassbits.com.

Part I: Basics

This part introduces reporting tools and NCReport, explains how to install it, and walks you through creating, testing, and running your first report.

Chapter 1. Introduction to Reporting Tools

A report generator, often also called a reporting tool, is a computer program or software application designed to automate the process of creating, designing, and producing various types of reports. These reports can encompass a wide range of data and information, such as financial statements, business intelligence reports, marketing analytics, inventory reports, and more.

Key features and functionalities of report generator software tools typically include:

- **Data Extraction:** They can extract data from various sources, such as databases, spreadsheets, and external APIs.
- **Report Design:** Report generators often provide tools for designing the layout and formatting of reports, including the ability to add tables, charts, graphs, and other visual elements.
- **Template Creation:** Users can create report templates or use pre-designed templates to maintain consistency in the look and feel of reports.
- **Data Manipulation:** These tools allow users to manipulate and transform data before it is included in a report, which can involve calculations, filtering, sorting, and grouping.
- **Automation:** Report generators automate the process of report generation, saving time and reducing the potential for human error.
- **Export Options:** They support various output formats, including PDF, Excel, HTML, CSV, and more, so that users can choose the format that best suits their needs.
- **Integration:** Many report generators can integrate with other software applications and databases, enabling seamless data extraction and reporting.
- **Scheduling:** Some report generators allow for the scheduling of report generation at specific intervals, which is particularly useful for recurring reports.
- **Collaboration:** Report generators may facilitate collaboration among team members by allowing them to work on reports simultaneously and share them with others.

Report generator software is commonly used in business environments to streamline the reporting process and provide decision-makers with timely and accurate information. It can be particularly valuable in industries such as finance, healthcare, marketing, and manufacturing, where regular reporting and data analysis are essential for operations and decision-making.

Chapter 2. About NCReport in general

2.1. A short history

NCReport is a mature software project with over 20 years of development history. Launched in 2002, it originated as a joint project for a Qt3 application before evolving into an independent GPL project. The system was initially conceived to address the need for robust data printing capabilities, a feature that was notably absent from the Qt/C++ programming environment at the time.

In 2007, the project underwent a complete rewrite and became a commercial product. This overhaul, designated version 2.0, adopted a fully object-oriented architecture. The 2.x line has been continuously developed ever since.

2.2. What is NCReport?

NCReport is a report generator and report writer tool: a report engine with a GUI designer, primarily for Qt applications, though it is by no means limited to the Qt environment. It enables applications to print data-driven reports, tables, lists, rich text documents, or any other paginated graphical content from one or more data sources. The system consists of two main parts: the report engine and the Designer GUI application. The report engine is also available as a command line executable. The report engine library can be integrated into any Qt application independently of the Designer. NCReport is already used and integrated by a growing community of commercial users and professionals.

2.3. Features

NCReport provides the following features and functions:

- **XML-Based Definitions:** Reports are defined using flexible XML templates.
- **Metric-Based, Band-Oriented System:** Precise control over layout and sections.
- **Diverse Data Sources:** Support for SQL databases, Text/CSV, QStringList, QAbstractItemModel, XML, and custom user-defined data source classes.
- **Multiple Output Formats:** Screen Preview, Printer, PostScript, PDF, SVG, Image, HTML, Excel (HTML-based), and Plain Text.
- **High-Performance Preview:** A dedicated, fast private preview window system.
- **Database Connectivity:** Support for both internal and external SQL database connections.
- **Rich Layout Items:** Labels (static text), Fields, HTML/Rich Text, Lines, Rectangles, Ellipses, Images, Barcodes, Tables (QTableView / item model), Cross tables, Graphics View content, and Custom Content items.
- **Advanced Pagination:** Support for Page Headers/Footers and Report Headers/Footers.
- **Data Grouping:** Unlimited nesting levels for data grouping with dedicated group headers and footers.

- Calculations: Variables for totals, aggregate functions, and system variables.
- Dynamic Content: Support for static/dynamic images and HTML content.
- Extensive Barcode Support: Rendering for over 50 barcode types via the integrated Zint library.
- Scripting & Logic: Template modes, expression evaluation, and script execution within fields.
- Application Integration: Pass parameters directly from the host application.
- Compatibility: Pure C (C17) code that builds with Qt5 and Qt6. Earlier releases also supported Qt4.
- Conditional Formatting: Dynamic formatting for Fields and Labels based on data values.

2.4. Why NCReport?

Modern software applications frequently interact with complex data structures and SQL databases. The ability to represent this data through professional, well-formatted reports is a critical requirement for enterprise-grade software.

The NCReport project represents thousands of development hours and benefits from continuous maintenance. By integrating this tool, developers save significant time and resources because they do not need to build custom printing logic from scratch. This is particularly advantageous for C developers using the Qt cross-platform framework. NCReport provides a fully portable, native C solution that scales across platforms.

2.5. How NCReport works

What exactly does NCReport do? In a few words, NCReport generates ready-to-print documents from raw data using a template. As a first step, an XML report definition must be created to serve as the template. The definition is a script for the report engine: it describes exactly what content to render, how it should look, where the data comes from, and so on. The report definition can be loaded from a local or remote file or from an SQL database, depending on the report source that is set. The report source is the part of the report engine that loads the report definition from its origin. The Designer is a separate GUI application for creating report XML definitions. When running a report, the report engine first parses the report definition and opens the specified data source(s). If an SQL data source is used, a valid, open SQL database connection is required (unless the data source defines its own private connection). After the data source(s) are successfully opened, the SQL query is run by the report director. The report engine begins to process the data row by row from the data source assigned to the first detail section. While the report is processing, the report director manages the rendering of the different sections and the items inside them. The result is rendered to the specified output, such as a printer, print preview, PostScript, PDF, SVG, image, HTML, Excel (HTML), or text.

2.6. Working schema

The following diagram shows the main parts of the NCReport system and how a report is produced. The host application (your Qt application, the command line tool, or the Designer) runs the report:

1. The report source manager loads the report definition from a file, a string, or an SQL database

table, and the definition reader parses the XML template.

2. The data source manager opens the data sources defined in the report and reads their records row by row.
3. The report director iterates the records, manages groups, page breaks, and totals, and the section director renders the sections and their items. Expressions, scripts, and variables are calculated by the evaluator.
4. The rendered pages are written to the selected output: the print preview, a printer, a PDF, SVG, or image file, HTML, Excel (HTML), or plain text.

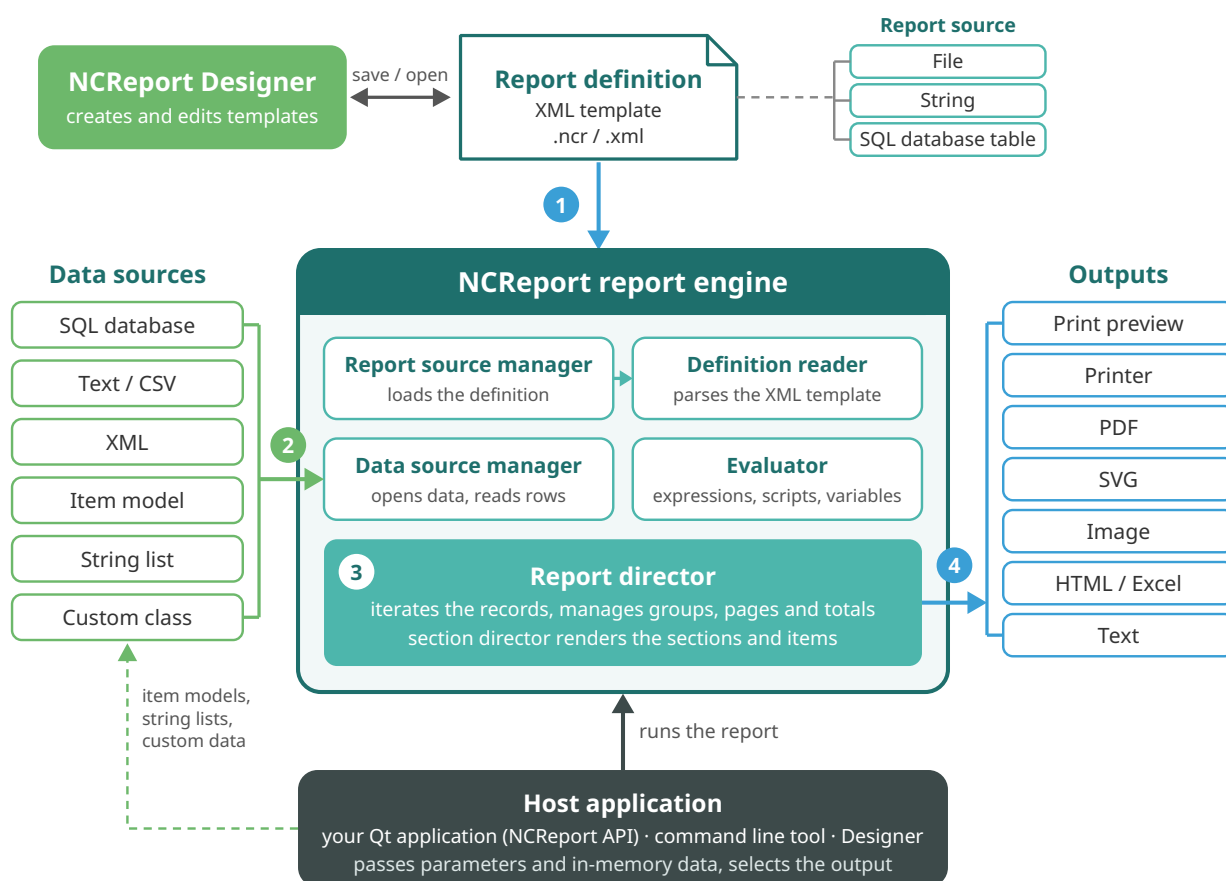


Figure 1. NCReport system structure and report processing

2.7. Report template file

NCReport uses the Extensible Markup Language (XML) format for report definitions. XML is a universal, standard file format, which makes report definition templates easy to read and process, both for humans and for other software. The Designer saves report definitions with the `.ncr` or `.xml` file extension.

Chapter 3. Installing NCReport

3.1. Requirements

- Linux or another Unix-like operating system, Microsoft Windows, or macOS 10.5 or later.
- At least 512 MB of memory and a 1 GHz CPU.
- 90 MB of free disk space.

NCReport is officially supported on Windows XP/Vista/10/11, on Linux 2.6 or later, and on macOS 10.4 or later. It can also be used on other platforms supported by Qt, but with limited or no support from us.

3.2. NCReport has been tested with:

- Qt 4.5 to Qt 6.10 under Windows XP/7/10/11
- Qt 4.5 to Qt 6.10 under Linux (Ubuntu 10.04, 12.04, 16.04, 18.04, 20.04, 22.04, 24.04, and Debian)
- Qt 4.7 to Qt 6.10 under macOS 10.6 to macOS 12

Current versions of NCReport are written in C++17 and are built with Qt5 or Qt6.

3.3. Install binary package on Linux

Make sure that the appropriate Qt version binaries are already installed on your Linux system. The required version is specified in the downloaded package. Unpack the NCReport Linux distribution to any directory you want (for example, `ncreport`):

```
$ mkdir ncreport
$ cd ncreport
$ tar xzvf ncreport2.x.x.tar.gz
$ cd bin
```

The NCReport binaries are intended to be used directly from the `bin` directory of the unpacked package; you can start them by simply executing them. To start the report designer:

```
$ ./NCReportDesigner
```

To start the command line report engine (see [Chapter 37](#)):

```
$ ./NCReport
```

Afterward, you may want to add this `bin` directory to your `$PATH`.

3.4. Install (commercial) source package under Linux

Make sure that the GCC/G C compiler and the appropriate version of the Qt (Qt5 or Qt6) development environment are already installed on your Linux system. In addition, you need to compile and install the appropriate Qt database drivers. The example reports mostly use the QMYSQL and QSQLITE database drivers.

Unpack the NCReport Linux source package in any directory you want and build it:

```
$ cd directory
$ tar xzvf ncreport2.x.x.tar.gz
$ cd NCReport2.x.x
$ qmake -r
$ make
```

The build options (optional modules such as the Designer, the console application, barcode support, and so on) can be set in the `src/config.pri` file before running `qmake`. To start the NCReport binaries, follow the steps described in the previous section.

3.5. Install binary package on Windows

We strongly recommend that you download and install one of the self-installing setup files (`NCReport_2.x.x_Windows.exe` or `NCReport_2.x.x_Windows_MinGW.exe`).

Simply run the setup executable and follow the instructions of the setup wizard. To start NCReport Designer, use the Start menu.

3.6. Install (commercial) source package under Windows

This option is available to license holders only. Make sure that a Windows C development environment is already installed on your system. If you use the open source version of Qt, the `GNU MinGW` compiler is included in the Qt SDK. The following example shows the build procedure using the `Microsoft Visual C` compiler.

Make sure that the appropriate version of the Qt development environment is already installed on your Windows system. In addition, you need to compile and install the appropriate Qt database drivers. The example reports mostly use the QMYSQL and QSQLITE database drivers.

Unpack the downloaded `ncreport2.x.x.zip`, `.tar.gz`, or `.7z` source package. Use a tool such as WinZip or 7-Zip; Windows has built-in support for `.zip` archives. To unzip the NCReport distribution into any directory and build it:

```
mkdir ncreport
cd ncreport
unzip ncreport_2.x.x_src.zip
qmake -r
nmake release
```

When using the MinGW compiler, run `mingw32-make release` instead of `nmake release`.

3.7. Contents of the installation directory

- `/bin` Contains the NCReport executable files.
- `/doc` Contains the User Guide and API documentation in HTML format.
- `/sql` Contains the SQL script files required for some of the example reports.
- `/reports` Contains the sample reports that demonstrate NCReport features.
- `/lib` Contains the binary library files (Unix/Linux only).
- `/testdata` Contains test files for demonstration purposes. The `defaulttestdata.xml` file is used by the Designer application to store test data. If you want to use it, copy this file to the `/bin` directory before starting NCReportDesigner.
- `/i18n` Contains internationalization files.
- `/images` Contains image files used by some of the sample reports, for example `mysql_productlist_with_dynimages_demo.xml`.
- `/src` Contains the source code of the NCReport system. The binary package contains only the source code of the demo and sample applications. The full source code is available to commercial license holders only.

3.8. Acknowledgements

On Windows, the NCReport installer is built using Inno Setup, Jordan Russell's software (<https://www.jrsoftware.org>). We highly recommend this excellent, free-to-use tool.

Chapter 4. Getting started

This chapter provides a quick introduction to the NCReport system. If you are already familiar with the tool, you only need to skim this chapter. To work with NCReport, you need to understand a few basic concepts of report design in general, and of NCReport in particular. These are covered here. You also need some hands-on experience with the way an NCReport report definition is structured. This is covered in the following chapters.

4.1. Creating a basic report

To begin, we go through the basic steps of creating a simple report. In our example, we build a plain product price list report grouped by product category. The finished report is included in the distribution as [reports/gettingstarted_demo.xml](#).

4.2. Beginning with a new report

Open the Designer application and start a new report by clicking the **[New]** toolbar button or by selecting **File > New**.

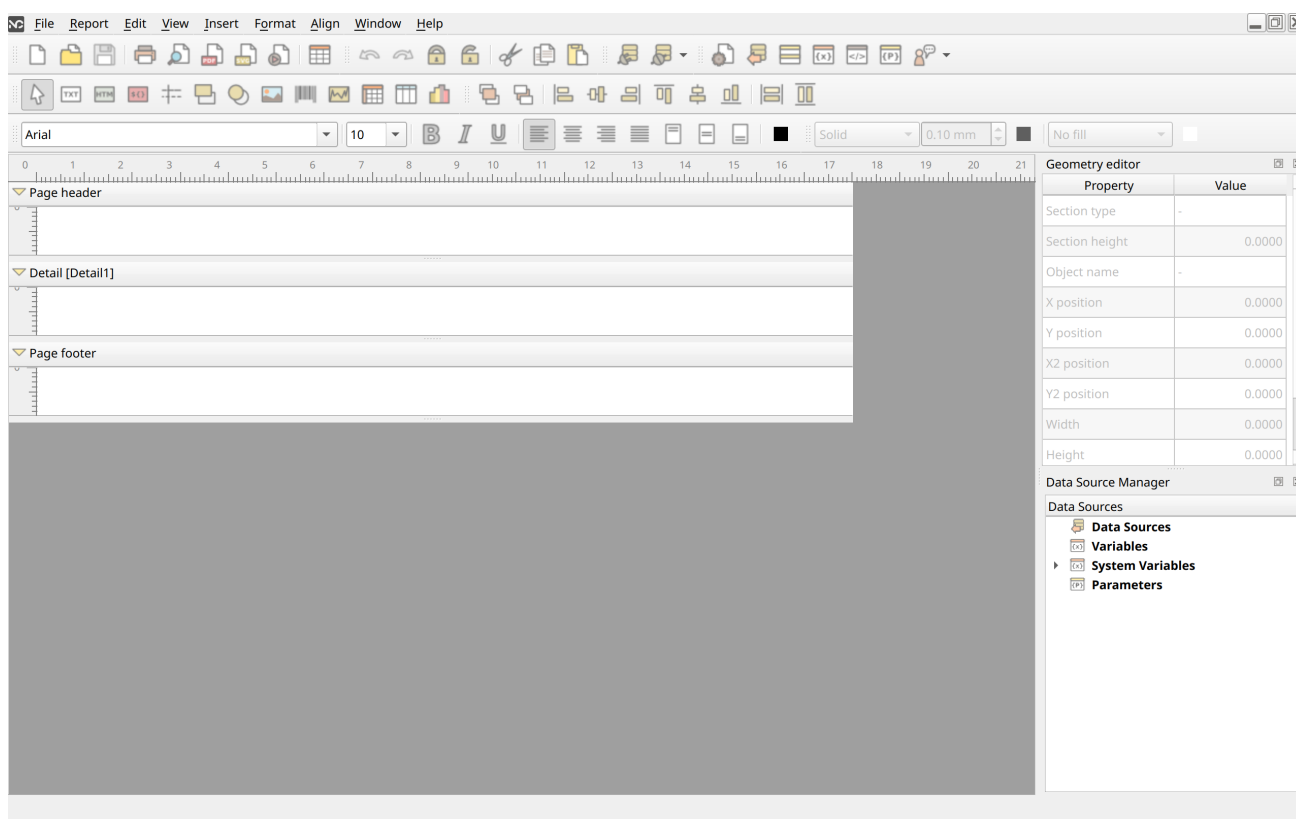


Figure 2. A new empty report in the Designer

4.2.1. Setting up page options

You can specify the page options of the current report in the **Report and page settings** dialog. To open it, select **Report > Report and page settings....** In this dialog, you can specify the following options

(see [Section 8.4](#) for the complete list):

Report name

The name of the report. This is an informative option only; it is not used by the report generator.

Report type

Three types of report are available. **Report** represents a normal report. **Text document** is a limited report mode in which the report can contain HTML text items only; the generated report is a paginated HTML document. **Label** mode prints the detail records as labels arranged in rows and columns on the page; the label dimensions are set on the **Label** tab of the dialog.

File encoding

The encoding of the XML file. This encoding is used when the report definition file is opened or saved. In most cases UTF-8 meets the requirements, but for special international characters you can choose any other supported encoding.

Default font

The default font family, size, and style used for the text labels and fields throughout the report. Individual item settings may override this option.

Page size

The size of the page. The combo box lists the standard paper size names (A4, Letter, and so on). To use a non-standard paper size, select **CUSTOM** and specify the width and height of the page in millimeters.

Background color

The background color of the report. This option is currently unused.

Header and footer settings

Use the check boxes to enable or disable the page header, page footer, report header, and report footer. To change the height of these sections, use the spin boxes next to their check boxes. You can also change the section heights by dragging with the mouse or by using the geometry editor.

Margins

The margin properties represent the top, bottom, left, and right margins of the page in millimeters. To change the margin values, use the spin boxes.

Orientation

This radio button option sets the orientation of the page. You can select either Portrait or Landscape orientation.

Specify the page properties as in this example and click **[OK]** to save the settings. We set only the report name and leave the other properties at their default values.

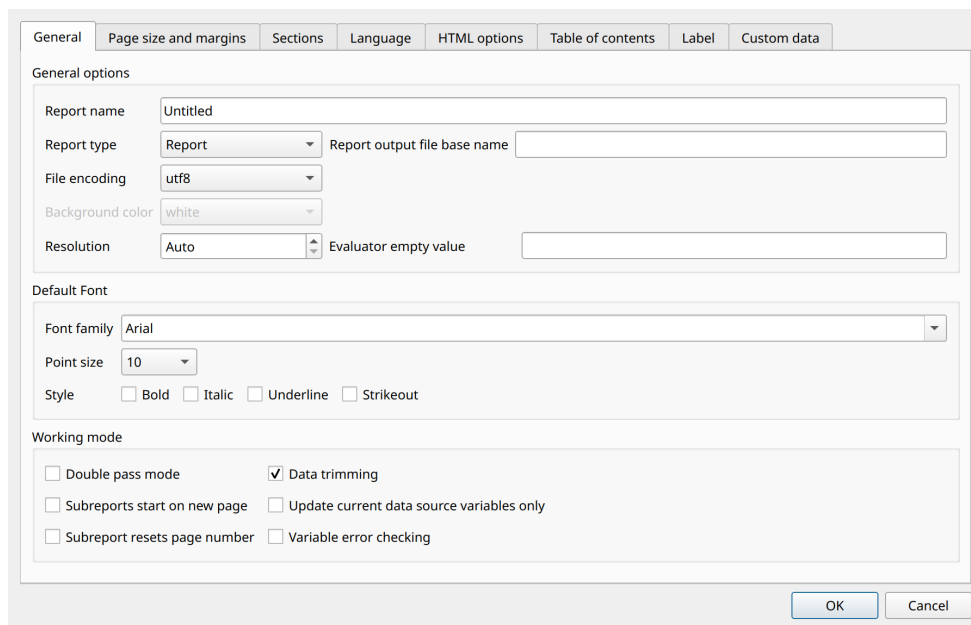
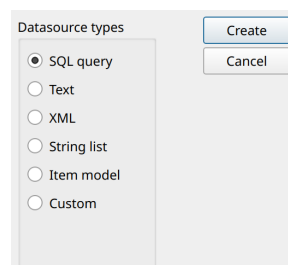


Figure 3. Page settings dialog

4.2.2. Adding a data source

A new, empty report contains a page header, a detail section, and a page footer by default. Before we start adding report items, we define the data source, which specifies where the data comes from. In our example, the data source is a string list (`QStringList`). To specify a data source in your report, select **Report > Data sources....** A dialog then appears in which you can add and remove data sources. To add a new data source, click the **[Add]** button in the dialog, select the `String list` data source type from the list of available data source types, and click the **[Create]** button.



In the data source dialog, we specify the following properties:

Data source ID

The identifier of the data source. This ID is used to assign the data source to a detail section and to refer to it in expressions.

Data source type

The type of the data source, which you have already chosen.

Location type

The location type describes where the data can be found. In this report, we keep the default **Static** location type.

String list ID

The ID of the string list that provides the data. A string list is passed to the report by the application (or, in the Designer, by the test data editor) under this ID. In our example, the string list ID is **prod**, and the column delimiter is **Tab**.

Specify the data source properties as in this example and click **[OK]** to save the data source settings.

The screenshot shows a dialog box titled "Data sources". On the left, there is a list box containing "prodlist". On the right, the "Data source properties" section is active. It contains the following fields:

- Data source type:** String list (dropdown)
- Enabled:** ☒
- Data source ID:** prodlist (text box)
- Parent datasource ID:** (empty text box)
- Opening/running role:** Beginning of the report (dropdown)
- Primary key column index:** 0 (spin box)
- Location type:** Static (dropdown)
- Allow empty data source:** ☒

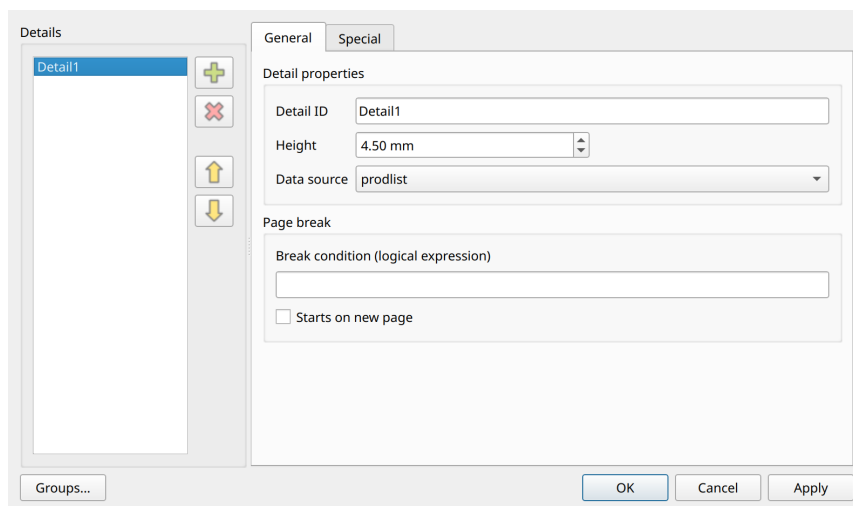
Below this section is the "QStringList datasource" section, which contains:

- String list ID:** prod (text box)
- Column delimiter:** Tab (dropdown)
- Custom delimiter character:** (empty text box)

At the bottom left of the dialog are two buttons: a green plus sign and a red minus sign. At the bottom right are "OK" and "Cancel" buttons.

4.2.3. Assigning the data source to the detail section

To assign the data source we defined earlier, select **Report > Details and grouping....** A dialog then appears in which you can manage the detail sections of the report. The default detail ID is **Detail1**, but you can change it to whatever you want. Select the previously defined data source from the **Data source** combo box.



Click the [**OK**] button to apply the detail settings.

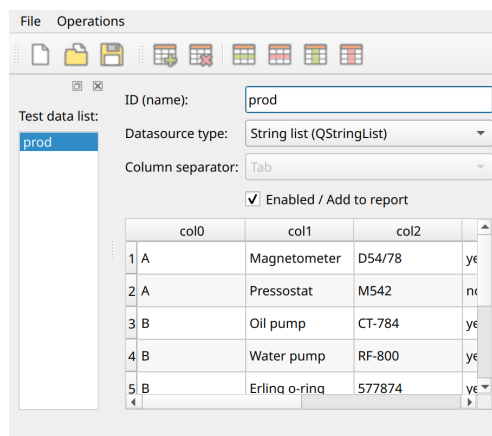
4.2.4. Preparing some test data

To try out our report in the Designer application, we need some test data. This step is not necessary if you test the report from your own application. The test data editor in the Designer makes designing, debugging, and testing reports easier. In our example, we create a simple product list with the following columns:

- **category** as column 0
- **product name** as column 1
- **product code** as column 2
- **active** as column 3
- **weight** as column 4
- **price** as column 5

A	Magnetometer	D54/78	yes	0.778	15.6
A	Pressostat	M542	no	2.547	30
B	Oil pump	CT-784	yes	1.510	17
B	Water pump	RF-800	yes	3.981	58
B	Erling o-ring	577874	yes	2.887	49
C	Hydraulic cup	HC55	no	0.435	39
C	Ballistic rocket	BV01	yes	1.260	157.9
C	Wheel	Q185/70	yes	25.554	199.0

To open the test data editor, select **Report > Test data editor....** A window then appears in which you can edit and set up test data. Three data source types are available for testing: **String list** (`QStringList`), **Item model** (`QStandardItemModel`), and **String parameter** (`QString`). We need the test data as a `QStringList`, so select the **String list** (`QStringList`) type, set the **ID (name)** to the string list ID of our data source (`prod`), and check the **Enabled / Add to report** check box.



Save the test data to apply the settings. The enabled test data, with its corresponding ID, tells the Designer to add the test data to the report under the specified ID before running the report.

4.2.5. Using the Geometry Editor

The geometry editor is a small tool window in the Designer for showing and editing the position and size of the selected item or section. To show or hide the geometry editor, toggle the **View > Geometry editor** menu item. The tool window appears on the right side of the main window. An item or section becomes current when you click it. You can type numeric size or position values into the editor. Any change made to an item's properties updates it immediately.

4.2.6. Designing the page header section

The page header contains the page headings. First, we add column titles as labels to the page header section. Labels are simple text items used to display descriptive information on a report, such as titles and headings. Labels are static items; their value never changes.

4.2.7. Adding Labels

Select the **[Label]** tool button or the **Insert > Label** menu item. The cursor then changes to a crosshair. Click in the page header of the report definition where you want the Label to be located. Doing so creates the Label object in that section and opens the Label settings dialog.

Add labels to the page header for the column titles and move them into position as in the example. Then select "Weight" and "Price" (multiple selection is supported) and align their text to the right by clicking the **[Right]** text alignment tool button.

Category	Product name	Available	Weight	Price

4.2.8. Resizing a section

Increase the height of the page header section by dragging the resizer bar at the bottom of the section. Another way to resize it is to type a **Section height** value in the geometry editor.

4.2.9. Drawing a line

To underline the labels, draw a line by selecting the **[Line]** button in the toolbar or the **Insert > Line** menu item. The cursor then changes to a crosshair. Click in the section where you want the line to start and drag the line to the end position. (To move the line, drag and drop it with the left mouse button.)

Category	Product name	Available	Weight	Price

4.2.10. Designing the Detail section

The core information in a report is displayed in its Detail section. This is the most important section of the report, since it contains the row-by-row data from the data source.

4.2.11. Adding Fields

Select the **[Field]** tool button or the **Insert › Field** menu item. The cursor then changes to a crosshair. Click in the detail section where you want the field to be located. Doing so creates the Field object in that section and opens the Field settings dialog. The following properties must be specified:

Field source type

The combo box contains the possible sources from which the field can pull data.

Field column/expression

This property specifies the data column from which the field's value is loaded. To identify data columns, specify:

- the name of the SQL column when using an SQL data source, or
- the column number, as `0,1,2...n` or `col0,col1,col2...coln`, when using a string list, item model, string parameter, or text data source.

Data type

The field's base data type. The following data types are supported: `Text`, `Numeric`, `Date`, and `Boolean`.

The field property dialog for the first column field:

Add fields to the Detail section and move them into position as in the example. The field column names are `col0`, `col1`, `col2`, `col3`, `col4`, `col5` (alternative naming: `0`, `1`, `2`, `3`, `4`, `5`). Select the `col4` and `col5` field items and align them to the right by clicking the **[Right]** text alignment tool button. Then, in the field dialog, set the data type to **Numeric** and use the **Numeric** tab page to set the number formatting properties.

Set the height of the detail section to 4.5 mm. After you add and format a title label in the page header section, the report should look like this:



4.2.12. Designing the page footer section

The page footer is usually used to display information such as the page number. In our example, we add only two system variable fields: the application info and the current page number.

4.2.13. Adding System variable fields

Select the **[Field]** tool button or the **Insert > Field** menu item. The cursor then changes to a crosshair. Click in the page footer section where you want the field to be located. Doing so creates the Field object in that section and opens the Field settings dialog.

4.2.14. Adding a page number field

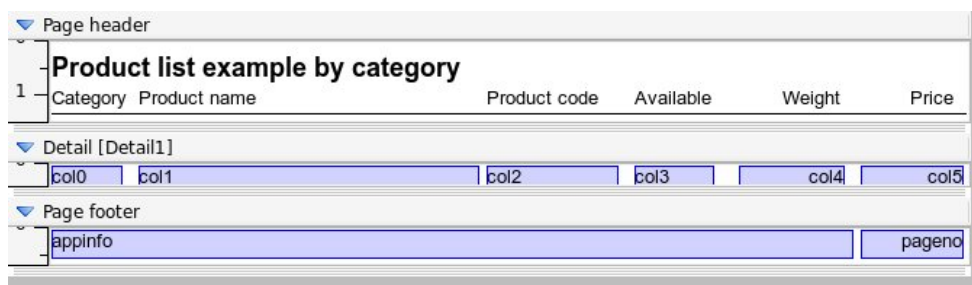
Specify the field properties as in this example (field source type: **System variable**, column: **pageno**):

4.2.15. Adding an application info field

Add another field to the page footer and specify its properties similarly to the previous one: Field source type: **System variable**, Field column/expression: **appinfo**.

4.2.16. Resizing a section

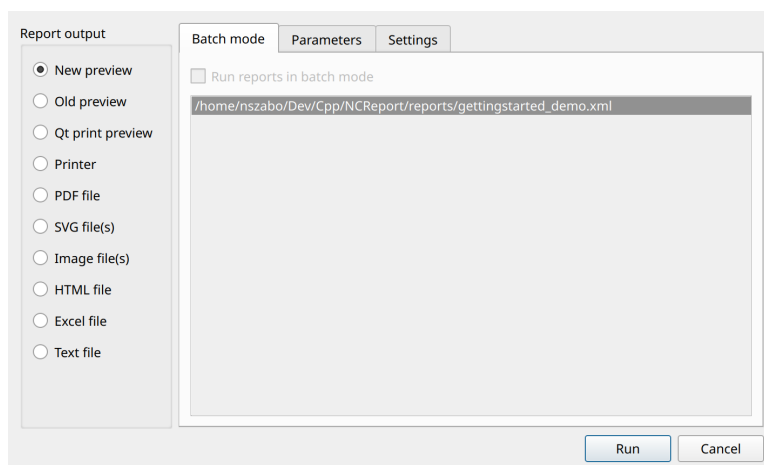
Decrease the height of the page footer section by dragging the resizer bar at the bottom of the section. Another way to resize it is to type a **Section height** value in the geometry editor. After setting the alignments and moving the fields into the right positions, the report should look like this:



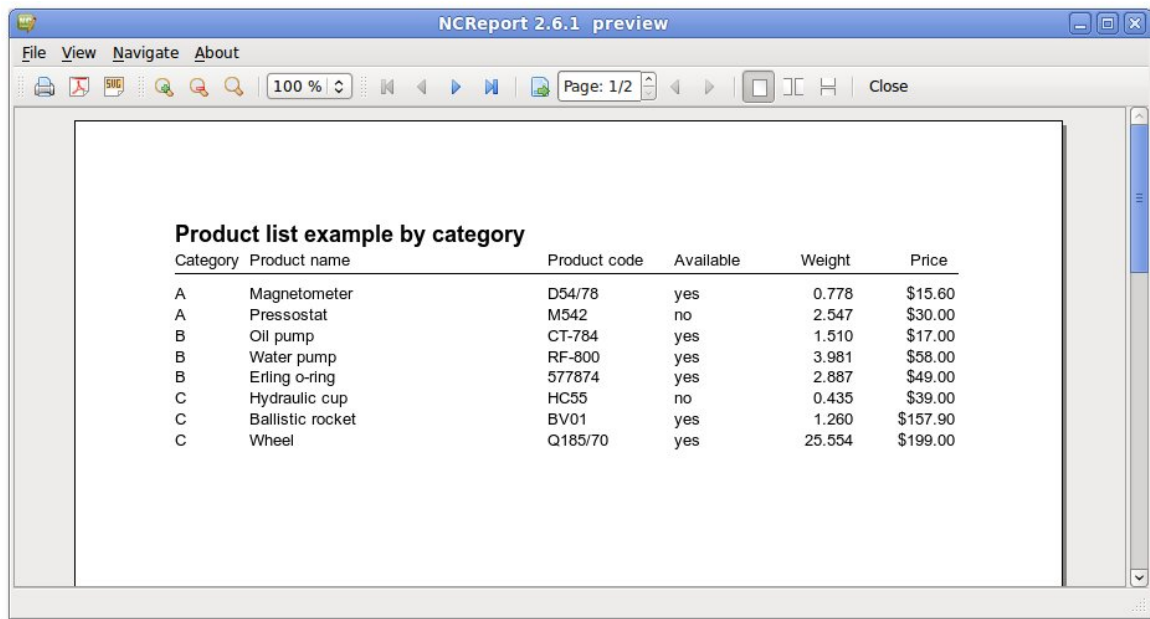
Page header					
1 Product list example by category					
Category	Product name	Product code	Available	Weight	Price
Detail [Detail1]					
col0	col1	col2	col3	col4	col5
Page footer					
appinfo					pageno

4.3. Testing the report in the Designer

Our sample report is now ready for testing. There are several ways to run the report from the Designer. Select **File > Run report...** (**Ctrl** + **R**); when the report runner dialog appears, choose the report output. To start running the report, click the **[OK]** button.



For a fast preview, select **File > Run report to > Preview** (**Ctrl** + **P**), and the Designer runs the report to print preview immediately. At this stage, the preview of our example report looks like this:



The screenshot shows the NCReport 2.6.1 preview window. The title bar reads 'NCReport 2.6.1 preview'. The menu bar includes 'File', 'View', 'Navigate', and 'About'. The toolbar contains icons for print, save, zoom (set to 100%), navigation, and a 'Close' button. The main content area displays a table titled 'Product list example by category'.

Category	Product name	Product code	Available	Weight	Price
A	Magnetometer	D54/78	yes	0.778	\$15.60
A	Pressostat	M542	no	2.547	\$30.00
B	Oil pump	CT-784	yes	1.510	\$17.00
B	Water pump	RF-800	yes	3.981	\$58.00
B	Erling o-ring	577874	yes	2.887	\$49.00
C	Hydraulic cup	HC55	no	0.435	\$39.00
C	Ballistic rocket	BV01	yes	1.260	\$157.90
C	Wheel	Q185/70	yes	25.554	\$199.00

4.4. Variables and Groups

The following section describes how to use some more advanced features of NCReport. We add a summary variable and then define a group in our example report.

4.4.1. Adding a variable for a summary

Variables are special numeric items used to provide counts, totals, and other aggregate values. Each variable has a name, a function type, and a data type, and is based on an expression, usually a data source column. To add a variable, select **Report > Variables....** A dialog then appears in which you can manage variables.

The following options are available for variables:

Variable ID

The name/ID of the variable.

Variable expression

The data source column (or expression) the variable is based on.

Function type

The aggregate function of the variable. Supported function types: **COUNT**, **SUM**, **AVERAGE**, **STD** (standard deviation), **MIN**, and **MAX**.

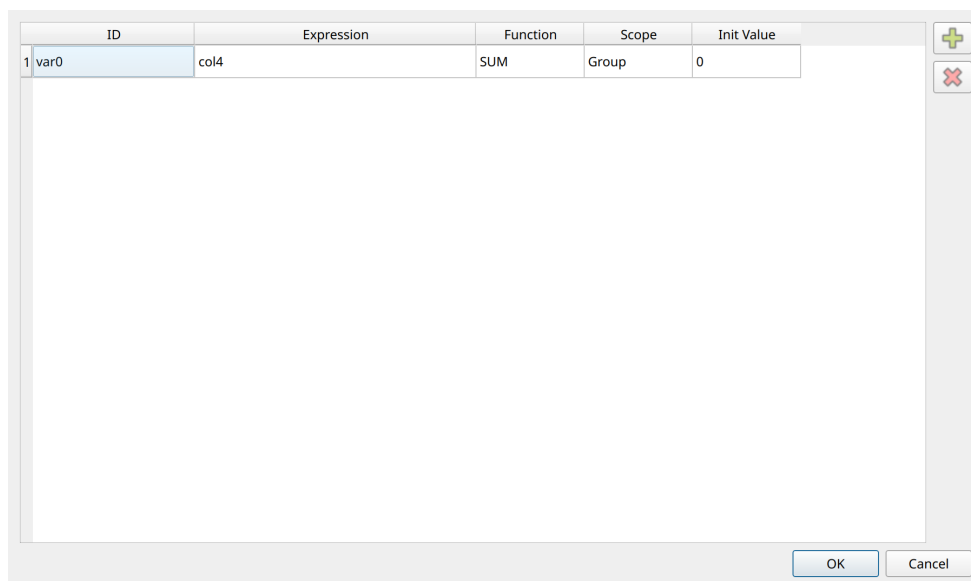
Reset scope

Specifies when the report engine resets the variable: **Report**, **Page**, or **Group**. For **Group** scope variables, the groups that reset the variable must also be set in the group settings dialog.

Initial value

The initial value of the variable.

Let's create a **var0** variable that sums the **col4** column (weight) with **Group** reset scope. Specify its properties as in this example:



ID	Expression	Function	Scope	Init Value	
1	var0	col4	SUM	Group	0

To apply the settings, click the **[OK]** button in the Variable dialog.

4.4.2. Defining a group

Reports often require summary data by group. In our example, we add a weight summary by product category to the report. First, select **Report > Details and grouping...** A dialog then appears in which you can manage the detail sections and the groups within a detail. Select the **Detail1** detail and click the **[Groups...]** button; the **Data grouping** dialog then appears. The following properties are available for a group:

Group ID

The name/ID of the group, used for identification purposes.

Group expression

The expression the group is based on. When the value of this expression changes, the group breaks. The group expression can be a data source column, any data expression, or a constant. To define a constant group, use the **%CONST** expression (or **0**); a constant group has only one group header and one group footer.

Header and Footer

To enable or disable the group header and footer, select or clear the corresponding check box. To set the initial height of these sections, use the spin boxes next to the check boxes.

Reset variables

This list contains the variables with **Group** reset scope. You can specify which variables the report generator resets when the group ends.

We want the grouping to be based on the **col0** column (the product category column). Specify the properties as in this example:

To apply the settings, click **[OK]** in the group dialog and then **[OK]** in the detail dialog. The group header and footer of the detail then appear in the report.

Page header					
Product list example by category					
Category	Product name	Product code	Available	Weight	Price
Group header [Detail1.Group0]					
Detail [Detail1]					
col0	col1	col2	col3	col4	col5
Group footer [Detail1.Group0]					
Page footer					
appinfo					pageno

4.4.3. Adding a summary field to the group footer

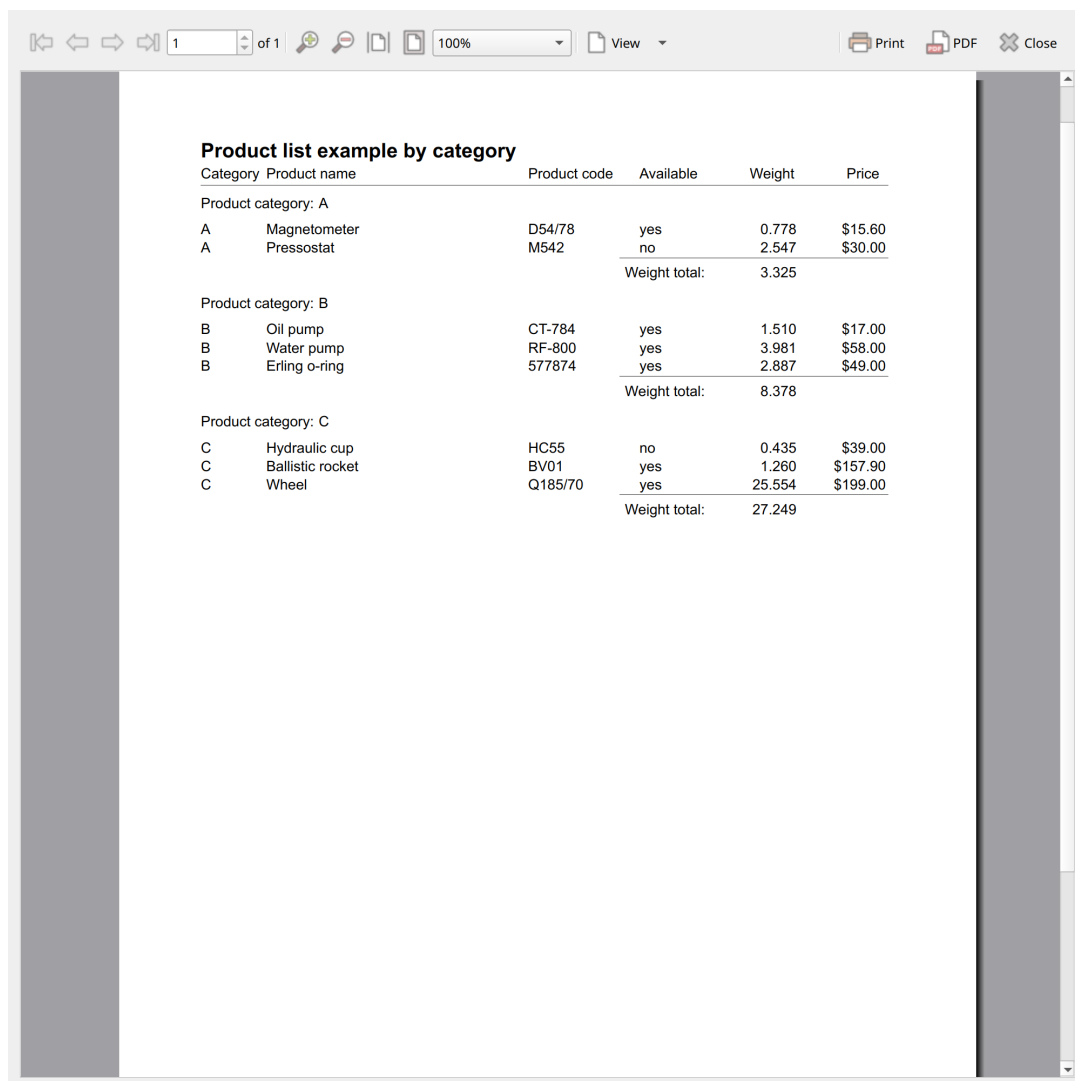
To add a field based on the **var0** variable, add another field to the group footer and specify its properties as in the example:

After you add the variable field, along with some labels and a line, to the group header and footer, the report should look like this:

Product list example by category					
Category	Product name	Product code	Available	Weight	Price
Group header [Detail1.Group0]					
Product category: col0					
Detail [Detail1]					
col0	col1	col2	col3	col4	col5
Group footer [Detail1.Group0]					
				Weight total: var0	
Page footer					
appinfo					pageno

4.4.4. Final testing of the report

Now we are ready! For preview testing, select **File** › **Run report to** › **Preview** again, and the Designer runs our report to print preview. The final preview of our example report looks like this:



The screenshot shows a software window titled 'NCReport' with a standard toolbar at the top. The main content area displays a report titled 'Product list example by category'. The report is organized into three sections, one for each product category (A, B, and C). Each section contains a table of products with columns for Category, Product name, Product code, Available status, Weight, and Price. The report also includes a 'Weight total' for each category.

Category	Product name	Product code	Available	Weight	Price
Product category: A					
A	Magnetometer	D54/78	yes	0.778	\$15.60
A	Pressostat	M542	no	2.547	\$30.00
Weight total:				3.325	
Product category: B					
B	Oil pump	CT-784	yes	1.510	\$17.00
B	Water pump	RF-800	yes	3.981	\$58.00
B	Erling o-ring	577874	yes	2.887	\$49.00
Weight total:				8.378	
Product category: C					
C	Hydraulic cup	HC55	no	0.435	\$39.00
C	Ballistic rocket	BV01	yes	1.260	\$157.90
C	Wheel	Q185/70	yes	25.554	\$199.00
Weight total:				27.249	

Congratulations! We have created a simple report with one group level. In the next section, we describe how to run this report from your application.

4.5. Integrating NCReport into a Qt application

4.5.1. Adding the NCReport library to an application

To use NCReport from your application, you first have to integrate NCReport into your application project and use the NCReport API classes. There are two ways to do this:

Statically adding source code to your project

Add all of the source code statically to your project and build it together with your application. In this case, you do not need the NCReport shared libraries. To do this, open your `.pro` project file and add the full engine source to the project, as the `src/testapp/testapp.pro` project does by including the `ncreport/ncreport.pri` file.

Linking the report engine as a shared library

Use the NCReport engine as a shared library. To use the NCReport library like any other library in your project, you need to specify it in your project file. The following project example shows the necessary settings:

```
QT = xml sql gui core
TEMPLATE = app
CONFIG += warn_on \
qt \
thread \
release

TARGET = MyApplication
INCLUDEPATH = ../ncreport/includes

HEADERS += ...
SOURCES += ...

win32 {
LIBS += ../lib/NCReport2.lib
}
unix {
LIBS += -lNCReport -L../lib -L/usr/local/bin
target.path = /usr/local/bin
}
```

With Qt5 and Qt6, also add the `printsupport` and `widgets` modules to the `QT` variable. If the engine was built with barcode support, the Zint library (`zint`, or `zint2` on Windows) must also be linked. For more information, see the "Declaring Other Libraries" chapter of the `qmake` manual in the Qt documentation.

4.5.2. Initializing the NCReport class

This step shows you how to initialize the NCReport class.

Includes. First, we have to add the includes. To include the definitions of the module's classes, use the following includes:

```
#include "ncreport.h"
#include "ncreportoutput.h"
#include "ncreportpreviewoutput.h"
#include "ncreportpreviewwindow.h"
```

Creating the NCReport object. We create the report object just like any other QObject-based object:

```
NCReport *report = new NCReport();
```

If the NCReport object was created earlier and passed as a parameter, you should initialize the report by calling the **reset()** method:

```
report->reset();
```

```
//or  
report->reset(true);
```

The **NCReport::reset()** function deletes all object references and prepares the report engine to run a report again. If the parameter is set to `true`, the report parameters and the added data, such as string lists and custom items, are also deleted.

4.5.3. Setting the Report's source

The report source determines where NCReport loads the XML report definition from. Report definitions can be opened from a file, which is suitable in most cases, but they can also be loaded from a table in an SQL database. In our example, we use a file as the report source:

```
report->setReportFile( fileName );
```

This code is equivalent to the following:

```
report->setReportSourceType( NCReportSource::File );  
report->reportSource()->setFileName( fileName );
```

Adding parameters. To add a parameter to NCReport, use the `addParameter()` method. The parameter ID is a string, and the value is a QVariant object.

```
report->addParameter( "id", QVariant("value") );
```

Since the value is a QVariant, it can also be passed directly, without an explicit conversion:

```
report->addParameter( "paramID", "Parameter value" );
```

4.5.4. Running the Report

Now we are ready to run the report to different outputs. To do so, use one of the `runReportTo...`() functions. Running the report to a printer:

```
report->runReportToPrinter();
```

Running the report to PDF:

```
QString fileName("mypdffile.pdf");  
report->runReportToPdf( fileName );
```

Running the report to print preview:

```
report->runReportToPreview();
```

If you run the report to preview, the result is stored in an `NCReportPreviewOutput` object. The report engine does not open the preview window automatically. After the report engine finishes successfully, we need to create an `NCReportPreviewWindow` object for previewing. Before doing so, we check whether a report error occurred.

```
if ( !report->hasError() ) {
    NCReportPreviewWindow *pv = new NCReportPreviewWindow();
    pv->setOutput( (NCReportPreviewOutput*)report->output() );
    pv->setWindowModality( Qt::ApplicationModal );
    pv->setAttribute( Qt::WA_DeleteOnClose );
    pv->show();
} else {
    QMessageBox::warning( this, tr("Error"), report->lastErrorMsg() );
}
```

To get the current output, use the `NCReport::output()` function.



When you run the report to preview, the report output object is not deleted by NCReport. When the `NCReportPreviewWindow` object is destroyed, the output is deleted automatically by its destructor.

4.5.5. Error handling

To catch errors, you can use the following functions:

```
bool error = report->hasError();
QString errmsg = report->lastErrorMsg();
```

Deleting the report object. After running the report, you may delete the report object. When the `NCReport` object is deleted, all of its child objects are deleted as well.

```
delete report;
```



Do not delete the `NCReport` object while the `NCReportPreviewWindow` object still exists. If you want to use the report object again without deleting it, just use the `NCReport::reset()` function.

Chapter 5. Report Items

Report items are the elements you can place in the report sections. Your report content is built from these elements. The following item types are available:

- Simple Static Text (label)
- HTML (Rich) Text
- Data Field
- Line
- Rectangle
- Ellipse / Circle
- Image
- Barcode
- Custom Graphical Content (Graph)
- Table View / Model based Table
- Cross Table
- Graphics View Content
- Chart (optional, HighCharts integration)

The report item buttons in the Designer's toolbar:



5.1. Static Text Item

The static text item (or label) represents simple text with uniform formatting. It cannot be dynamic; its text is saved statically in the report definition. You can use this item for labels, prefixes, suffixes, or any other static content that does not change. Static text can be a single line or a wrapped multi-line text block. The text format (font, color, and alignment) can be set.

To change the text font format and attributes in the Designer, select one or more static text items and use the text formatting toolbars to change the settings.

5.2. HTML (Rich) Text

HTML text items are complex text objects with HTML markup, implemented using the QTextDocument class. They display rich text specified using a subset of HTML 4 markup. The text can be either static or dynamic, so it is possible to load rich text from a data source, a file, or a parameter at runtime.

5.3. Data Field

The field is the most important report element. It can represent one or more pieces of data from a data source, a parameter, a variable, a system variable, or an expression, depending on its **field source type**. Field values can be formatted according to their data type (text, numeric, date, or boolean).

5.4. Line

A graphical line defined by a start point and an end point. The line characteristics are configurable: you can change the line style, width, and color. Lines are static objects.

5.5. Rectangle

A graphical rectangle specified by its position, width, and height. The rectangle characteristics are configurable: you can change the line style, width, and color. Rectangles can be transparent or filled. Rectangles are basically static objects, but with the **Dynamic Position** feature their height and width can be controlled dynamically by a data expression.

5.6. Ellipse

A graphical ellipse or circle specified by its position, width, and height. The ellipse characteristics are configurable: you can change the line style, width, and color. Ellipses can be transparent or filled. Ellipses are basically static objects, but with the **Dynamic Position** feature their height and width can be controlled dynamically by a data expression.

5.7. Image

An image loaded from a file or embedded in the report. The supported formats include PNG, JPEG, BMP, and SVG. You can assign a direct link to a file or use a static image stored in the report. The image is specified by its width and height, and many characteristics are configurable, such as alignment, aspect ratio, and transformation mode. Reports can contain both static and dynamic images (for example, images loaded from a data source column or from a file name stored in the data), so the image element is very flexible.

5.8. Barcode

NCReport provides barcode rendering for more than 50 barcode symbologies (for example, EAN, UPC, Code 128, Code 39, QR Code, Data Matrix, and PDF417), thanks to the integrated Zint back-end library. The barcode item can render a barcode from a static value or from a dynamic source, such as a data source column. See [Chapter 17](#) for details.

5.9. Table View / Model

The Table View item is intended for rendering `QTableView` tables with WYSIWYG print support. It prints tables the same way as they appear in the `QTableView` screen widget, following the formats of the original widget. The cells get their display appearance information from the table's item model. Some basic table settings, such as the header background, line style, and line color, are currently fixed. The table item can also render a `QAbstractItemModel` without a `QTableView`.

5.10. Cross Table

The cross table is also known as a pivot table, a cross-tab report, or rotated data within a table. In cross tables, the data source records are represented as horizontal columns, and the data source columns are printed as the rows of the table. Such tables often include horizontal and/or vertical totals as well.

5.11. Custom Graphics Content

The Graph/Custom item is a special member of the NCReport items. It enables you to render `QPainter`-drawn content in reports using your own C++/Qt code. The typical use of this feature is rendering graphs or similar user-defined content.

5.12. Graphics View Content

The Graphics View item renders the content of a `QGraphicsView` widget, for example a Qt Charts chart view, into the report. The application registers the `QGraphicsView` object in the report definition using the `NCReportDef::addGraphicsView(id, view)` method, and the item refers to the view by its class ID. In the Designer, the item is displayed as a placeholder showing the class ID, because the actual content is available only at runtime.

5.13. Chart

The Chart item renders charts using the HighCharts JavaScript charting library. The chart is defined by an HTML/JavaScript template file and is filled with report data at runtime. The item is available only if NCReport is built with the optional `highcharts` extension (see `src/config.pri`), which requires the Qt WebEngine module. This integration is provided without official support. See [Chapter 18](#) for the available charting options.

Chapter 6. Parameters

Parameters are data passed to the report generator from outside. The application that runs the report passes information as parameters to the NCReport object using the `NCReport::addParameter(...)` method. Parameters are mostly used in SQL queries and in data, template, and script expressions. Fields can also use the `Parameter` source type, so parameter values can be printed directly in the report.

6.1. Parameter Syntax

To embed a parameter into a query or an expression, use the following syntax:

```
${parameterID}
```

Example of using a parameter in an SQL query:

```
SELECT productId, productName FROM db.products WHERE primaryKey=${parameterID}
```

Example of using a parameter in a script expression:

```
"${P{firstname}}" + "${P{lastname}}"
```

Example of using a parameter in a data template expression:

```
Dear ${P{firstname}} ${P{lastname}},
```

6.2. Testing Parameters

The following example uses the SQL data source of the report built in the Designer part of this book. To modify the SQL query, select **Report > Data sources....** Modify the SQL query of the data source as follows:

```
SELECT ProductID, ProductName,
QuantityPerUnit, UnitPrice, QuantityPerUnit*UnitPrice as value
FROM products
WHERE ProductID > ${P{prodID}}
ORDER BY ProductName
```

Then add a parameter with the ID `prodID` to the report. The report runner dialog of NCReport Designer lets you add parameters for testing. To open the runner dialog, select **File > Run report....** Click the **[Add]** button, set the name to `prodID`, and set the value to `70`. After running the report to the preview, you get the following result:

The screenshot shows the NCReport 2.6.1 preview window. The report displays a list of products with their respective prices and quantities. The data is organized into columns: an index column, a product name column, and three numerical columns representing price, quantity, and total value. The report is on page 2 of 2.

42	Product: Singaporean Hokkien Fried Mee	14.00	12.00	168,00
20		81.00	60.00	4 860,00
21		10.00	9.00	90,00
61	Product: Sirop d'arable	28.50	24.00	684,00
46	Product: Spegesild	12.00	10.00	120,00
35		18.00	16.00	288,00
62	Product: Tarte au sucre	49.30	41.00	2 021,30
19		9.20	8.00	73,60
29		123.79	80.00	9 903,20
14		23.00	21.00	483,00
54	Product: Tourtiere	7.45	6.00	44,70
23		9.00	8.00	72,00
7		30.00	25.00	750,00
50	Product: Valkoinen suklaa	16.25	14.00	227,50
63	Product: Vegie-spread	43.90	37.00	1 624,30
64	Product: Wimmers gute Semmelknadel	33.25	28.00	931,00
47	Product: Zaanse koeken	9.50	8.00	76,00
Total value:				98 013,66

Parameters are evaluated in SQL queries, scripts, and print when expressions in all reports. Design-time parameters can also be managed using the **Report > Parameters...** dialog.

6.3. API code: Passing parameters to the report

```
NCReport* report = new NCReport(parent);
QString companyName = tr("ABC Ltd");
report->addParameter("myid", companyName);
```

Part II: Designer

NCReport Designer is a standalone GUI application for creating and modifying report templates. This part introduces the Designer's user interface, shows how to design a report step by step, and closes with a complete reference of the Designer's tools and settings.

Chapter 7. Getting Started with NCRReport Designer

This chapter covers the fundamental steps that most users take when creating reports with NCRReport Designer. We introduce the main features of the tool by creating a simple report that you can use with the NCRReport engine.

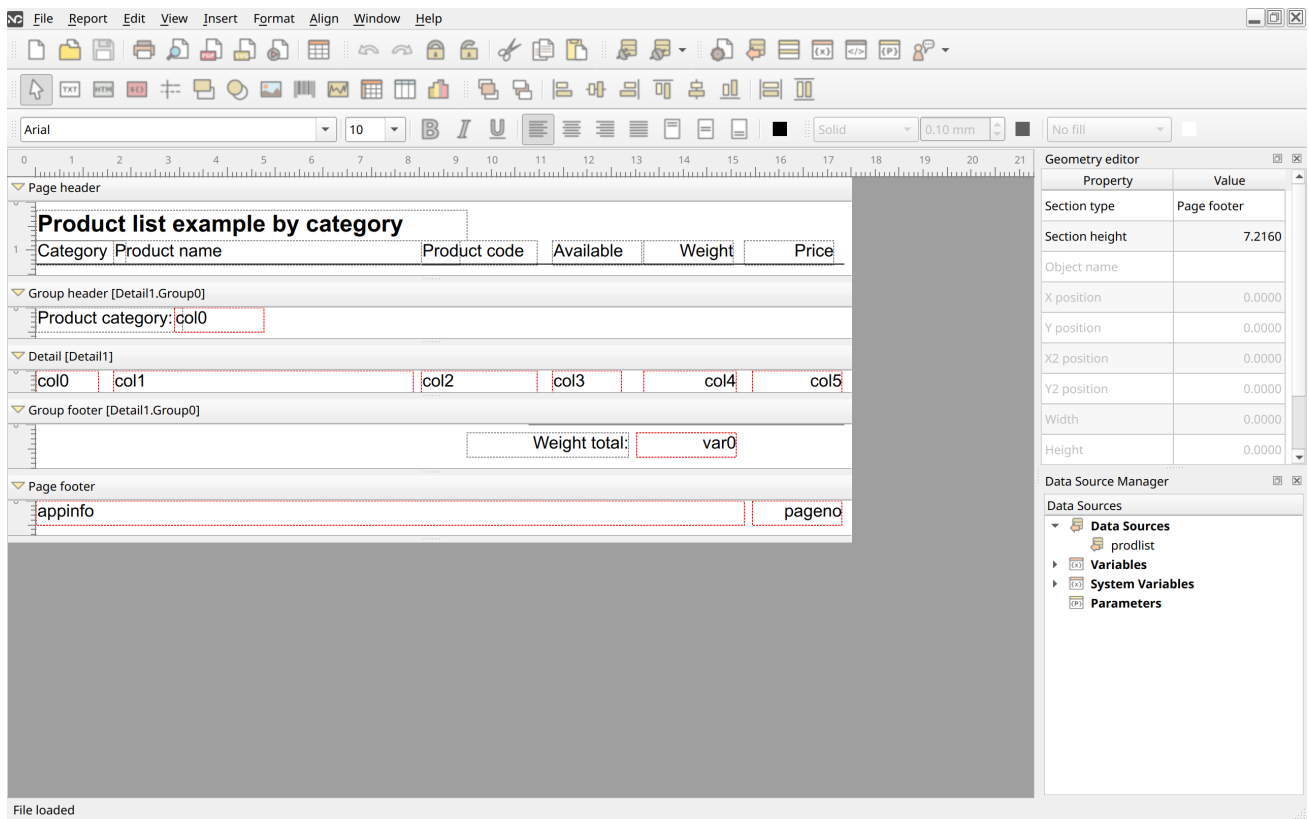
7.1. Launching Designer

The way you launch NCRReport Designer depends on your platform:

- On Windows, open the Start menu and click the NCRReport Designer shortcut created by the installer.
- On Unix or Linux, you may find an NCRReport Designer icon on the desktop or in the desktop start menu. You can launch the Designer from this icon. Alternatively, enter `./NCRReportDesigner` in a terminal window in the `bin` directory of NCRReport.
- On macOS, double-click the NCRReportDesigner application in the Finder.

7.2. The User Interface

NCRReport Designer has a standard multiple document interface (MDI): every opened report is displayed in its own window inside the main window, so you can work on several reports at the same time. The main window consists of a menu bar, toolbars, the MDI area that contains the report windows, and two dock windows on the right side: the **Geometry editor** for editing the position and size of items and sections, and the **Data Source Manager** that lists the data sources, variables, system variables, and parameters of the active report. Both dock windows can be shown or hidden from the **View** menu.



7.3. NCReport Designer Main Window

The menu bar provides all the standard actions for opening and saving report files, managing report sections, using the clipboard, and so on. The toolbars display the most common actions used when editing a report; these are also available from the main menu.

Most features of NCReport Designer are accessible from the menu bar or the toolbars. Some features are also available from context menus that can be opened over the report sections and items. On most platforms, the right mouse button opens context menus.

The main menu contains the following menus:

File

File operations: **File** > **New** (**Ctrl** + **N**), **File** > **Open...** (**Ctrl** + **O**), **File** > **Open from database...**, **File** > **Save** (**Ctrl** + **S**), **File** > **Save As...**, and the **File** > **Recent files** submenu. The **File** > **Run report to** submenu runs the current report directly to a selected output: Preview (**Ctrl** + **P**), Qt preview, Printer, PDF file, SVG file, Text file, HTML file, Excel file, or Image file. **File** > **Run report...** (**Ctrl** + **R**) opens the report runner dialog, in which you can choose the output, add test parameters, and set other run options. The File menu also contains **File** > **Copy source to clipboard** (copies the XML source of the report), **File** > **Preferences...**, and **File** > **Exit** (**Ctrl** + **Q**).

Report

The settings of the current (active) report and its sections: **Report** > **Report and page settings...**, **Report** > **Data sources...**, **Report** > **Details and grouping...**, **Report** > **Variables...**, **Report** > **Scripts...**, and **Report** > **Parameters...** **Report** > **Edit selected item...** (**Ctrl** + **1**) and **Report** > **Edit**

selected section... (**Ctrl** + **2**) open the settings dialog of the selected item or section. **Report** › **Database connection manager...** manages the SQL database connections used for testing; the **Report** › **Database connections** submenu lists the defined connections, and the **Report** › **Report language** submenu switches the current language of a multilingual report. **Report** › **Test data editor...** (**Ctrl** + **T**) opens the test data editor.

Edit

Edit › **Undo** (**Ctrl** + **Z**), **Edit** › **Redo** (**Ctrl** + **Y**), **Edit** › **Lock items** (**Ctrl** + **L**) and **Edit** › **Unlock items** (**Ctrl** + **Shift** + **L**) for protecting the selected items from accidental changes, **Edit** › **Set zone ID...** for assigning the same zone ID to all selected items, and the clipboard actions **Edit** › **Cut**, **Edit** › **Copy**, and **Edit** › **Paste**.

View

View › **Zoom in**, **View** › **Zoom out**, and **View** › **Zoom one** (reset zoom), **View** › **Show grid** and **View** › **Snap to grid**, and the **View** › **Geometry editor** and **View** › **Datasource manager** items, which show or hide the corresponding dock windows.

Insert

The report items that can be added to the report: Label, Text, Field, Line, Rectangle, Ellipse, Image, Barcode, Graph, Cross tab, Table View, and Graphics View (and Chart, if the HighCharts extension is enabled). The **Selection** item switches back to selection mode.

Format

Text formatting actions for the selected text items: Bold, Italic, Underline, horizontal alignment (Left, Center, Right, Justify), vertical alignment (Top, Middle, Bottom), and Text color.

Align

Arrangement and alignment actions for the selected items: Bring to Front, Move to Back, Align left, Center horizontally, Align right, Align top, Center vertically, Align bottom, Set widths the same, and Set heights the same.

Window

Window management actions for the report windows that are open concurrently: Close (**Ctrl** + **F4**), Close All, Tile, Cascade, Next, and Previous, followed by the list of open windows.

Help

The About box of the application and the About Qt box.

The toolbars (File, Edit, Database Connections, Report, Tools, Align, Text properties, and Shape properties) provide quick access to the most frequently used actions of these menus.

7.4. Geometry editor

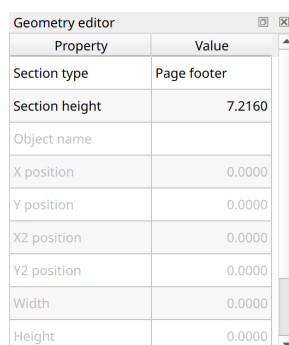
The geometry editor is a dock window that displays and edits the position and size of the current report item or section. To show or hide it, toggle the **View** › **Geometry editor** menu item. Items and

sections become current when you click them.

The editor is a two-column property table (**Property** and **Value**) that contains the following properties:

- **Section type** and **Section height**: the type and the height of the current section. The section height can be edited.
- **Object name**: identifies the current item (read-only).
- **X position** and **Y position**: the position of the item relative to its section, in millimeters.
- **X2 position** and **Y2 position**: the end point coordinates of line items.
- **Width** and **Height**: the size of the item, in millimeters.
- **Z-value**: the stacking order of the item.

You can type numeric size or position values into the editable cells. Any change made to an item's properties updates it immediately.



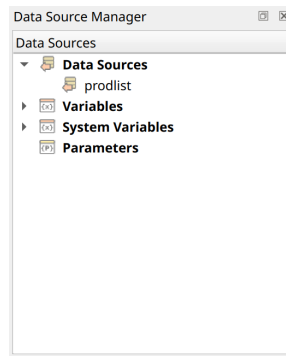
Property	Value
Section type	Page footer
Section height	7.2160
Object name	
X position	0.0000
Y position	0.0000
X2 position	0.0000
Y2 position	0.0000
Width	0.0000
Height	0.0000

7.5. Data Source Tree

The Data Source Tree (shown as the **Data Source Manager** dock window) is located on the right side of the main Designer window. To show or hide it, toggle the **View > Datasource manager** menu item. The tree makes it easy to add fields to the report with a simple drag-and-drop action. It contains the following top-level nodes:

- **Data Sources**: the data sources defined in the report. If the data columns are available at design time, they appear under the appropriate data source item.
- **Variables**: the variables defined in the report.
- **System Variables**: the built-in system variables of the report engine, such as the current page number.
- **Parameters**: the parameters defined at design time.

The tree is updated whenever you add or modify a data source, a variable, or a parameter in the report. Therefore, we recommend that you begin building the report by defining the data sources first.



7.5.1. Adding a Field using the Data Source Tree

To add a field to any report section, drag the selected data source column and drop it onto the section at the position you want. A Field item is created at the drop position (the position of the mouse pointer is used), with the appropriate field source type and column already set. Variables, system variables, and parameters can be dragged to the sections in the same way; the created field then has the [Variable](#), [System variable](#), or [Parameter](#) source type.

7.5.2. Editing from the Data Source Tree

Double-clicking an item in the tree opens the related settings dialog: a data source opens the data source settings dialog, a variable opens the variable settings dialog, and a parameter opens the parameter editor, with the clicked item selected.

7.6. Field Expression Builder

When you work with Field items, a helper tool is available for creating correct field expressions. To open the Expression Builder, click the **[Build expression...]** button in the field settings dialog. Use the combo boxes to select the desired element, and add it by clicking the small **[...]** button beside the combo box. The expression is then inserted into the text area at the cursor position. Depending on the type of the inserted element, the expression builder applies the correct syntax. The dialog also contains a data source tree, which you can use for the same purpose: double-click a data source column, a parameter, or a variable in the tree to insert it.

7.6.1. Expression Builder Dialog Controls

The following list describes the combo boxes of the dialog and their purpose.

Field source type

Selects the field source type. Do this first, because the source type determines which other controls are available.

Data source

Selects a data source from all data sources available in the report. To insert the data source, click the button beside the combo box.

Column

Selects a data source column. To insert the column, click the button beside the combo box.

Variable

Selects any available variable, including the system variables. To insert the variable, click the button beside the combo box.

Data source function

Selects one of the available data source level functions. To insert the function expression, click the button beside the combo box.

Value function

Selects one of the available data source value functions. To insert the function expression, click the button beside the combo box.

Parameter

Selects one of the parameters defined at design time. To insert the parameter, click the button beside the combo box.

7.6.2. Logical Operation Buttons

The dialog also has operator buttons that speed up editing a script or a logical expression: arithmetic operators (+, -, *, /), comparison operators (==, !=, <, <=, >, >=), assignment (=), and the logical operators AND, OR, XOR, and NOT. When you click a button, the corresponding operator is inserted into the text area. The [**Clear**] button clears the text area.

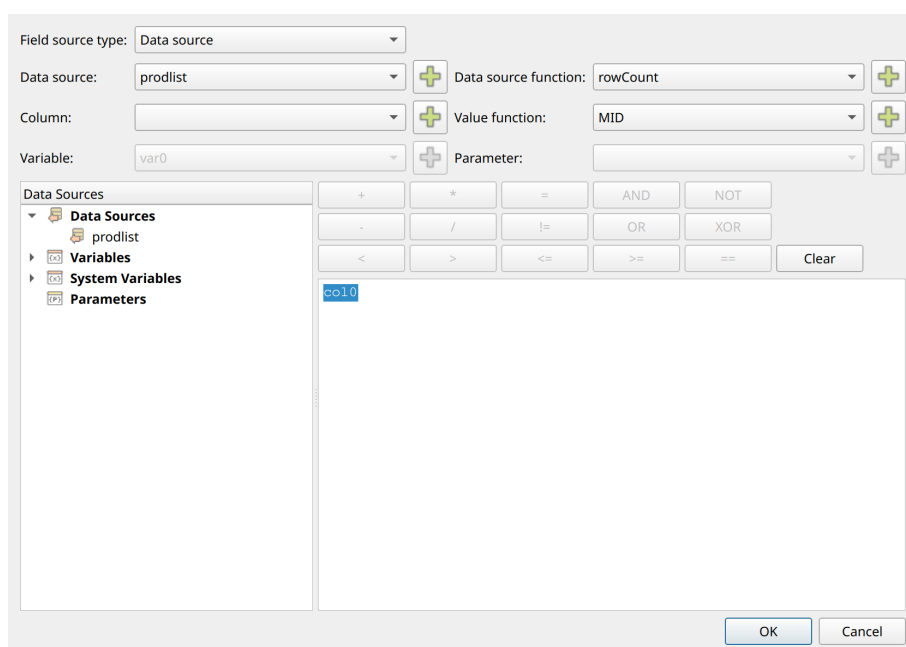


Figure 4. Expression Builder Dialog

Chapter 8. Designing a Report

In this chapter, we look at the main steps users take when creating a new report with NCReport Designer. Creating a new report usually involves the following activities:

- Deciding what kind of report structure to create.
- Deciding which kinds of data sources to use.
- Connecting to an SQL database, if an SQL data source is used.
- Defining the data sources.
- Setting up the page and report options and the report sections that are needed.
- Deciding which kinds of items to use in the different sections.
- Composing the report layout by adding report items to the sections.
- Adding variables, groups, and parameters, if needed.
- Testing the report.

You may prefer to perform these activities in a different order. We present them in roughly the order above and leave it up to you to find the approach that suits you best. To demonstrate the process, we look at the steps needed to create a simple product list report with NCReport Designer. The report draws its data from an SQL database data source, which lets us illustrate some additional features of the tool, such as database connections and SQL queries.

8.1. Connecting to a database from Designer

NCReport Designer lets you test the report from inside the Designer. Since our example report uses an SQL data source on a MySQL database, we first have to connect to the "northwind" database. SQL database connections are managed by the connection manager of the Designer. Select **Report > Database connection manager....** The **Database connections** dialog then appears. In this dialog you can add one or more SQL connections.

The following options are available for connections:

Database driver

The Qt SQL database driver to use, for example QMYSQL, QPSQL, QSQLITE, or QODBC.

Connection name

The name of the database connection. Qt uses this name in the `QSqlDatabase::addDatabase(...)` function. This value must be specified in order to identify the connection.

Host name (IP address)

The name or IP address of the database host.

Database name

The name of the database.

Username

The user name of the connection.

Password

The password of the connection.

Port

The port number of the connection. If **Default** is selected, the default port of the driver is used.

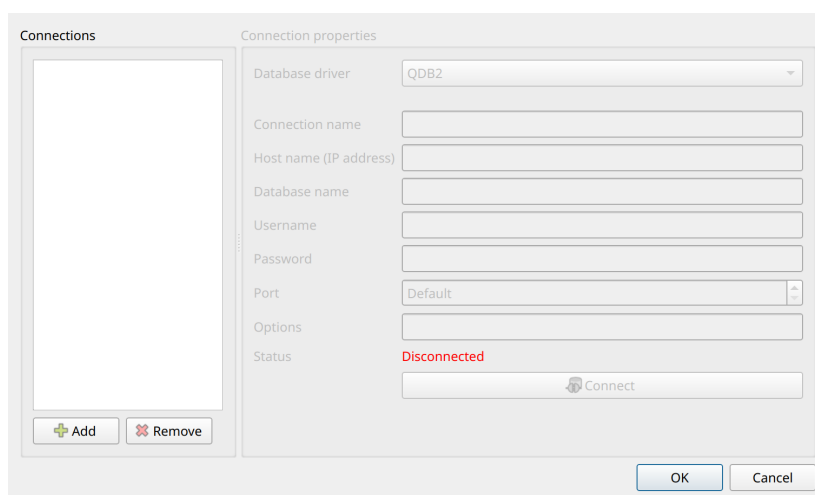
Options

Driver-specific connection options.

Status

Shows whether the connection is currently connected or disconnected.

8.1.1. SQL connection dialog



- **[Connect]** Tries to establish the connection.
- **[Add]** Adds a new connection and enables its options for editing.
- **[Remove]** Removes the connection selected in the list.
- **[OK]** Saves your connection settings.
- **[Cancel]** Closes the dialog without saving any changes and returns you to the Designer main window.

After you have specified the parameters of the new connection, click the **[Connect]** button to establish the connection. If the connection succeeds, your report is ready to run. Before running the report, we rename our connection to **northwind**; the connection name of our data source must be changed to **northwind** as well. To do so, open the **Report > Data sources...** dialog again and change the connection name of the data source to **northwind**.

The defined connections are also listed in the **Report > Database connections** submenu and on the **Database Connections** toolbar.



You do not need an SQL connection if you use only non-SQL data sources in your report, for example a text, string list, XML, or item model data source.

8.1.2. Connection names (ID)

Each SQL data source has its own connection name, which is used for running its SQL query. The QSqlDatabase connection must have the same name as the one set in the data source. In the Designer, create a connection in the connection manager with the same connection name you defined in the report.

8.1.3. Using the default connection

When the SQL data source does not have a connection name, the report engine runs the SQL query using the default database connection. To test such a report, create a connection in the Designer's connection manager with `default` as the connection name. This connection is opened as the default connection, so you can run your report from the Designer with an empty connection name in the data source.

8.2. Beginning a new report

To begin a new report definition, select **File** > **New** (**Ctrl** + **N**) or click the **[New]** toolbar button. A new report window opens in the MDI area with a temporary name (`report1.ncr`, `report2.ncr`, and so on), which you can change when you first save the report. By default, the new empty report contains a page header, a detail, and a page footer section. The default font of the report is taken from the Designer preferences (**File** > **Preferences...**).

When you save the report with **File** > **Save** or **File** > **Save As...**, you can use either the `.ncr` or the `.xml` file extension; both are saved in the same XML format.

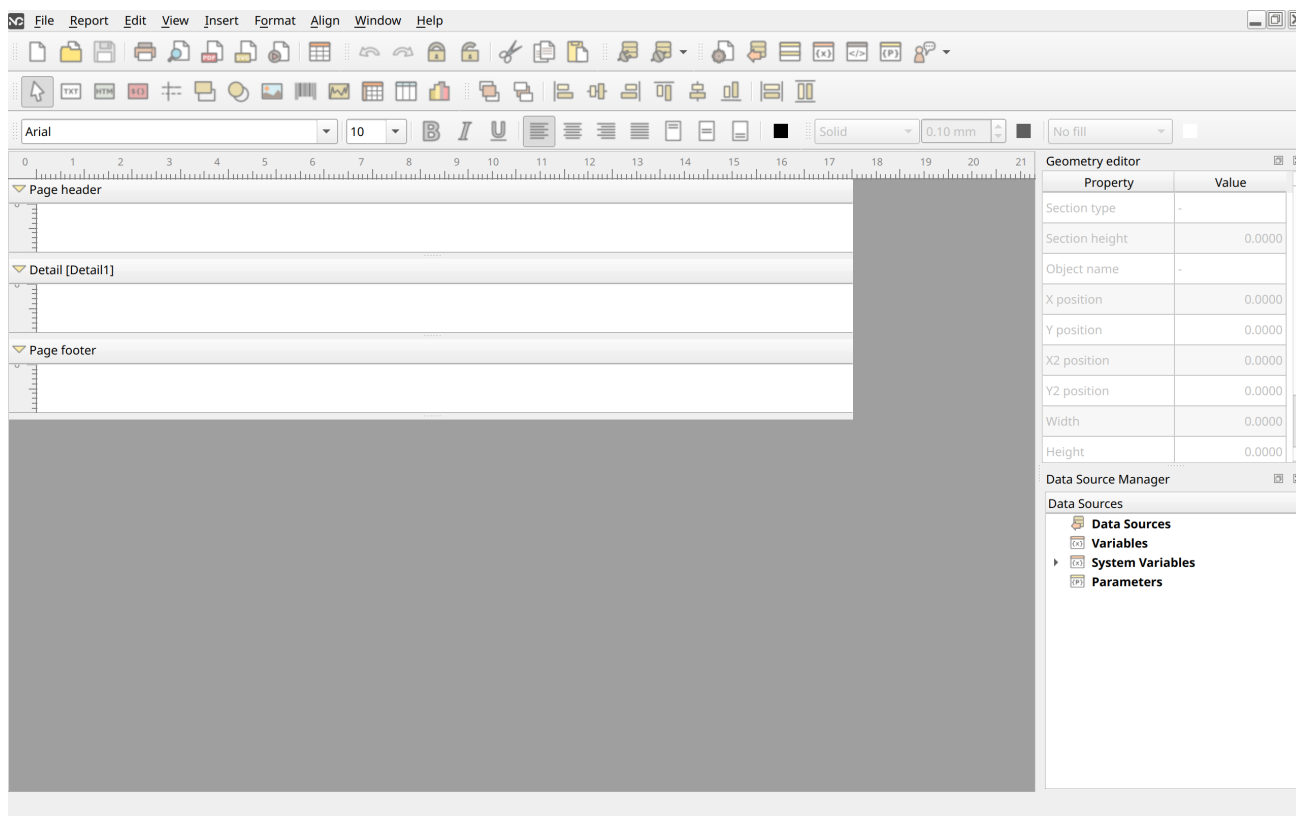


Figure 5. New report

8.3. Report sections

Report sections are the function-specific areas of a report. Reports are built from sections, many of which are recurring areas, such as the detail, header, or footer sections. The most important section is the Detail, since it contains the fields that change row by row. Each section can contain all kinds of report items. An item's coordinates are always relative to its parent section. A report can contain the following sections: report header, report footer, page header, page footer, group headers and footers, and details.

To change the height of a section, drag the resizer bar under the section area to the size you want, or type the height value in millimeters in the geometry editor, if it is enabled. To make a section current, click an empty area of the section. The settings dialog of the current section can be opened with **Report > Edit selected section...** (**Ctrl** + **2**).

8.3.1. Detail

The core information in a report is displayed in its Detail section. This is the most important section of the report, since it contains the row-by-row data from the data source. The Detail section has the following characteristics:

- It generally prints in the middle of a page (between the headers and footers).
- It contains the core information of a report.
- It displays multiple rows of data returned by a data source.

- It generally contains fields.
- Multiple independent details are allowed in one report; they are printed one after the other.
- Every detail is assigned to one data source.

8.3.2. Page header

The page header contains the page headings. The page header has the following characteristics:

- It always prints at the top of a page.
- It always contains the first information printed on a page.
- It displays only one (the current) row of data returned by a data source.
- Only one page header is allowed per page.

Most reports need a page header. To add or remove the page header, select **Report > Report and page settings...**; a dialog then appears in which you can set the page options of the current report. To enable or disable the page header, use the **Page header** check box on the **Sections** tab.

8.3.3. Page footer

Page footers are commonly used to close the pages. The page footer has the following characteristics:

- It always prints at the bottom of a page.
- It displays only one (the current) row of data returned by a data source.
- Only one page footer is allowed per page.

The page footer is usually used to display information such as the page number, the report title, and so on. Most reports need a page footer. To enable or disable the page footer, use the **Page footer** check box in the **Report > Report and page settings...** dialog.

8.3.4. Report header

The report header contains the report headings. The report header has the following characteristics:

- It is always printed after the page header.
- It is printed only once, at the beginning of the report.
- It displays only one (the current) row of data returned by a data source.

To enable or disable the report header, use the **Report header** check box in the **Report and page settings** dialog, which you can open by selecting **Report > Report and page settings....**

8.3.5. Report footer

The report footer is commonly used to close the report. The report footer has the following

characteristics:

- It is always printed at the end of the report, before the page footer.
- It displays only one (the current) row of data returned by a data source.
- Only one report footer is allowed per report.

To enable or disable the report footer, use the **Report footer** check box in the **Report and page settings** dialog, which you can open by selecting **Report › Report and page settings....**

8.4. Setting up page and report options

The page and report options of the current report are specified in the **Report and page settings** dialog. To open it, select **Report › Report and page settings....** The options are arranged on several tab pages.

8.4.1. General

Report name

The name of the report. This is an informative option only; it is not used by the report generator.

Report type

Report is a normal report. **Text document** is a limited mode in which the report contains HTML text items only and the result is a paginated HTML document. **Label** mode prints the records of the first detail as labels arranged in rows and columns; the label layout is set on the **Label** tab (see [Chapter 24](#)).

File encoding

The encoding of the XML file, used when the report definition file is opened or saved. In most cases UTF-8 is suitable, but for special international characters you may choose a different encoding.

Resolution

The print resolution (dpi) of the report for the printer and file outputs. **Auto** means that the default resolution of the output is used.

Background color

The background color of the report. This option is currently not used.

Report output file base name

The base name of the output file, used when the report is generated to file outputs.

Evaluator empty value

The text that the expression evaluator substitutes when a referenced value is empty or cannot be evaluated.

Default Font

The font family, point size, and style (bold, italic, underline, strikeout) used by default when text, label, or field items are added to the report. Each item may override this setting.

Working mode

Report processing options:

- **Update current data source variables only:** only the variables that belong to the currently processed data source are updated.
- **Subreports start on new page** and **Subreport resets page number:** options for reports that contain subreports.
- **Double pass mode:** enables two-pass report generation, which makes it possible to print the total number of pages.
- **Variable error checking:** report processing stops with an error if a variable cannot be evaluated.
- **Data trimming:** enables the data trimmer for special calculations.

The screenshot shows the 'General' tab of the NCReport configuration dialog. It is divided into three main sections:

- General options:** Includes fields for 'Report name' (Untitled), 'Report type' (Report), 'Report output file base name', 'File encoding' (utf8), 'Background color' (white), 'Resolution' (Auto), and 'Evaluator empty value'.
- Default Font:** Includes a 'Font family' dropdown (Arial), 'Point size' (10), and 'Style' checkboxes for Bold, Italic, Underline, and Strikeout.
- Working mode:** Includes checkboxes for 'Double pass mode', 'Subreports start on new page', 'Subreport resets page number', 'Data trimming' (checked), 'Update current data source variables only', and 'Variable error checking'.

At the bottom right are 'OK' and 'Cancel' buttons.

8.4.2. Page size and margins

Page size

The size of the page. The combo box lists the standard paper size names. To use a non-standard size, select **CUSTOM** and enter the **Width** and **Height** of the page in millimeters.

Endless paper

The page height is adjusted to the length of the report content, which is useful for continuous (roll) paper.

Full page mode

Sets the printer to full page mode, in which the page coordinates refer to the whole paper rather

than the printable area.

Margins

The top, bottom, left, and right margins of the page in millimeters.

Orientation

The orientation of the page: Portrait or Landscape.

The screenshot shows the 'Page size and margins' dialog box with the following settings:

- Page size:**
 - Size: A4 (dropdown)
 - Width: 210.00 mm (spin box)
 - Height: 297.00 mm (spin box)
 - ☒ Full page mode
 - ☐ Endless paper
- Margins:**
 - Top: 15.00 mm (spin box)
 - Bottom: 15.00 mm (spin box)
 - Left: 20.00 mm (spin box)
 - Right: 20.00 mm (spin box)
- Orientation:**
 - ☒ Portrait
 - ☐ Landscape

At the bottom right are 'OK' and 'Cancel' buttons.

8.4.3. Sections

Headers and footers

Use the check boxes to enable or disable the page header, page footer, report header, and report footer. To change the height of these sections, use the spin boxes next to the check boxes. You can also change the section heights by dragging with the mouse or by using the geometry editor.

Zone spacing

The vertical space between the zones of a section. See the chapter about zones for details.

Page break is allowed between zones

If enabled, a page break may occur between the zones of a section.

General Page size and margins Sections Language HTML options Table of contents Label Custom data

Headers and footers

☒ Page header 15.23 mm ☐ Report header 0.00 mm

☒ Page footer 7.22 mm ☐ Report footer 0.00 mm

Zone settings

Zone spacing: 0.00 mm

☐ Page break is allowed between zones

OK Cancel

8.4.4. Language

Report languages

The languages of a multilingual report. Select a language in the combo box and click **[Add]** to add it, or click **[Remove]** to remove the selected language. The order of the languages can be changed with the **[Up]** and **[Down]** buttons.

Multi language role if missing translation

Specifies what is printed when a translation is missing: the text in the default language (**Use default language**) or nothing (**Leave blank**).

General Page size and margins Sections Language HTML options Table of contents Label Custom data

Report languages

English Add

Default Remove

Up

Down

Multi language role if missing translation

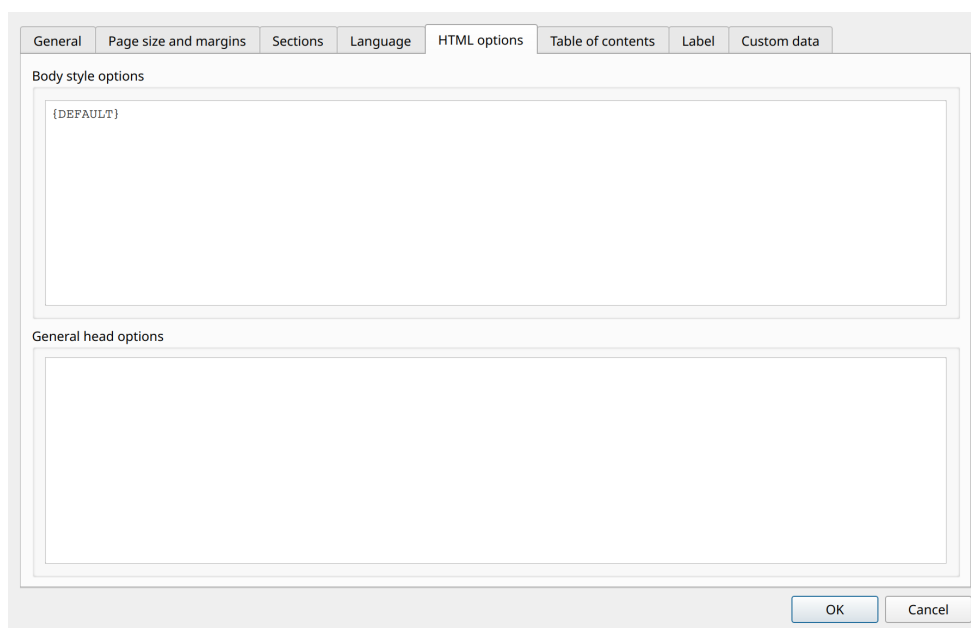
☒ Use default language ☐ Leave blank

OK Cancel

8.4.5. HTML options

Body style options and General head options

Custom style and head content used when the report is generated to HTML output.



The screenshot shows a dialog box with several tabs: General, Page size and margins, Sections, Language, HTML options (selected), Table of contents, Label, and Custom data. The 'HTML options' tab contains two text areas. The first, labeled 'Body style options', contains the text '{DEFAULT}'. The second, labeled 'General head options', is empty. At the bottom right are 'OK' and 'Cancel' buttons.

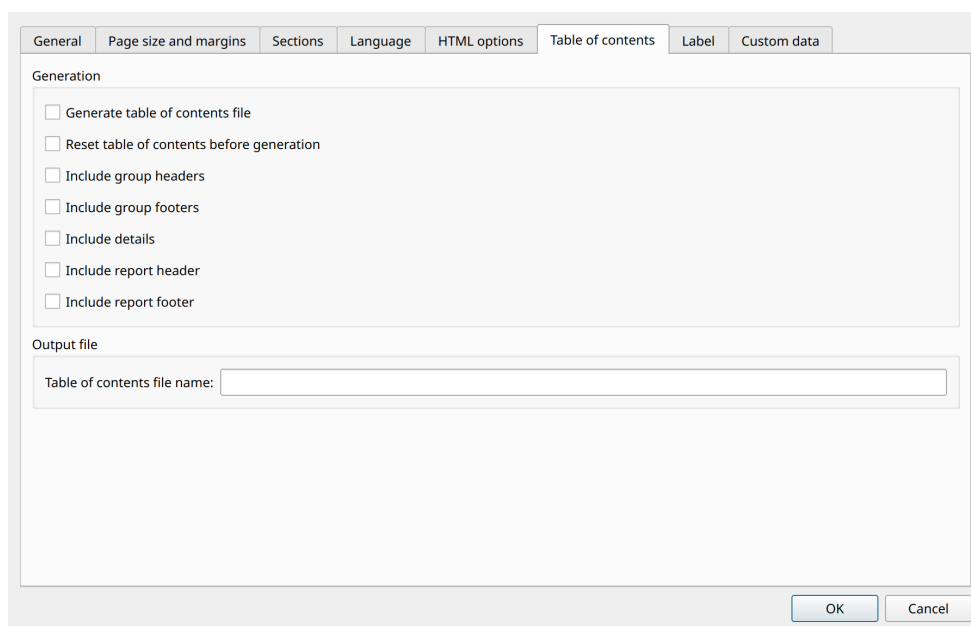
8.4.6. Table of contents

Generation

Enables the generation of a table of contents file (**Generate table of contents file**), optionally resetting it before generation, and specifies which sections are included: group headers, group footers, details, report header, and report footer.

Output file

The name of the table of contents file.



The screenshot shows a dialog box with tabs: General, Page size and margins, Sections, Language, HTML options, Table of contents (selected), Label, and Custom data. The 'Table of contents' tab has two sections. The 'Generation' section contains a list of checkboxes: 'Generate table of contents file', 'Reset table of contents before generation', 'Include group headers', 'Include group footers', 'Include details', 'Include report header', and 'Include report footer'. The 'Output file' section contains a text field labeled 'Table of contents file name:'. At the bottom right are 'OK' and 'Cancel' buttons.

8.4.7. Label

These options are used when the report type is **Label**:

Direction

The order in which the labels are filled: Horizontal or Vertical.

Label size

The width and height of one label in millimeters.

Gaps

The horizontal and vertical gaps between the labels in millimeters.

Dimension

The number of label rows and columns on a page.

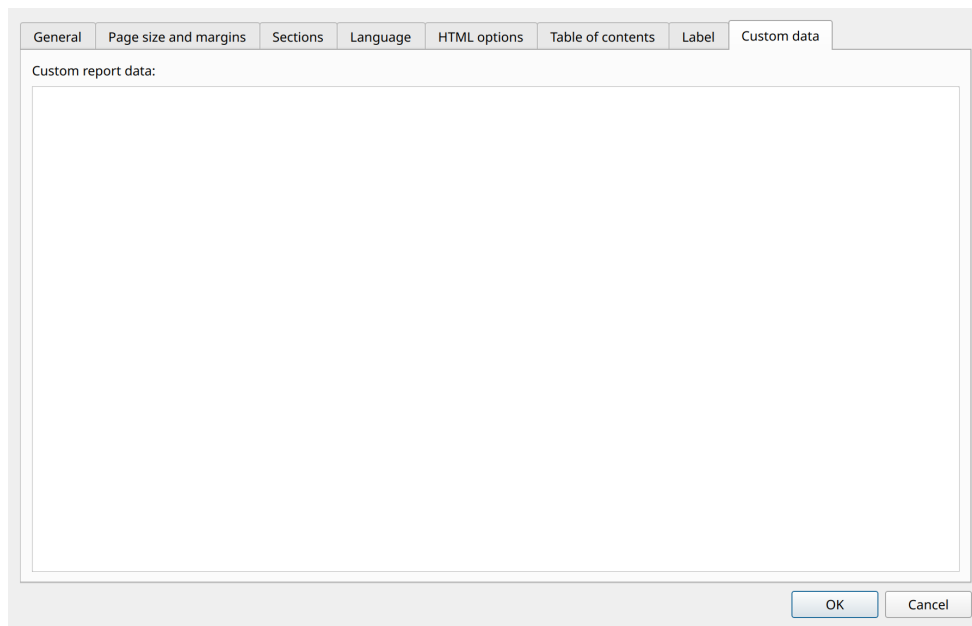
In label mode, the label size and the number of rows and columns must be set, and the page and report headers and footers must be disabled; otherwise the dialog displays an error message.

The screenshot shows the 'Label' tab of a configuration dialog. The dialog has several tabs at the top: General, Page size and margins, Sections, Language, HTML options, Table of contents, Label (selected), and Custom data. The 'Label' tab contains four sections: 'Label size' with 'Width' and 'Height' set to 0.00 mm; 'Gaps' with 'Horizontal gap' and 'Vertical gap' set to 0.00 mm; 'Dimension' with 'Number of rows' and 'Number of columns' set to 0; and 'Direction' with 'Horizontal' selected via a radio button. At the bottom right are 'OK' and 'Cancel' buttons.

8.4.8. Custom data

Custom report data

Free-form custom data stored in the report definition, which can be used by the application.



The following buttons are available for applying or canceling the settings:

- **[OK]** Applies your settings.
- **[Cancel]** Closes the dialog without saving any changes and returns you to the Designer main window.

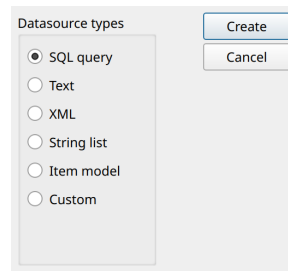
Specify the report page properties in the **Report and page settings** dialog and confirm the settings by clicking the **[OK]** button.

8.5. Adding data sources

At the very beginning, you have to decide which data source type or types you intend to use in the report. Since the report generator builds a printable representation of data from a data source, at least one data source must be defined in the report. Data may be fetched by an SQL query using Qt's SQL database drivers, or from other sources that do not require an SQL connection, such as a text, XML, a string list, an item model, or a custom-defined data source. A report can contain multiple data sources, and each detail is connected to one selected data source. A data source does not have to be assigned to a detail; in this case, you can use the unassigned data source as a single-record (first row) source of data. See the details later.

To add a data source to your report, select **Report > Data sources....** A dialog then appears in which you can add and remove data sources. To add a new data source, click the **[Add]** button and then select the data source type from the list of available types: SQL query, Text, XML, String list, Item model, or Custom.

8.5.1. Data source types dialog



In our example, we choose the SQL query data source type. After you click the **[Create]** button, a new SQL query data source is added to the list in the dialog. You can then specify the data source options. The following properties are available for data sources:

Data source ID

This string property identifies the data source. You refer to the data source by this ID.

Data source type

The type of the data source you chose earlier. It cannot be changed after the data source has been added to the list.

Location type

Describes where the data or the SQL query can be found: inside the report file or in an external file. Its value may be Static, File, Http, Ftp, or Parameter. The Http and Ftp types are currently not supported for SQL queries. Its meaning differs slightly for the different types of data source. For an SQL query, the Static location type is suitable; it means that the SQL query is saved statically in the report file. The Parameter type means that the data is passed to NCReport as a parameter. For example, a QString text or an SQL query can be added as a parameter, depending on the data source type.

File name

When a non-Static location type is selected, you can specify the name of the file that contains the data.

Opening/running role

Specifies when the data source is opened: at the beginning of the report, as a child data source (subquery), when a group changes, or for subreport iteration.

Connection Name

The ID of the SQL database connection. This name is the same ID that is used in the `QSqlDatabase::addDatabase()` function to identify the database connection. When you add a database connection in your application before running the report, you must use this connection name.

Use private database connection

If you want the SQL data source to use its own database connection, select this check box (on the **Database connection** tab). The private connection parameters then become enabled, and you can specify the required properties of the SQL connection: driver, host, database, username, password, and port (optional).

Query

A text area in which you can edit the SQL query. In almost every case it is a `SELECT...FROM` statement using the SQL syntax of the specified database. Only one SQL query is allowed per data source. The query can contain parameters (see later); use the **[Insert Parameter]** button to insert one, and the **[Test Query]** button to try the query.

8.5.2. SQL data source

In our example, we keep the default data source ID and choose the Static location type. We set the connection name to `Con0`. Then we specify the SQL query.



This example requires a running MySQL database server with the northwind demo database and its tables. SQL script files for creating the sample database and tables are included in the `sql` directory of the NCReport distribution.

Let's use this simple query:

```
SELECT ProductID, ProductName, QuantityPerUnit, UnitPrice,
QuantityPerUnit*UnitPrice as value
FROM products
WHERE ProductID>20
ORDER BY ProductName
```

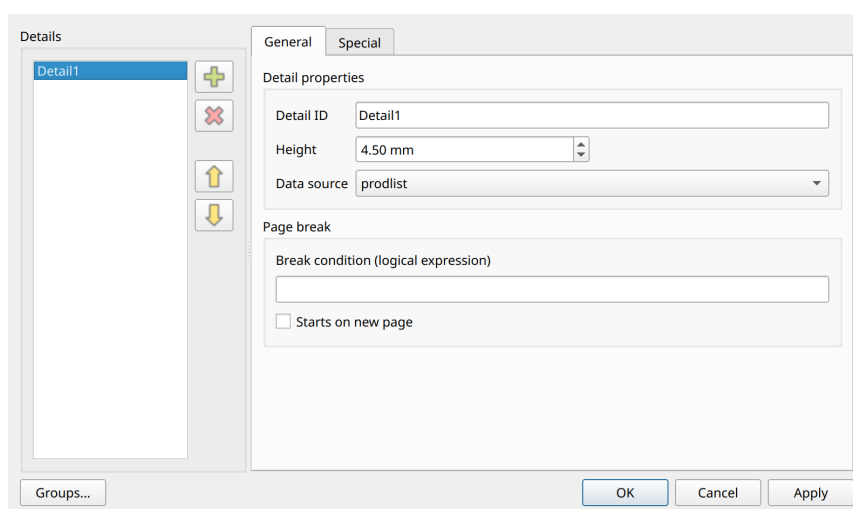
The following buttons are available for applying or canceling settings:

- **[OK]** Applies your data source settings.
- **[Cancel]** Closes the dialog without saving any changes and returns you to the Designer main window.

Confirm the data source settings by clicking the **[OK]** button.

8.6. Assigning a data source to the Detail

To assign the data source we defined earlier, select **Report > Details and grouping....** A dialog then appears in which you can manage the detail sections of the report. A default detail named **Detail1** is already defined. You can rename it by changing the **Detail ID**. Select the previously defined data source from the **Data source** combo box, which contains all data sources defined in the report. A detail must have a data source assigned for the report to work.



The main options of the Detail settings dialog are the following:

Detail ID

The name of the detail section.

Height

The height of the detail section in millimeters. You can also change the height by dragging with the mouse or by using the geometry editor.

Data source

The data source assigned to the detail. Previously defined data sources can be selected in the combo box.

Page break

A break condition (logical expression) that forces a page break, and the **Starts on new page** option.

Groups...

Opens the group management (**Data grouping**) dialog of the selected detail.

The **Special** tab contains further options: a secondary data source, the parent section ID, alternating row background colors (odd or even rows), and repeating the detail a given number of times.

You can add more details with the **[Add]** button or remove an existing detail with the **[Remove]** button. At least one detail section must exist, so the only detail cannot be removed.

The following buttons are available for applying or canceling settings:

- **[OK]** Applies your settings.
- **[Cancel]** Closes the dialog without saving any changes and returns you to the Designer main window.

Confirm the detail settings by clicking the **[OK]** button.

8.7. Adding report items

After we have defined the data source and specified the report options, we can design the report by adding items to the sections. The **Insert** menu and the **Tools** toolbar display the report items that can be used when designing a report. Let's summarize the various report items of NCReport:

Label

The Label represents simple static text. Label items are used to display descriptive information in a report, such as titles and headings. Labels are static items; their values do not change when the report is rendered.

Text

The Text item displays HTML (rich) text. Its content can be static or loaded dynamically, for example from a data source.

Field

The Field is the core report item. By data type, fields may be text, numeric, date, or boolean. Field items are used to pull dynamic data into a report from the specified data source, such as a database. For example, a Field item may be used to present SQL data, variables, and parameters. NCReport handles data formatting for the different types of field, such as numbers, dates, and text.

Line

Line items are generally used for drawing vertical or horizontal lines for headings, underlining titles, and so on. Lines are defined by their start and end point coordinates.

Rectangle

Rectangles are usually used for drawing boxes or borders around a specified area. A rectangle makes drawing boxes easier than drawing four separate lines.

Ellipse

The Ellipse item draws a circle or an ellipse in a report, for example as a decoration or a border around text.

Image

Image items are used to insert either static or dynamic images into a report. Static images, such as a company logo often displayed in the report header, can be loaded from a file or stored in the report definition. Dynamic images can be loaded from a data source, for example from an SQL column or from a file name stored in the data.

Barcode

The Barcode item renders barcodes using the integrated Zint library, which supports more than 50 barcode symbologies (for example EAN-13, Code 128, QR Code, Data Matrix, and PDF417). Barcodes can be either static or dynamic items, similar to images. Static barcodes read their value from the report definition; dynamic barcodes are loaded from the specified data source.

Graph (custom item)

The Graph/Custom item is a special member of the NCReport items. It enables you to render QPainter-drawn content in reports using your own C++/Qt code. The typical use of this feature is rendering graphs or similar content.

Cross tab

The cross table (pivot table) item. See the chapter about cross tables for details.

Table View

Renders a QTableView widget or a QAbstractItemModel as a table.

Graphics View

Renders the content of a QGraphicsView widget provided by the application at runtime.

Chart

Renders HighCharts charts. This item is available only if the optional HighCharts extension is enabled.

8.7.1. Adding heading Labels

First, let's add the labels that represent the column headers of the data rows. To create a new Label object, select the **[Label]** tool button or the **Insert › Label** menu item. The cursor then changes to a crosshair. Click in the section where you want the label to be located. (In our example, we add the labels to the page header.) Doing so creates the Label object in that section and opens the label settings dialog, in which you can set the properties of the label.

The following options are available for labels:

Text

The text of the label.

Word wrapping

If this check box is selected, the text is wrapped to fit the width of the item.

Print when expression

A logical expression that defines when the label is printed. See the details later.

8.7.2. Label dialog

The following buttons are available for applying or canceling settings:

- **[OK]** Applies your label settings.
- **[Cancel]** Closes the dialog without saving any changes and returns you to the Designer main window.

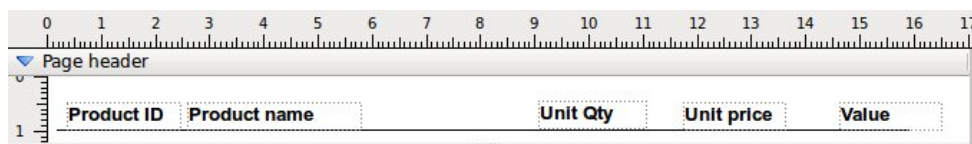
Add the following labels to the page header: Product ID, Product name, Unit Qty, Unit price, and Value, and move them to the location you want by drag and drop. To move a label, drag it with the left mouse button and drop it at the new location. To delete a label, select it and press the **Delete** key.



Figure 6. Adding labels

8.7.3. Adding a Line

To create a new Line object, select the **[Line]** button in the toolbar or the **Insert > Line** menu item. The cursor then changes to a crosshair. Click in the section where you want the line to start and drag the line to the end position. To move a line, drag it with the left mouse button and drop it at the new location. To delete a line, select it and press the **Delete** key.



The style, width, and color of the selected line can be set using the **Shape properties** toolbar. To open the line properties dialog, double-click the line. The dialog provides the following option:

Print when expression

A logical expression that defines when the line is printed. See the details later.

8.7.4. Adding Fields

Now we add the most important items to the report. Field objects contain dynamic information retrieved from a data source, a parameter, or a variable. To create a new Field object, select the **[Field]** tool button or the **Insert > Field** menu item. The cursor then changes to a crosshair. Click in the section where you want the field to be located, which is generally the Detail section. Doing so creates the Field object in that section at the clicked position and opens the field settings dialog, in which you can set the properties of the field.

The following options are available for fields:

Field source type

The combo box contains the possible sources from which the field can pull data: Data source, Parameter, Variable, System variable, Script expression, and Template. You can find information about the various source types in the NCReport specification.

Field column/expression

The data column or expression from which the field's value is taken. When the field uses an SQL query data source, this is the name of the corresponding SQL column in the query. For other data sources, such as Text, this value is usually the number of the data column. The **[Build expression...]** button opens the expression builder.

Data type

The field's base data type. The following data types are supported: Text, Numeric, Date, DateTime, Time, and Boolean.

Word wrapping

If this check box is selected, the field text is wrapped to fit the width of the item.

Template arg() string

A string expression containing a `%1` placeholder, which serves the same purpose as the `QString("String %1").arg(value)` code. The field's value is embedded into this expression.

Print when expression

A logical expression that defines when the field is printed. See the details later.

8.7.5. Field dialog

Field source type: Data source Data type: Text

Field

col0

Build expression...

General Text Numeric Date/Time

Description / title:

Template arg() string:

Background mode: Transparent Background color: User color 1

Elide mode: Off

Options

☐ Word wrapping ☐ Background only in Designer ☐ Pin to left

☐ Word wrap anywhere ☐ Clickable in Preview ☐ Pin to right

☐ Automatic height ☐ Hide repeated values ☐ Use appearance roles

Identify Print when HTML Dynamic position Dynamic style TOC

Item ID: 1LDDE Zone ID: 0

Function ID:

☐ Page break after printing this item

Conditional format... OK Cancel

The following table summarizes the various formulas you can specify in fields as the field column expression. The formula depends on which field source type you use.

8.7.6. Field column formulas

Field source type	Field column formula	Description
Data source	<code>[dataSourceID.]column</code>	The column is a valid column name of the data source, for example an SQL column name in your SQL query. If dataSourceID is specified, the report engine uses the data source with this ID. If you do not specify dataSourceID, the default (currently processed) data source is used, which is the one assigned to the detail. Data source references can also contain functions. Read more in the Expressions chapter.
Parameter	<code>parameterName</code>	The name/ID of the parameter
Variable	<code>variableName</code>	The name/ID of the variable
System variable	<code>variableName</code>	The name/ID of the system variable

Script expression	<code>expression</code>	You can use even a complex script expression for the field. Data source values, parameters, and variables can all be used in expressions. For more information, see the Expressions chapter.
Template	<code>template expression</code>	A template is a simple text substitution of data source values, parameters, or variables, all joined into one string.

Some properties are available only for certain data types. They are located on separate tab pages of the dialog (**Text**, **Numeric**, and **Date/Time**). The following additional options are available for numeric fields:

Number formatting

If this option is selected, number formatting is turned on.

Use localized settings

If this option is selected, the report engine uses localized number formats based on the application's current QLocale settings.

Blank if value equals zero

If this option is selected, the field's value does not appear when it equals zero.

Substitution string if value equals zero

If this option is selected, the text specified in **Substitution string** is printed instead of a zero value.

Decimal precision

The number of digits after the decimal point.

Field width

The width of the number in characters. Specifies the minimum amount of space that the value is padded to. A positive value produces right-aligned text, whereas a negative value produces left-aligned text. `Auto` means no padding.

Format character

A single character that specifies the number format code. Possible values are **e**, **E**, **f**, **g**, and **G**. With **e**, **E**, and **f**, the precision is the number of digits after the decimal point. With **g** and **G**, the precision is the maximum number of significant digits. The value is used by the `QString::arg(double a, int fieldWidth = 0, char format = 'g', int precision = -1, QChar fillChar)` function.

8.7.7. Field dialog - numeric data

The following buttons are available for applying or canceling settings:

- **[OK]** Applies your field settings.
- **[Cancel]** Closes the dialog without saving any changes and returns you to the Designer main window.

To continue our example report, add the following fields to the detail section, using these column names and data types: ProductID (Numeric), ProductName (Text), QuantityPerUnit (Numeric), UnitPrice (Numeric), and Value (Numeric).

Detail [Detail1]				
ProductID	ProductName	QuantityPerUnit	UnitPrice	Value

Figure 7. Details section with fields

8.7.8. Adding Variables for totals

Before we add a variable field to the report, let's look at how variables are handled in NCReport. Variables are special items used for providing counts, totals, and other aggregate values. Each variable has a name, a function type, and a data type, and is based on an expression, usually a data source column. To add a variable, select **Report > Variables...** A dialog then appears in which you can manage variables. The variables are listed in a table with the columns **ID**, **Expression**, **Function**, **Scope**, and **Init Value**, which can be edited directly in the table.

The following options are available for variables:

Variable ID

The name/ID of the variable.

Variable expression

The data column (or expression) from which the variable's value is taken.

Function type

The aggregate function of the variable: **COUNT** increments by 1 for every detail row; **SUM** sums the values of the specified data column; **AVERAGE**, **STD**, **MIN**, and **MAX** calculate the average, the standard deviation, the minimum, and the maximum of the values.

Reset scope

When the variable is reset: **Report**, **Page**, or **Group**.

Initial value

The initial value of the variable.

8.7.9. Variable dialog

The screenshot shows a dialog box titled 'Variable dialog'. It contains a table with the following columns: ID, Expression, Function, Scope, and Init Value. The first row is selected and contains the values: 1, var0, col4, SUM, Group, and 0. To the right of the table are two buttons: a green plus sign (+) for adding a new variable and a red minus sign (-) for removing a variable. At the bottom right of the dialog are 'OK' and 'Cancel' buttons.

ID	Expression	Function	Scope	Init Value	
1	var0	col4	SUM	Group	0

The variables added to the report are shown in the variable table. Clicking a row makes that variable active. The following buttons are available in the dialog:

- **[Add]** Adds a new variable and enables the variable options for editing.
- **[Remove]** Deletes the selected variable.
- **[OK]** Saves your variable settings.
- **[Cancel]** Closes the dialog without saving any changes and returns you to the Designer main window.

Add a new variable by clicking the **[Add]** button, and then specify the options as follows: Variable ID: `total_value`, Variable expression: `value`, Function type: **SUM**, Reset scope: **Group**. To print the total, we first have to add a new group to the detail. In the next section, we explain how to use the grouping feature.

8.7.10. Adding a group to a detail

While many reports can be defined using a single Detail section with multiple columns and rows of data, others, like our example report, require summary data such as totals and subtotals. For such reports, NCReport supports group sections. Group sections have the following characteristics:

- They are always associated with a Detail section.
- They are defined by group headers and group footers.
- Group headers always print above their detail rows.
- Group footers always print below their detail rows.
- They reference a data column (or expression) on which the group headers and footers break.
- A new group footer and group header are printed each time the value of the referenced column changes.
- An unlimited number of group levels is allowed.

In the group dialog, the groups of the detail are shown in the order you added them. The added group sections appear in the Designer after you apply the group settings. Groups are structured hierarchically: the first group is the primary (outermost) group level, the second one is the second level, and so on.

To add a new group to the detail, select **Report > Details and grouping....** The Detail settings dialog then appears. Select the **Detail1** detail in the list, then click the **[Groups...]** button to open the **Data grouping** dialog, which always belongs to the selected detail. To add a new group, click the **[Add]** button.

The following options are available for a group:

Group ID

The name/ID of the group, used for identification purposes.

Group expression

The data source column (or expression) the group is based on. When the value of this column changes, the group breaks. The constant expression **%CONST** can also be used as the group expression (a group expression of **0** is converted to **%CONST** when the report is loaded). In that case the group never breaks; it just ends at the end of the data. This is useful for grand total fields.

Header and Footer

To enable or disable the group header and footer, select or clear the corresponding check box. To set the initial height of these sections, use the spin boxes next to the check boxes. The **Reprint header on new page** option repeats the group header at the top of each new page.

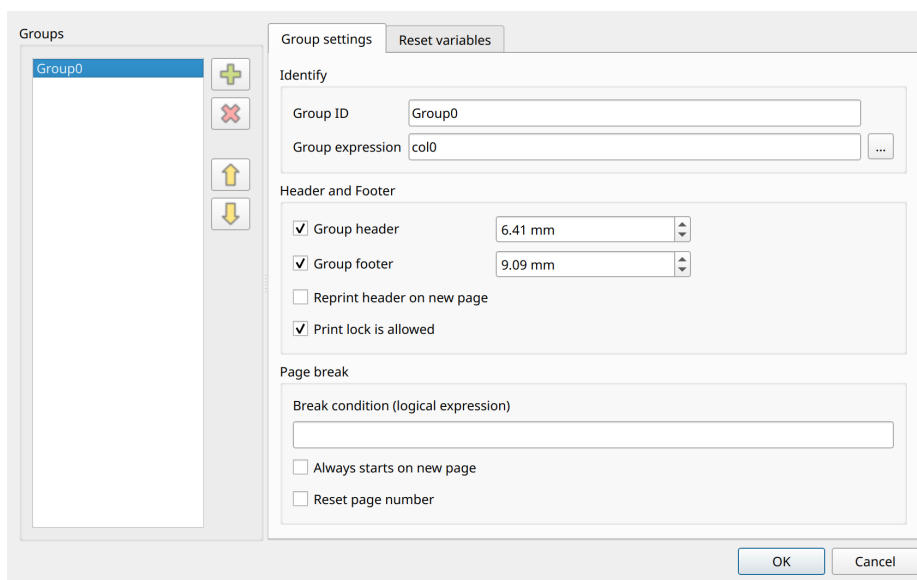
Page break

A break condition (logical expression), and the **Always starts on new page** and **Reset page number** options.

Reset variables

This list contains the variables that can be reset when the group ends. Only the variables with **Group** reset scope are listed.

8.7.11. Group dialog



The groups added to a detail appear in the group list. Clicking a list item makes that item active. To delete the selected group, click the **[Remove]** button. The following buttons are available in the dialog:

- **[Add]** Adds a group and enables the group options for editing.
- **[Remove]** Removes the group selected in the list.
- **[OK]** Saves your group settings.
- **[Cancel]** Closes the dialog without saving any changes and returns you to the Detail settings dialog.

Now let's add a new group with the following settings: Group ID: **Group0**, Group expression: **%CONST**, group header and footer enabled, and the **total_value** variable reset. After you click the **[OK]** button, the new group sections (header and footer) appear in the report. Close the Detail settings dialog by clicking the **[OK]** button.

8.8. Adding a total variable field

Now we have a group with a header and a footer. Group footer sections are generally used for showing totals and subtotals. Let's add a new field to the group footer with the following properties: Field source type: **Variable**, Field column: **total_value**, Data type: **Numeric**.

Now we have almost all of the fields we need. All that is left to do is add some missing lines and labels and adjust the report.

8.9. Other items

The remaining tasks are summarized below:

- Add a "Total value:" label to the group footer section, near the total field.
- Add a line above the totals.
- Move the items to align them with the appropriate columns.
- Add a line to the page footer, similar to the line in the page header.
- Add a field to the page footer: Field source type: **System variable**, Field column: **pageno**, Data type: **Numeric**, QString::arg() expression: **Page: %1**.

8.10. Adjustment and formatting

To finish the report, we can now format and adjust the items. Perform the following tasks:

- Adjust the height of the sections to a suitable size by dragging the bottom edge of the section with the mouse or by using the geometry editor. The height of the detail is important, since it is repeated for every record.
- Select the labels in the page header and set the font weight to bold by clicking the **[Bold]** tool button in the toolbar. Multiple items can be selected at once.
- Select the ProductID field in the Detail section and set its font weight to bold.
- Right-align the text of all of the numeric fields by clicking the **[Right]** text alignment tool button in the toolbar.
- Set the number format options for the numeric fields: Number formatting: on, Decimal precision: 2.
- Also turn on **Use localized settings** for the value and total_value fields.

Save the report. Now you should have something similar to this:

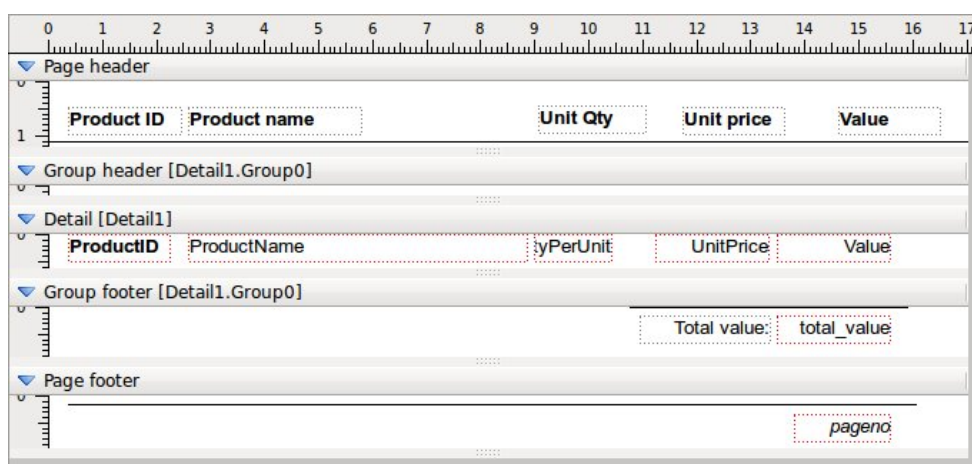
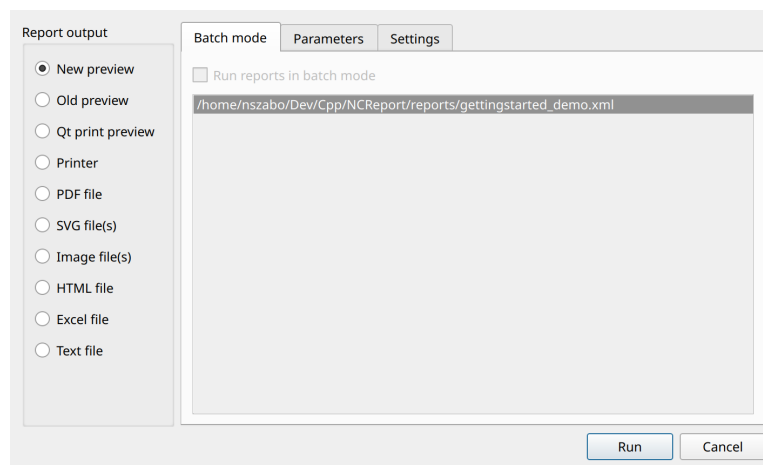


Figure 8. Report is ready

8.11. Running the report

To run the report from the Designer, select **File > Run report...** (**Ctrl** + **R**). The **Run report** dialog then appears. Select the output of the report: New preview, Old preview, Qt print preview, Printer, PDF file, SVG file(s), Image file(s), HTML file, Excel file, or Text file. You can add and remove test parameters with the **[Add]** and **[Remove]** buttons on the **Parameters** page; for more about parameters, see the Parameters chapter. The dialog also lets you set a page range and run reports in batch mode. Start the report by clicking the **[OK]** button.

To run the report directly to a given output without the dialog, use the **File > Run report to** submenu, for example **File > Run report to > Preview** (**Ctrl** + **P**).

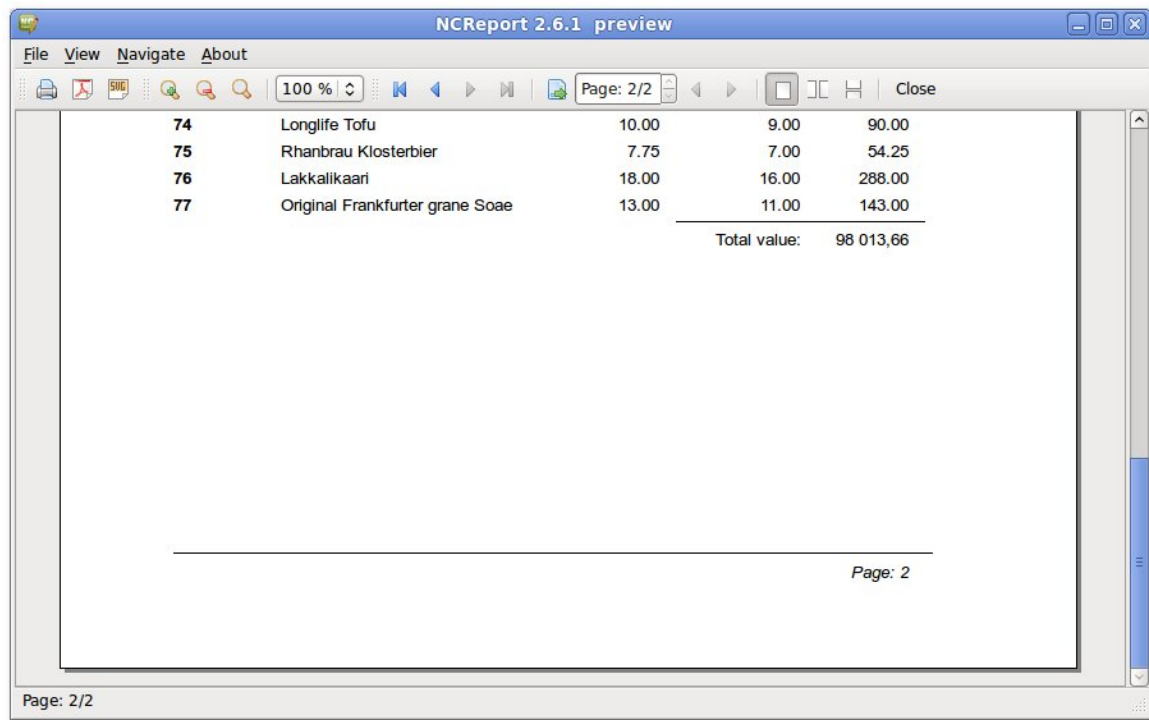


When you run the report to the preview window, you should see something similar to this:

Product ID	Product name	Unit Qty	Unit price	Value
3	Aniseed Syrup	10.00	9.00	90.00
4	Chef Anton's Cajun Seasoning	22.00	20.00	440.00
5	Chef Anton's Gumbo Mixj	21.35	19.00	405.65
6	Grandma's Boysenberry Spread	25.00	21.00	525.00
7	Uncle Bob's Organic Dried Pears	30.00	25.00	750.00
8	Northwoods Cranberry Sauce	40.00	34.00	1360.00
9	Mishi Kobe Niku	97.00	72.00	6984.00
10	Ikura	31.00	26.00	806.00
11	Queso Cabrales	21.00	19.00	399.00
12	Queso Manchego La Pastora	38.00	32.00	1216.00
13	Konbu	6.00	5.00	30.00
14	Tofu	23.00	21.00	483.00
15	Genen Shouyu	15.50	14.00	217.00
16	Pavlova	17.45	15.00	261.75
17	Alice Mutton	39.00	33.00	1287.00
18	Camaronv Tigers	62.50	46.00	2875.00

Figure 9. Preview output - page 1

The second page:



NCReport 2.6.1 preview

File View Navigate About

100 % Page: 2/2 Close

74	Longlife Tofu	10.00	9.00	90.00
75	Rhanbrau Klosterbier	7.75	7.00	54.25
76	Lakkalikaari	18.00	16.00	288.00
77	Original Frankfurter grane Soae	13.00	11.00	143.00
Total value:				98 013,66

Page: 2

Page: 2/2

Figure 10. Preview output - page 2

Chapter 9. Designer Reference

The **Getting Started with NCReport Designer** chapter lists the menus, toolbars and keyboard shortcuts of the Designer. This chapter goes further: it describes how the editing tools work in detail, explains the preferences and where they are stored, and covers the less obvious features such as opening and saving reports in a database, the test data editor, the user interface languages and the command line arguments of the Designer.

9.1. Editing Tools

9.1.1. Selecting items

When the **Selection** tool (the arrow in the **Tools** toolbar) is active, you can select report items in the sections:

- Click an item to select it. Clicking an item or an empty area of a section clears the selection in all other sections.
- Drag a rectangle over an empty area of a section to select every item inside the rectangle (rubber band selection).
- Hold down **Ctrl** and click items to add them to, or remove them from, the selection. With **Ctrl** held down, the selection in the other sections is kept, so you can select items in several sections at the same time.

The Designer remembers the order in which the items were selected. The first selected item is the *reference item* of the alignment and sizing operations (see [Section 9.1.8](#)), and it is the item shown in the **Geometry editor**.

After you place a new item with one of the insert tools, the Designer automatically switches back to the **Selection** tool.

9.1.2. Mouse and keyboard actions

The following actions are available on the design area:

Action	Result
Double-click an item	Opens the property dialog of the item.
Double-click an empty area of a section	Opens the section properties dialog.
Enter or Return	Opens the property dialog of the selected item, if exactly one item is selected.
Del	Deletes the selected items.
Arrow keys	Move the selected items. If Snap to grid is on, an item moves by one grid step, otherwise by one pixel.
Ctrl + arrow keys	Move the selected items by five steps (five grid steps or five pixels).

Action	Result
Shift + arrow keys	Resize the selected items: Up and Down change the height, Left and Right change the width.
Ctrl + Shift + arrow keys	Resize the selected items by five steps.
Shift or Ctrl while drawing a line	Constrains the new line to horizontal or vertical.
Drag an end point of a selected line	Moves that end point. With Ctrl held down, the whole line is moved instead.
Drag items from the Data Source Manager	Creates field items in the section. When several entries are dropped at once, the new fields are placed below each other; with Alt held down, they are placed side by side.

The mouse wheel does not zoom the design area; use the zoom actions instead (see [Section 9.3](#)).

9.1.3. Context menus

Right-clicking an item opens its context menu with the following entries: **Item properties...**, **Copy**, **Cut**, **Delete item**, **Lock item** (or **Unlock item** for a locked item), **Bring to front**, **Move to back**, **Section properties...**, and depending on the section type **Detail and groups...** (detail sections) or **Group settings...** (group headers and footers).

Right-clicking an empty area of a section opens the section context menu: **Section properties...**, **Detail and groups...** or **Group settings...**, **Paste**, and the quick insert entries **Insert Field**, **Insert Label**, **Insert Text** and **Insert Barcode**. The quick insert entries create the item at the clicked position (snapped to the grid) and immediately open its property dialog.

9.1.4. Copy, cut and paste

Edit > **Copy** (**Ctrl** + **C**) copies the selected items of the current section to an internal clipboard of the Designer as an XML fragment. **Edit** > **Cut** (**Ctrl** + **X**) does the same and then deletes the items. **Edit** > **Paste** (**Ctrl** + **V**) inserts the copied items into the current section:

- The internal clipboard belongs to the Designer main window, so items can be copied from one report window and pasted into another one. It is not the system clipboard; to copy the whole report definition to the system clipboard, use **File** > **Copy source to clipboard** (see [Section 9.4](#)).
- The pasted items keep their original position, shifted by one grid step to the right and down, so they do not cover the originals exactly.
- The pasted items get new item IDs, they are placed above the existing items of the section, and they become selected.
- Copy and cut work on the selected items of the current (focused) section only.

9.1.5. Undo and redo

Edit › Undo (Ctrl + Z) and **Edit › Redo** (Ctrl + Y) step backward and forward in the editing history. Every report window has its own history; the Undo and Redo actions always work on the active window.

The following operations are recorded in the history: inserting items (also by paste and drag and drop), deleting items, moving items, resizing items, moving the end points of lines, and the alignment and same width/height operations of the **Align** menu. Changes made in property dialogs, locking and unlocking, setting zone IDs and the z-order operations are not recorded.

9.1.6. Locking items

Edit › Lock items (Ctrl + L) locks the selected items in all sections, **Edit › Unlock items** (Ctrl + Shift + L) unlocks them. A single item can also be locked or unlocked from its context menu. A locked item is still selectable, and its selection handles are drawn in red instead of green. A locked item cannot be moved (neither with the mouse nor with the arrow keys), cannot be resized with the mouse, and is not deleted by **Delete**, **Cut** or the Del key. Its properties can still be edited in the property dialog.



The lock state is a design-time protection only. It is not saved in the report definition, so all items are unlocked again when the report is reopened.

9.1.7. Z-order

Items in a section can overlap. The **zValue** attribute of each item determines the stacking order: items with a higher value are drawn above items with a lower value.

- **Align › Bring to Front** gives every selected item a z-value that is higher than that of any other item in the section.
- **Align › Move to Back** gives every selected item a negative z-value, which puts it below the other items.

Both actions are also available in the item context menu (**Bring to front**, **Move to back**). The actions skip line items. The program also contains **Move forwards** and **Move backwards** actions, but they have no effect in the current version and they are not added to any menu or toolbar.

9.1.8. Aligning and sizing items

The **Align** menu and toolbar actions align or resize the selected items relative to the reference item, which is the first selected item. The reference item itself does not move. Line items are ignored by these operations, both as reference and as aligned item.

Action	Result
Align left	The left edges of the items are aligned to the left edge of the reference item.
Align right	The right edges are aligned to the right edge of the reference item.

Action	Result
Align top	The top edges are aligned to the top edge of the reference item.
Align bottom	The bottom edges are aligned to the bottom edge of the reference item.
Center vertically	The items are moved horizontally so that their horizontal centers are aligned with the center of the reference item (they are centered on a common vertical axis).
Center horizontally	The items are moved vertically so that their vertical centers are aligned with the center of the reference item (they are centered on a common horizontal axis).
Set widths the same	Every item gets the width of the reference item; the heights are not changed.
Set heights the same	Every item gets the height of the reference item; the widths are not changed.

The operations apply to the selected items of all sections and can be undone in one step.

9.1.9. Setting the zone ID

Edit › Set zone ID... opens a small dialog in which you can enter a zone ID. When you click **[OK]**, the zone ID is assigned to all selected items of the current section. The action does nothing if no item is selected in the current section. Zones are described in the **Zones** chapter.

9.2. Grid and Snap to Grid

The design area can display a grid of dots, and the items can be aligned to this grid automatically:

- **View › Show grid** shows or hides the grid dots.
- **View › Snap to grid** turns snapping on or off. When snapping is on, the position of the items is rounded to the nearest grid point when they are inserted or moved, the size is rounded when they are resized with the mouse, and the arrow keys move and resize the items by one grid step.

Both settings apply to all open report windows. The grid size is set in the Preferences dialog (**File › Preferences...**), on the **Report editor** tab, in the **Default grid** group:

Visible

The default state of **Show grid**. The default is off.

Snap to grid

The default state of **Snap to grid**. The default is on.

Grid X, Grid Y

The horizontal and vertical distance between grid points in screen pixels of the design area. The default is 6 for both.

Reset grid

Restores the defaults: grid hidden, snapping on, 6 x 6 grid.



Toggling **Show grid** or **Snap to grid** from the **View** menu affects the current session. These two settings, together with the grid size, are saved only when you close the Preferences dialog with **[OK]**.



Do not set the grid size to 0. The grid drawing and snapping calculations divide the section into grid steps, and a zero step size is not handled.

9.3. Zooming

Zooming magnifies the sections of the active report window; the rulers follow the zoom level.

Action	Shortcut	Result
View > Zoom in	Ctrl + +	Increases the magnification by one step (30%).
View > Zoom out	Ctrl + -	Decreases the magnification by one step. It cannot go below 100%.
View > Zoom one	Ctrl + 0	Resets the magnification to 100%.

The magnification ranges from 100% to 400% in ten steps. Each report window has its own zoom level. Newly created and newly opened reports are displayed at 100%. Zooming does not change the report definition and does not mark the report as modified.

9.4. XML Source of the Report

File > Copy source to clipboard puts the complete XML report definition of the active report on the system clipboard. The XML is generated from the current state of the report in the Designer, including the unsaved changes, so you do not need to save the file first. This is handy for pasting the definition into a text editor, a support request, or your application source code.

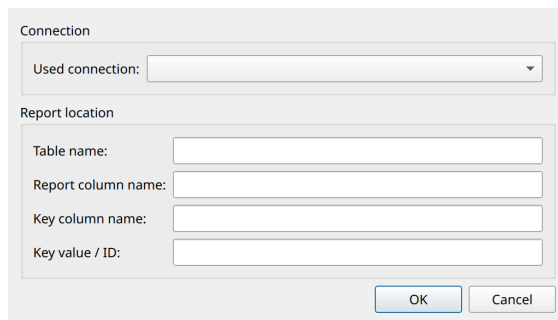
The Designer also contains a **Show source** action (status tip: "Show XML source code"), but in the current version it is not added to any menu and its implementation is empty, so it is not available. To see the source, copy it to the clipboard or open the saved **.ncr** file in a text editor.

9.5. Opening and Saving Reports in a Database

Report definitions can be stored in a column of an SQL database table instead of a file. The Designer can open a report definition from such a table and save it back.

9.5.1. Opening a report from a database

Select **File > Open from database....** The **Open report from database** dialog appears:



Used connection

One of the database connections defined in the **Database connection manager** (see the database connection chapter). The Designer connects to it when you click [**OK**]. If no connection is selected, the default database connection of the application is used.

Table name

The table that stores the report definitions.

Report column name

The column that contains the XML report definition. The column must be able to store text of the size of a report definition.

Key column name

The column that identifies the report.

Key value / ID

The key of the report to open.

From these values the report engine (`NCReportSource`) builds the following query:

```
SELECT <report column> FROM <table> WHERE <key column>=<'key value'>
```

The key value is always written as a quoted string literal. The XML text is read from the first column of the first record of the result. The opened report window is titled `DB:<key value>`. The values you enter are remembered (settings `mDBConnectionID`, `mDBTable`, `mDBColumn`, `mDBKeyColumn` and `mDBKeyValue`) and offered again the next time the dialog is opened.

9.5.2. Saving a report to the database

A report opened from a database is saved back to the same record with **File > Save** (`Ctrl + S`). The engine executes the statement

```
UPDATE <table> SET <report column>=? WHERE <key column>=<'key value'>
```

with the XML report definition bound to the parameter. Because only an `UPDATE` is executed, the record must already exist: the Designer cannot insert a new report record into the table. To store a new report, insert a record with its key first (for example with a placeholder definition), then open it from

the database in the Designer.

A minimal table that fits these requirements could look like this:

```
CREATE TABLE reports (
  id      VARCHAR(32) PRIMARY KEY,  -- key column
  reportdef TEXT                    -- report column (XML)
);
```

The table and column names are free; you enter them in the dialog.



- The loader checks the size of the query result. With database drivers that cannot report the number of rows of a `SELECT` (for example SQLite), the report cannot be opened and the error "No report definition found!" is displayed.
- **File > Save As...** does not convert a database report into a file report: the report is still written into its database record, only the window title changes to the selected file name.
- Reports opened from a database are not added to the recent files list.

9.5.3. Database mode for applications that embed the Designer

When the Designer main window (`NCRDMainWindow`) is used in your own application, the default values of the **Open from database** dialog can be preset with `setDefaultConnectionID()`, `setDefaultDBTable()`, `setDefaultDBColumn()`, `setDefaultDBKeyColumn()` and `setDefaultDBKeyValue()`. The Designer's `main.cpp` contains an example of this (compiled when `NCRD_DATABASE_MODE_DEMO` is defined).

In your application, a report stored in the database is run by setting the database source on the report source object (`NCReportSource::Database` source type with the connection ID, table name, column name, key column name and key value).

9.6. Layout Direction

The Designer base window contains a **Switch layout direction** function. It toggles the layout direction of the whole Designer application between left-to-right and right-to-left (`QApplication::setLayoutDirection()`). It mirrors the user interface only (menus, toolbars, dock windows and dialogs); it does not change the report definition or the way reports are rendered.

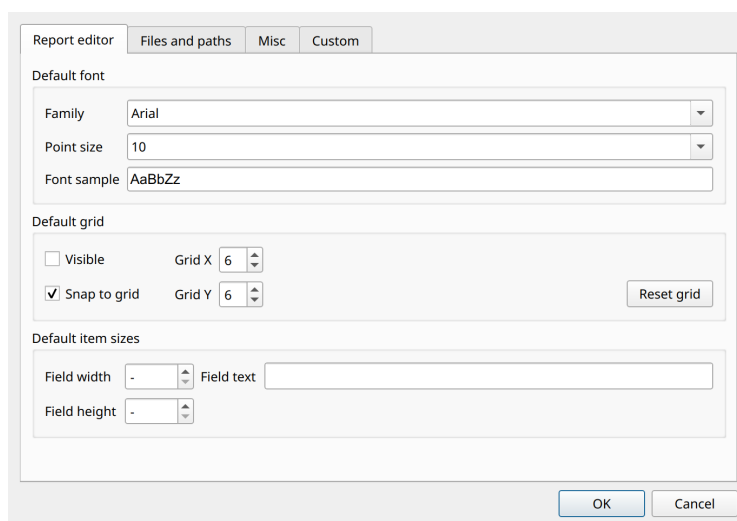
In the current version the **Switch layout direction** menu item is disabled in the source code (commented out of the **View** menu), so the function is not available from the user interface.



The Designer is a standard Qt application, so the standard Qt command line option `-reverse` can be used to start it with a right-to-left user interface.

9.7. The Preferences Dialog

File > Preferences... opens the Preferences dialog. The settings take effect when you click **[OK]**, and they are saved at the same time.



9.7.1. Report editor tab

Default font: Family, Point size

The default font of new reports. When you create a new report, this font becomes its default font (the `fontname` and `fontsize` report options), which can be changed later in the report and page settings. The default is Arial 10. The **Font sample** field shows a preview.

Default grid

Grid visibility, snapping and grid size, see [Section 9.2](#).

Default item sizes: Field width, Field height

The size, in screen pixels, of new field items created with the **Field** tool or the **Insert Field** context menu entry. The value - (0) means the built-in default size. The maximum is 500.

Default item sizes: Field text

The initial text (column reference) of new field items. If empty, new fields contain the text "Field".

9.7.2. Files and paths tab

Report directory

The initial directory of the **Open** and **Save As** file dialogs, used until you open or save a file in another directory during the session.

Output directory

The initial directory of the file dialogs that select the output file when running a report to a file (PDF, SVG, image, HTML, text, Excel), until you choose another output directory during the session.

Template directory

Passed to the report definitions as the template directory. It is used when looking up template files, for example the HighCharts chart templates.

Zint directory

The directory of the Zint barcode program. The value is applied when the Designer starts. The current rendering engine does not use it, because barcodes are rendered by the Zint library linked to NCReport.

Output file counter

The Designer proposes output file names in the form `report_result_NNNN.<extension>`, where `NNNN` is this counter (padded to four digits). The counter is incremented with every proposed file name.

HTML Output

The strategy of the HTML output: **Sections build from one table** or **Sections build from multiple tables**, and an optional **CSS file**. These are passed to the HTML output when a report is run to an HTML file.

9.7.3. Misc tab

Open report windows maximized

New and opened report windows are displayed maximized in the MDI area.

Encode stored database connection passwords

The passwords of the database connections are stored Base64-encoded in the Designer settings instead of plain text.

Supports custom XML attributes

When a report definition is opened, the Designer remembers the XML attributes of items and sections that it does not recognize (for example attributes added by your own tools). If this option is on, these attributes are written back when the report is saved; otherwise they are dropped.

Printer resolution

High resolution (default) or **Screen resolution**. With **Screen resolution** the reports run from the Designer use the screen resolution print quality.

Default printer name, Print without printer dialog

If **Print without printer dialog** is checked, the printer output prints directly to the printer named in **Default printer name** without showing the print dialog.

Printer DPI

The print resolution in DPI used for the reports run from the Designer; it is also used by the new preview. **Auto** (0) keeps the default resolution. The maximum is 1200.



Base64 encoding is not encryption. Anyone who can read the settings can decode the passwords. Protect the settings file or registry key if the stored passwords are sensitive.

9.7.4. Custom tab

Custom settings

Free text in `name=value` lines. The Designer itself does not use these values; they are intended for applications that embed or extend the Designer. Such code can read a value with `NCRDMainWindow::customSetting(name)`.

9.7.5. Where the settings are stored

The Designer stores its preferences and state with `QSettings`, using the organization name `GrassBits`, the organization domain `ncreportsoftware.com` and the application name `NCReportDesigner`. The physical location follows the usual Qt conventions of the platform: the registry key `HKEY_CURRENT_USER\Software\GrassBits\NCReportDesigner` on Windows, a `GrassBits/NCReportDesigner.conf` file in the user's configuration directory (usually `~/.config`) on Linux, and a property list file in `~/Library/Preferences` on macOS. The organization and application names can be changed with command line arguments (see [Section 9.10](#)).

The most important keys are:

Key	Content
<code>reportdir, outputdir, templatedir, zintdir</code>	Directories of the Files and paths tab.
<code>defaultfontfamily, defaultfontpointsize</code>	Default font of new reports.
<code>gridvisible, snaptogrid, gridX, gridY</code>	Grid settings.
<code>defaultFieldWidth, defaultFieldHeight, defaultFieldText</code>	Default item sizes.
<code>filecounter</code>	Output file counter.
<code>htmlOutputStrategy, htmlOutputCssFile</code>	HTML output settings.
<code>showmaximized, bpa, supportsCustomAttribs</code>	Open report windows maximized, Encode stored database connection passwords, Supports custom XML attributes.
<code>printerResolution, printerDPI, defaultPrinterName, noPrinterDialog</code>	Printer settings.
<code>customSettings</code>	Text of the Custom tab.
<code>connections</code>	The database connections of the connection manager.
<code>parameters/<name></code>	Test parameter values entered in the Designer.
<code>recentFileList</code>	The recent files (at most 10).
<code>mDBCConnectionID, mDBTable, mDBCColumn, mDBKeyColumn, mDBKeyValue</code>	Last values of the Open report from database dialog.
<code>geometry, state</code>	Main window geometry and toolbar/dock layout.

9.8. The Test Data Editor

The test data editor (**Report > Test data editor...**, **Ctrl + T**) provides data for the reports run from the Designer, so that reports using string lists, item models or string parameters can be tested without your application. The basic usage is shown in the **Getting Started** chapter; this section describes the editor in detail.

9.8.1. Test data items

A test data file contains any number of test data items. The list on the left side of the editor shows their IDs; the right side shows the settings and the content of the selected item:

ID (name)

The ID under which the data is passed to the report. It must match the ID that the data source of the report refers to (for example the string list ID or the model ID).

Datasource type

String list (QStringList), **Item model (QStandardItemModel)** or **String parameter (QString)**.

Column separator

The column separator of the content. Only **Tab** is available.

Enabled / Add to report

Only enabled items are passed to the report.

The table below the settings holds the content. Each table row is one record (one line of the string list, one row of the item model, or one line of the string parameter); for string lists and item models the table columns are the columns of the data.

The editor menus and toolbars provide the following operations: **New**, **Open** (**Ctrl + O**), **Save** (**Ctrl + S**), **Save as...** and **Close** for the file, **Add new data table** and **Remove data table** for the test data items, **Add row** (**Ctrl + R**), **Insert row** (**Ctrl + Shift + R**), **Remove row** (**Ctrl + Del**), **Add column** (**Ctrl + C**), **Insert column** and **Remove column** (**Ctrl + Shift + Del**) for the content table, and **Load content from file...**, which appends the lines of a tab-separated text file (*.txt) to the table.

9.8.2. How the test data is passed to the reports

Every time a report is run from the Designer, the Designer reads the file `defaulttestdata.xml` and adds every enabled item to the report:

- A string list item is added with `addStringList()`; every non-empty line becomes a list element.
- An item model item is converted to a `QStandardItemModel` (the tab-separated values become the columns) and added with `addItemModel()`.
- A string parameter item is added as a report parameter with the given ID; its value is the content text.

The file name is relative, so `defaulttestdata.xml` is looked up in the current working directory of the Designer process. The editor opens this file automatically when it starts. Test data saved under any other name is not used when running reports, so keep the data you want to test with in `defaulttestdata.xml` (you can open other test data files in the editor and save their content into it).



In the current version, **Save as...** only selects the new file name; it does not write the file. Choose **Save** afterwards to write the data.

The test data file is a simple XML document. The root element `NCReportTestData` (version `1.0`) contains one `data` element per item, with the `id`, `type` (`list`, `model` or `spar`) and `enabled` attributes; the content is the element text with tab-separated columns:

```
<?xml version="1.0" encoding="UTF-8"?>
<NCReportTestData version="1.0">
  <data id="prod" type="list" enabled="true">A    Magnetometer    D54/78    yes 0.778    15.6
B    Oil pump    CT-784    yes 1.510    17
</data>
</NCReportTestData>
```

The `testdata` directory of the NCReport distribution contains a sample `defaulttestdata.xml`, and the `reports` directory contains demo reports that use test data (`testdata_stringlist_demo.xml`, `testdata_itemmodel_demo.xml`, `testdata_stringparameter_demo.xml` and `testdata_stringparameter_group_demo.xml`).

9.9. User Interface Languages

The user interface texts of the Designer are translatable. The translations are in the `i18n` directory of the NCReport source: the `ncreport_<language>.ts` source files and the compiled `ncreport_<language>.qm` files. The following `.qm` files are available:

File	Language
<code>ncreport_cs.qm</code>	Czech
<code>ncreport_de.qm</code>	German
<code>ncreport_en.qm</code>	English
<code>ncreport_es.qm</code>	Spanish
<code>ncreport_fr.qm</code>	French
<code>ncreport_hu.qm</code>	Hungarian
<code>ncreport_pl.qm</code>	Polish
<code>ncreport_ru.qm</code>	Russian
<code>ncreport_zh_CN.qm</code>	Chinese (Simplified)

The `TRANSLATIONS` variable of the project file `src/translations.pri` lists the German, Hungarian, Russian, Chinese, French, Spanish, Czech and Polish `.ts` files; these are the files updated by `lupdate` and compiled by `lrelease`. Some translations are incomplete; the untranslated texts are displayed in English.

The Designer selects its language at startup:

1. The language code is taken from the system locale (`QLocale::system().name()`), for example `de_DE`, unless the `--tr=<code>` command line argument is given (see [Section 9.10](#)).
2. The Designer loads the translation file `ncreport_<code>.qm`. If the exact file does not exist, Qt also tries the shorter variants of the name, so for `de_DE` the file `ncreport_de.qm` is used.
3. The Qt translation of the standard Qt dialogs (`qtbase_<code>.qm`) is loaded as well.

With Qt 5.3 and later, both files are loaded from the translations directory of the Qt installation (`QLibraryInfo::TranslationsPath`). With older Qt versions, `ncreport_<code>.qm` is loaded from the `i18n` directory relative to the working directory. If no translation file is found, the Designer starts in English. There is no language setting in the Preferences dialog.



The user interface language is independent of the language of a multilingual report. The **Report > Report language** submenu switches the current language of the report content, as described in the **Internationalization** chapter.

9.10. Command Line Arguments

The Designer accepts the following command line arguments:

Argument	Meaning
<code><file></code>	The report file to open at startup.
<code>--tr=<code></code>	The language code of the user interface translation, for example <code>--tr=hu</code> or <code>--tr=zh_CN</code> . Overrides the system locale.
<code>--appname=<name></code>	The application name used for the settings (default: <code>NCReportDesigner</code>).
<code>--orgname=<name></code>	The organization name used for the settings (default: <code>GrassBits</code>).
<code>-print</code> or <code>--print</code>	Prints the opened report to the printer right after startup.

With `--appname` and `--orgname` you can keep several independent sets of Designer settings, for example separate connection lists and preferences for different projects:

```
NCReportDesigner invoice.ncr --tr=de --appname=NCReportDesigner-ProjectA
```



The report file must be the first argument. The Designer takes the first command line argument as the file name whenever any argument without a leading `-` is present, so `NCReportDesigner --tr=de invoice.ncr` would try to open a file named `--tr=de`. Use `-print` only together with a report file.

Part III: Advanced Features

More complex, professional reports need additional features and functions. This part describes the advanced functions of the NCReport reporting system: data sources, expressions, variables, formatting, special items, page control, internationalization, and the available outputs.

Chapter 10. Data Sources

A data source supplies the rows (records) and columns that the report engine iterates over while it renders detail sections, and from which fields, expressions and templates take their values. Every report has at least one data source, and a single report can contain any number of them. Each data source is identified by its ID, which is used to assign it to a detail section and to refer to its columns in expressions.

This chapter is a reference of the data source types that the NCReport engine supports, the options they understand, and the way they are stored in the report definition file.

10.1. Data source types

The engine knows the following data source types (`NCReportDataSource::DataSourceType`). The value in the second column is the `type` attribute of the `<datasource>` element in the report definition.

Type	type attribute	Engine class	Data comes from
SQL query	<code>sql</code>	<code>NCReportSQLDataSource</code>	An SQL <code>SELECT</code> query executed through Qt's SQL module.
Text	<code>txt</code>	<code>NCReportTextDataSource</code>	Delimited text (TSV, CSV and similar): static text, a file, an HTTP URL or a parameter.
XML	<code>xml</code>	<code>NCReportXMLDataSource</code>	An XML document in the NCReport data format: static text, a file or an HTTP URL.
String list	<code>list</code>	<code>NCReportStringListDataSource</code>	A <code>QStringList</code> added to the report from the application.
Item model	<code>model</code>	<code>NCReportItemModelDataSource</code>	A <code>QAbstractItemModel</code> added to the report from the application.
Custom	<code>custom</code>	Subclass of <code>NCReportDataSource</code>	Any data provided by your own data source class.



The engine also contains a class named `NCReportTXTDataSource`. It is an obsolete class that simply inherits `NCReportTextDataSource` without adding anything. The `txt` type in a report definition always creates an `NCReportTextDataSource`; use that class in new code.

10.2. Common data source properties

The following properties are shared by every data source type. They are stored as attributes of the `<datasource>` element.

id

The unique identifier of the data source. Detail sections refer to the data source by this ID, and so do the `#{datasourceID.column}` references.

type

The data source type: `sql`, `txt`, `xml`, `list`, `model` or `custom`. An unknown value stops loading the report with the error *Invalid datasource type*.

source

The location type, see [Section 10.3](#). Values: `static` (default), `file`, `http`, `ftp`, `parameter`.

openRole

The opening role, which controls when the engine opens (or re-opens) the data source, see [Section 10.4](#). Values: `begin` (default), `child`, `group`, `sub`.

parentID

The ID of the parent data source for child data sources, or the ID of the group for data sources opened on group change.

allowEmpty

If `false`, a data source opened at the beginning of the report that contains no rows stops the report with the error *Data not found in datasource*. The older attribute `emptyIsForbidden` (with the inverse meaning) is still accepted when reading.

enabled

Disabled data sources are not opened. If the attribute is missing, the data source is enabled.

primaryKeyColumn

The index of the primary key column, used by data source relations (parent/child data sources).

briefDesc

An optional short description.

headers

An optional comma-separated list of column names. It is used by the string list data source to give names to the columns.

The text content of the `<datasource>` element depends on the location type: it holds the static data (or the SQL query) for the Static type, the file name or URL for the File, Http and Ftp types, and the parameter ID for the Parameter type.

10.3. Location types

The location type (`NCReportDataSource::LocationType`) tells the engine where the data or the query specification can be found.

Static

The data (the SQL query, the delimited text, the XML document) is stored directly in the report definition.

File

The data is read from an external file. The element text is the file name.

Http

The data is downloaded from a URL with the engine's built-in HTTP client (`NCReportHttpClient`). The element text is the URL. Used by the Text and XML data sources.

Ftp

Listed in the enumeration, but the built-in Text and XML data sources do not implement it: opening such a data source fails.

Parameter

The data is passed in a report parameter. The element text is the parameter ID, and the data itself is added from the application with `NCReport::addParameter()`. Used by the Text data source.

Not every type uses every location. The SQL data source executes the query stored in the report definition (Static). The string list and item model data sources always receive their data from the application, so for them only Static is meaningful.

File names are resolved by the engine before use: besides absolute and relative paths, the placeholders `{TEMP}` (or `%TEMP%`), `{HOME}` (or `%HOME%`), `{CURR}`, `{ROOT}`, `{NCREPORT_DIR}` and `{VAR0}...{VAR9}` are replaced, and a relative file name that does not exist is looked up relative to the directory of the report file. The file name of Text and XML data sources may also contain parameter references such as `$P{datafile}`, which are evaluated when the data source is opened.

10.4. Opening roles

The opening role (`NCReportDataSource::OpenRoles`) determines when the data source is opened.

Beginning of the report (begin)

The data source is opened when the report starts. This is the default.

Child data source (child)

The data source is a child of the data source given in `parentID`, and it is updated according to the current record of the parent. See the *Data Relation system* chapter.

Group change (group)

The data source is re-evaluated and updated when the group given in `parentID` changes.

Sub-report iteration (sub)

The data source drives the sub-report iteration, see [Chapter 28](#).

10.5. Referring to data source columns

In fields, expressions and templates, a data source column is referred to as `$D{datasourceID.column}`. If the data source ID is omitted (`$D{column}`), the data source assigned to the current detail section is

used. See the *Using references in expressions* section of the *Data/Script Expressions* chapter for the complete syntax, and the data source functions (row count, current row and similar) that are also available in expressions.

How the `column` part is matched depends on the data source type:

Data source	Column reference
SQL	The column (field) name of the query result.
Text	If the first row is a column header, the header names. Otherwise <code>col0</code> , <code>col1</code> , ... or simply <code>0</code> , <code>1</code> , ... (zero-based).
XML	The names defined in the <code><columns></code> element, or <code>col0</code> , <code>col1</code> , ...
String list	<code>col0</code> , <code>col1</code> , ... or <code>0</code> , <code>1</code> , ...; if the <code>headers</code> attribute is set, the names given there.
Item model	The horizontal header text of the model (<code>Qt::EditRole</code>); otherwise <code>col0</code> , <code>col1</code> , ... or <code>0</code> , <code>1</code> , ...
Custom	Whatever the class implements in <code>value()</code> and <code>columnIndexByName()</code> .

For item models (and for any data source that honors the role argument of `value()`), a role suffix can be appended to the column name to read another Qt data role or the header data:

Suffix	Returned data
<code>:e</code>	<code>Qt::EditRole</code>
<code>:d</code>	<code>Qt::DisplayRole</code>
<code>:u</code>	<code>Qt::UserRole</code>
<code>:c</code>	<code>Qt::DecorationRole</code>
<code>:h</code>	Horizontal header data (<code>Qt::EditRole</code>)
<code>:hd</code>	Horizontal header data (<code>Qt::DisplayRole</code>)
<code>:hu</code>	Horizontal header data (<code>Qt::UserRole</code>)
<code>:hc</code>	Horizontal header data (<code>Qt::DecorationRole</code>)
<code>:ht</code>	Horizontal header data (<code>Qt::ToolTipRole</code>)

For example, `$D{model_ds.price:d}` returns the display text of the `price` column of the current row.

10.6. SQL data source

The SQL data source executes an SQL query through Qt's SQL module (`QSqlQuery`) and iterates over the result. The query is stored in the report definition. It may contain parameter references (`$P{...}`) and data source references (`$D{...}`); both are evaluated before the query is executed, which is how parameterized and child (sub) queries are built.

The data source uses either a database connection created by the application (global connection) or its own private connection, built from the parameters stored in the report.

connection

internal (or **global**): the application's connection is used. **external** (or **private**): the data source opens its own connection with the parameters below.

connID

The connection name, as passed to `QSqlDatabase::addDatabase()` in the application. An empty value or **default** means Qt's default connection. The connection ID may contain parameter references. If the application calls `NCReport::setConnectionName()`, that connection name overrides the one stored in the report.

driver, host, database, user, password, port

The private connection parameters (Qt SQL driver name such as `QMYSQL`, `QPSQL`, `QSQLITE`, `QIBASE`, `QODBC`, `QOCI`, and so on). If the database name contains `{FILE}`, the marker is removed and the remaining path is converted to an absolute file path (useful for SQLite database files).

connOpt

Connection options passed to `QSqlDatabase::setConnectOptions()`.

fwOnly

If **true**, the query runs in forward-only mode (`QSqlQuery::setForwardOnly()`).

forceUpdateOnStepBackRel

Forces the update of a child data source when stepping backward in a data relation.

SQL data source using the application's connection

```
<datasource id="product" type="sql" source="static" openRole="begin"
  connection="internal" connID="northwind">SELECT ProductID, ProductName,
UnitPrice FROM products WHERE ProductName LIKE '$P{prodFilt}'</datasource>
```

SQL data source with a private connection

```
<datasource id="DS" type="sql" source="static" connection="external" connID="interbase"
  host="" database="/data/employee.fdb" user="sysdba" password="masterkey"
  port="" driver="QIBASE">SELECT * FROM employee ORDER BY job_country</datasource>
```

In C++ the application only has to create the database connection before running the report:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL", "northwind");
db.setHostName("localhost");
db.setDatabaseName("northwind");
db.setUserName("user");
db.setPassword("password");
db.open();

NCReport report;
report.setReportFile("products.ncr");
report.addParameter("prodFilt", "C%");
report.runReportToShowPreview();
```

10.7. Text data source

The text data source (`txt`) reads delimited text. Each line is one record, and the columns are separated by a delimiter character. The text can be stored statically in the report, read from a file, downloaded from an HTTP URL, or passed in a report parameter.

columnDelimiter

The column delimiter: `tab` (default), `comma`, `semicolon`, `space`, `vbar` (`|`) or `custom`.

customDelimChar

The delimiter character when `columnDelimiter="custom"`.

hasColumnHeader

If `true`, the first line contains the column names, and columns can be referred to by these names. Otherwise the columns are named `col0`, `col1`, ...

encoding

The text encoding of a data file, for example `UTF-8` (applied when the engine is built with Qt 5).

Text data source with static, tab-delimited data

```
<datasource id="items" type="txt" source="static" openRole="begin"
    columnDelimiter="tab" encoding="UTF-8">1    Item1
2    Item2
3    Item3</datasource>
```

CSV file with a header row

```
<datasource id="customers" type="txt" source="file" columnDelimiter="comma"
    hasColumnHeader="true">{CURR}/data/customers.csv</datasource>
```

Text data passed in a parameter

```
<datasource id="ds1" type="txt" source="parameter" columnDelimiter="tab">data1</datasource>
```

```
QString data;
data += "1\tChai\t16.0000\n";
data += "2\tChang\t17.0000\n";
report.addParameter("data1", data);
```

10.8. XML data source

The XML data source (`xml`) reads data from an XML document in a simple NCReport data format. The document can be stored statically in the report, read from a file or downloaded from an HTTP URL. The data source has no type-specific attributes.

The document may contain several data sets. The engine reads only the `<datasource>` element whose `id` attribute equals the ID of the report data source. Inside it, `<columns>` defines the column names and

every `<row>` element is one record, with one `<d>` element per column, in column order. An optional `<parameters>` block of `<d id="...">` elements is read into the data source's parameter list (`NCReportDataSource::parameters()`).

XML data document

```
<?xml version="1.0" encoding="UTF-8"?>
<NCReportData>
  <parameters>
    <d type="string" id="TransactionNumber">02236412</d>
  </parameters>
  <datasources>
    <datasource id="xmlds1">
      <columns>
        <column type="string">ProductNumber</column>
        <column type="string">ProductName</column>
        <column type="numeric">NetPrice</column>
      </columns>
      <row>
        <d>CK02214</d>
        <d>Monitor</d>
        <d>199.50</d>
      </row>
    </datasource>
  </datasources>
</NCReportData>
```

XML data source reading the document from a file

```
<datasource id="xmlds1" type="xml" source="file" openRole="begin"
  allowEmpty="true">{CURR}/data/xmltestdata.xml</datasource>
```

Columns are referred to by the names in `<columns>`, for example `$D{xmlds1.ProductName}`.

10.9. String list data source

The string list data source (`list`) iterates over a `QStringList` that the application adds to the report with `NCReport::addStringList()`. Each list item is one record, and the columns are separated by the delimiter character.

listID

The ID of the string list, the same ID that is passed to `addStringList()`.

columnDelimiter, customDelimChar

The column delimiter, with the same values as for the text data source (default `tab`).

String list data source

```
<datasource id="sl0" type="list" source="static" columnDelimiter="vbar" listID="sl0"></datasource>
```

```
QStringList list;
list << "24|Renate Moulding|Desert Hot Springs, CA|1|2008-01-01";
```

```
list << "78|Alfred Muller|Miami Beach, FL|1|2008-01-03";

NCReport report;
report.addStringList(list, "s10");
```

Columns are referred to as `col0`, `col1`, ... (or `0`, `1`, ...). If the `headers` attribute of the `<datasource>` element contains a comma-separated list of names, the columns can be referred to by these names instead.

10.10. Item model data source

The item model data source (`model`) reads data from any `QAbstractItemModel`. Each model row is one record. The application adds the model to the report with `NCReport::addItemModel()`.

modelID

The ID of the item model, the same ID that is passed to `addItemModel()`.

useFormatRoles

Enables the use of the model's appearance (formatting) roles. In the Designer: *Use model appearance roles*.

The text content of the element (in the Designer: *Relation Data Specification*) is optional. It is evaluated (data source references are replaced) when the data source is updated in a relation, and the result is emitted with the `NCReport::dataSourceUpdateRequest()` signal, so that the application can refill the model for the current parent record.

Item model data source

```
<datasource id="imds" type="model" source="static" modelID="model1"></datasource>
```

```
QStandardItemModel *model = new QStandardItemModel(2, 2);
model->setHorizontalHeaderLabels(QStringList() << "id" << "name");
model->setItem(0, 0, new QStandardItem("1"));
model->setItem(0, 1, new QStandardItem("Chai"));
model->setItem(1, 0, new QStandardItem("2"));
model->setItem(1, 1, new QStandardItem("Chang"));

NCReport report;
report.addItemModel(model, "model1");
```

Columns are matched against the model's horizontal header data, so `#{imds.name}` works for the model above; `col1` or `1` can be used as well. Role suffixes (see [Section 10.5](#)) give access to other data roles and to the header.

10.11. Custom data sources

A custom data source is your own subclass of the abstract `NCReportDataSource` class. It has to implement the record navigation and data access methods: `open()`, `close()`, `first()`, `last()`, `next()`, `previous()`, `hasNext()`, `seek()`, `size()`, `isValid()` and both `value()` overloads. A complete example is

shown in the *Custom data sources* section of the *Using the NCRReport API* chapter.

There are three ways to connect custom data source objects to a report:

Adding the object directly

Create the object, set its ID with `setID()` and add it with `NCRReport::addCustomDataSource()` (or the equivalent `NCRReport::addDataSource()`). When the report definition contains a `<datasource>` element with the same ID, the engine uses the added object instead of creating a new one.

Data source factory

Subclass `NCRReportDataSourceFactory` and implement `createCustomDataSource(const QString& dsID)`. Set the factory with `NCRReport::setCustomDataSourceFactory()`. The engine calls it for each custom data source of the report that has not been added directly.

Custom data source provider

Subclass `NCRReportCustomDSProvider` and implement `createCustomDataSource(const QString& dsID, NCRReportDataSource::DataSourceType dstype, QObject* parent)`. Set it with `NCRReportDef::setCustomDSProvider()` on `NCRReport::reportDef()`. When a provider is set, the engine asks it to create *every* data source of the report, whatever its type, so the provider can, for example, replace the built-in SQL data source with its own implementation.

Custom data source in the report definition

```
<datasource id="cds0" type="custom" openRole="begin" source="static">cds0</datasource>
```

```
TestDataSource *ds = new TestDataSource(); ①
ds->setID("cds0");
// ... fill the data source ...

NCRReport report;
report.addCustomDataSource(ds);
report.setReportFile("customdatasource_demo.xml");
report.runReportToShowPreview();
```

① `TestDataSource` is an `NCRReportDataSource` subclass.

10.12. Data sources in the Designer

Data sources are managed in the **Data source settings** dialog (**Report > Data sources...**). The list on the left contains the data sources of the report; new ones are added with the **[Add]** button, which first asks for the data source type: *SQL query*, *Text*, *XML*, *String list*, *Item model* or *Custom*. The type cannot be changed later.

The upper part of the dialog holds the common properties: *Data source type*, *Data source ID*, *Opening/running role* (*Beginning of the report*, *Child datasource (subquery)*, *When group change*, *Subreport iteration*), *Location type* (*Static*, *File*, *Http*, *Ftp*, *Parameter*), *File name*, *Parent datasource*, *Primary key column index*, *Allow empty data source* and *Enabled*.

The lower part shows a page specific to the selected type:

SQL query

The *SQL Query* tab with the *Connection Name* and the query editor (with **[Insert Parameter]** and **[Test Query]** buttons), the *Query test* tab that shows the result of a test run, and the *Database connection* tab with *Connection options*, *Forward only mode*, *Force update on stepping backward in relation* and *Use private database connection*, which enables the *Private connection parameters* (driver, host, database, username, password, port, with a **[Test]** button).

Text

Column delimiter, *Custom delimiter character*, *Encoding*, *First row as column header* and the *Static text* editor.

XML

The *Static XML string* editor.

String list

String list ID, *Column delimiter* and *Custom delimiter character*.

Item model

Item model ID, *Use model appearance roles* and *Relation Data Specification*.

Custom

Custom specification, *Description* and a *Columns* list.

The screenshot shows the 'Data source properties' dialog for a 'Text' data source. The 'Data source type' is set to 'Text', and it is 'Enabled'. The 'Data source ID' is 'DS0'. The 'Opening/running role' is 'Beginning of the report', and the 'Primary key column index' is '0'. The 'Location type' is 'Static', and 'Allow empty data source' is checked. In the 'Text datasource' section, the 'Column delimiter' is 'Tab', the 'Encoding' is 'UTF-8', and the 'First row as column header' checkbox is unchecked. The 'Static text' area contains the character '0'. The dialog has 'OK' and 'Cancel' buttons at the bottom right.

Figure 11. Text data source page

The screenshot shows the 'Data sources' dialog box. On the left, a list titled 'Data sources' contains the entry 'imds'. On the right, the 'Data source properties' section includes the following fields: 'Data source type' (dropdown menu showing 'Item model'), 'Data source ID' (text box with 'imds'), 'Opening/running role' (dropdown menu showing 'Beginning of the report'), and 'Location type' (dropdown menu showing 'Static'). There are also checkboxes for 'Enabled' (checked) and 'Allow empty data source' (checked). A 'Parent datasource ID' text box is empty. Below this, the 'Item model datasource' section has an 'Item model ID' text box with 'model1' and an unchecked checkbox for 'Use model appearance roles'. A large empty text area labeled 'Relation Data Specification' is at the bottom. The dialog has 'OK' and 'Cancel' buttons at the bottom right.

Figure 12. Item model data source page

After the data sources are defined, their columns appear in the Designer's data source tree, from where they can be dragged into the report sections.

Chapter 11. Data/Script Expressions

Since version 2.0, NCReport can handle data and script expressions using the Qt Script module, a powerful feature of Qt (available since Qt 4.3). Qt Script is based on the ECMAScript scripting language, as defined in the ECMA-262 standard. Fields can even contain script code instead of a simple data source column, parameter, or variable. In this case, the report engine evaluates the specified script code each time the fields are refreshed. Report items can also have a **Print only when expression is true** (short name: **printWhen**) property. Print-when expressions are script expressions too, but they must always return a boolean result. To use a script expression in fields, you have to set the field source type to **Expression** in the **Field property dialog**.



In Qt 6, the Qt Script module is no longer available. Instead, the report script expressions are evaluated by [QJSEngine](#) (the JavaScript engine of the Qt QML module). The JavaScript grammar differs slightly from the Qt Script grammar, so you may need to update some script expressions in your reports when you migrate to Qt 6.

11.1. Using references in expressions

Expressions can contain the following references: data source column, parameter, variable, field result, and predefined script. When expressions are evaluated, the references are always replaced with their current values. The syntax of the references is as follows:

Syntax	Description
<code>\$D{[datasourceID.]column}</code>	Data source column reference. Returns the current value of the data source column from the current data row/record. If <code>datasourceID</code> is not specified, the default current data source (the one assigned to the current detail) is used.
<code>\$P{parameterID}</code>	Parameter reference. Returns the value of the parameter by name/ID.
<code>\$V{variableID}</code>	Variable reference. Returns the current value of the variable by name/ID.
<code>\$F{fieldID}</code>	Field reference. Returns the current display value of the specified field. The fieldID is the auto-generated but editable ID of the field, generated when the field is added to a section.
<code>\$S{scriptID}</code>	Script reference. Inserts the predefined script stored in the report under the specified ID. See Chapter 13 .



If an expression contains a reference of string/text type, the reference must be enclosed in quotation marks. For example: `"$D{ds.lastname}"=="Smith"`. You don't need quote marks for numeric or boolean values; for example, `$D{price}==750.0` is a correct formula.

11.2. References in templates

A template is a text in which you can embed data references without using script formulas. Templates can be used in fields as a source type, or in rich texts if template mode is on.

Example of using data references in templates:

```
First name: ${datasource1.firstname} Last name: ${datasource1.lastname} Date from: ${P{date_from}}
```

11.3. Reference examples

Example of using a script expression in fields:

```
"${datasource1.productName}"+ " first string "+" second string "+"${P{parametername}}"
```

Example of using a script expression as the **Print only when expression is true** property. The expression must return a logical value:

```
${productPrice}<1500
```

Example of using data references in templates:

```
"Date period: ${ds.datefrom} - ${ds.dateto}"  
"Dear ${ds.firstname} ${ds.lastname},"
```

11.4. Testing Field Expression

Now let's try out how expressions work with fields, using our previous report example. Open the report in the designer and select the productName field in the detail section. Open the field properties dialog by double-clicking on the field item. Change the field source type to Expression, then modify the field column expression to the following script expression:

```
if (${ProductID}>40) "Product: "+"${ProductName}"; else "";
```

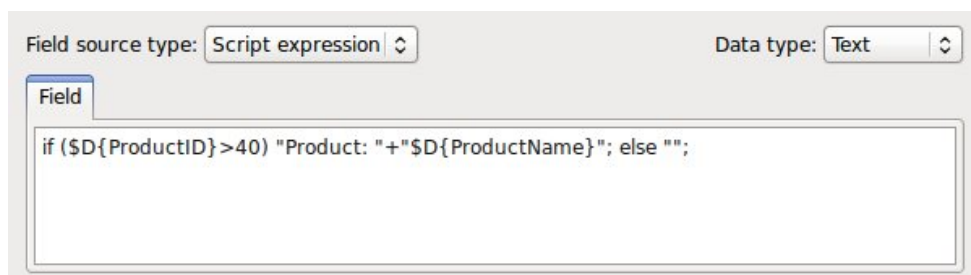


Figure 13. Field expression

In this case, the report engine first replaces the references in the code and then evaluates the script code before each rendering action. Close the dialog by clicking **[OK]**, then save the report. Now run

the report to the preview window and look at the result:

Product ID	Product name	Unit Qty	Unit price	Value
17		39.00	33.00	1 287,00
3		10.00	9.00	90,00
40		18.40	16.00	294,40
60	Product: Camembert Pierrot	34.00	28.00	952,00
18		62.50	46.00	2 875,00
38		263.50	131.00	34 518,50
39		18.00	16.00	288,00
4		22.00	20.00	440,00
5		21.35	19.00	405,65
48	Product: Chocolade	12.75	11.00	140,25
58	Product: Escargots de Bourgogne	13.25	12.00	159,00
52	Product: Filo Mix	7.00	6.00	42,00
71	Product: Flotemysost	21.50	19.00	408,50
33		2.50	2.00	5,00
15		15.50	14.00	217,00
56	Product: Gnocchi di nonna Alice	38.00	32.00	1 216,00

Figure 14. Result of field expression

This example works, but it is not the most effective way of using expressions. In most cases, when you use expressions in fields, you don't need very complex code. If you need a condition to determine whether your field should be visible, we recommend using the **Print only when expression is true** feature instead. We test this feature in the next section.

11.5. Print-When Expressions

A print-when expression is always a logical script expression that returns true or false. It is considered only if it is defined for an item or section. When the print-when expression is set, the item (or section) is printed only when the expression returns true. As usual, the expression can contain any data source, parameter, or variable reference.

Examples:

Hide an item when the value of the DataSource1.intcolumn column is less than 10:

```
$D{DataSource1.intcolumn}>=10
```

A print-when example with a boolean column:

```
$D{DataSource1.boolcolumn}
```

The item appears when DataSource1.stringcolumn is not empty:

```
"$D{DataSource1.stringcolumn}">"
```

11.6. Testing a print-when expression

Print-when script expressions are code that returns a boolean result; they are often called logical expressions. To test one, open the field properties dialog by double-clicking the same field item, and type the following code as the print-when logical expression:

```
$D{ProductID}>40
```

11.7. Print only when expression is true condition

Print only when expression is true:

Zone ID:

Then modify the previous field column expression to the following:

```
"Product: "+"$D{ProductName}"
```

After you confirm the settings and save the report, run the report again. You should get the same result.

11.8. Templates in Fields and Texts

Templates are special expressions in which the data references are simply embedded in a text. A template is not a script, so you cannot use script language elements, only data source, parameter, and variable references. For example:

```
Customer name: $D{ds1.name} Address: $D{ds1.address}  
Interval: $P{datefrom} - $P{dateto}
```



Template expressions are faster than script expressions because they require no evaluation, only a simple insertion.

11.9. Script expressions in special locations

Instead of a data reference, parameter reference, or variable, it is even possible to use a script expression in some special locations, such as a group expression or a text item. This feature makes grouping and text manipulation even more flexible.

11.9.1. Script as group expression

If you want to use a direct script as a group expression, you can write a script expression between `<%` `%>` markers. Since this is a single-line property, avoid building overly complex expressions here. Example:

```
<% if ($D{ds.age}<12) "KID"; else "MALE"; %>
```

A predefined script (see [Chapter 13](#)) can also be referenced within the markers:

```
<% $S{myscript} %>
```

11.9.2. Date functions in group expressions

A group expression can also group data by a part of a date column. The following functions are available; the function name is followed by the column reference between curly braces:

\$YEAR{column}

Groups by the year of the date.

\$MONTH{column}

Groups by the month of the date (1 to 12).

\$WEEK{column}

Groups by the week number of the date.

\$DAY{column}

Groups by the day of the week (1 = Monday to 7 = Sunday).

Example:

```
$MONTH{orders.OrderDate}
```

11.9.3. Script in HTML Texts

The HTML text item has many flexible scripting features. You can embed data references or expressions if template mode is turned on. To include a script expression within a text, use the `<% %>` markers. Within the markers you can write a script; the result of the script will be inserted into the text at the script's location.

11.10. Data Source Functions

Field expressions can contain some simple function references. These functions help you apply basic operations on data or get meta information from data sources. There are data source level functions and data column level functions. The function name is appended to the data source ID or the column ID, depending on the function, separated by a dot.

11.11. Data source (meta) functions

The syntax of a data source function:

```
DataSourceId.function()
```

In scripts or templates:

```
$D{DataSourceId.function()}
```

rowCount()

Returns the number of rows in the data source.

Example:

```
products.rowCount() or $D{products.rowCount()}
```

isAvailable()

Returns the result of the data source's `isAvailable()` method.

Example:

```
products.isAvailable() or $D{products.isAvailable()}
```

isValid()

Returns the result of the data source's `isValid()` method.

Example:

```
products.isValid() or $D{products.isValid()}
```

isEmpty()

Returns true if the data source has no data records.

Example:

```
products.isEmpty() or $D{products.isEmpty()}
```

isNotEmpty()

Returns true if the data source has at least one data record.

Example:

```
products.isNotEmpty() or $D{products.isNotEmpty()}
```

update()

Forces an update of the data source by calling its `update()` method, then moves to the first record. The function returns an empty value. This can be useful if you want to update a user-defined data

source manually. Use this function carefully.

Example:

```
products.update() or ${products.update()}
```

11.12. Data source column (value) functions

The data source column functions help with some basic text operations.

The syntax of a data source column function:

```
DataSourceId.column.function()
```

In a Detail section when using the assigned data source:

```
column.function()
```

In scripts or templates:

```
${DataSourceId.column.function()}
```

In a Detail section when using the assigned data source:

```
${column.function()}
```

MID(pos,n)

Returns a substring that contains n characters of the value, starting at the specified position pos (the first character is at position 0).

Example:

```
DataSource1.firstname.MID(2,5) or ${DataSource1.firstname.MID(2,5)}
```

LEFT(n)

Returns a substring that contains the n leftmost characters of the value.

Example:

```
DataSource1.firstname.LEFT(3) or ${DataSource1.firstname.LEFT(3)}
```

RIGHT(n)

Returns a substring that contains the n rightmost characters of the value.

Example:

```
DataSource1.firstname.RIGHT(2) or ${DataSource1.firstname.RIGHT(2)}
```

USERFUNC(arguments)

Executes the `NCReportDataSource::getUserFunctionValue(value, arguments)` method of the data source and returns its result. The text between the parentheses is passed as the `arguments` string. This is intended for custom-implemented data sources.

Example:

```
DataSource1.lastname.USERFUNC() or ${DataSource1.lastname.USERFUNC()}
```

Chapter 12. Variables and Totals

Variables are the report engine's built-in accumulators. They provide counts, sums, averages, and other aggregate values such as group totals, page totals, running balances, and grand totals. The basics of creating a variable in the Designer are shown in the "Variables and Groups" section of the Getting Started chapter, and the XML format is listed in the Specification part. This chapter explains how variables work at run time: when they are updated and reset, what each section of the report sees, and how to build typical totals with them. The last section is the complete reference of the built-in system variables.

12.1. How variables work

A variable is defined once for the whole report, in the `<variables>` element of the report definition. In the Designer, select **Report > Variables...** and edit the variables in the table of the dialog (columns **ID**, **Expression**, **Function**, **Scope**, and **Init Value**).

```
<variables>
  <variable id="total_value" type="num" funcType="sum" scope="group" initValue="0">value</variable>
  <variable id="balance" type="num" funcType="sum" scope="report" initValue="0">value</variable>
  <variable id="rows" type="num" funcType="count" scope="page" initValue="0">ProductID</variable>
</variables>
```

id

The name of the variable. In expressions and templates, the variable is referenced as `$V{id}`; in a field, set the field source type to **Variable** (`fType="var"`) and enter the ID as the column.

funcType

The aggregate function: `count`, `sum`, `average`, `std`, `min`, or `max` (case-insensitive). An unknown name is read as `count`.

scope

The reset scope: `report`, `page`, or `group` (case-insensitive). An unknown name is read as `report`.

initValue

The value the variable starts from and returns to whenever it is reset.

type

The data type. Variables are always numeric (floating point) in the engine; the Designer writes `type="num"`, and the attribute is not evaluated when the report is loaded.

Element text

The variable expression: the source of the value that is aggregated (see [Section 12.1.2](#)).

12.1.1. Function types

Each time the variable is updated, the current value of its expression is passed to the aggregate

function.

Function	Result
COUNT	The number of updates, that is, the number of processed rows (the expression value itself is not used, but it must be a valid column). The initial value is the starting count.
SUM	The sum of the values.
AVERAGE	The arithmetic mean of the values processed since the last reset.
STD	The standard deviation of the values processed since the last reset. The engine calculates the population standard deviation (the squared deviations are divided by n , not $n-1$).
MIN	The smallest value. The initial value setting is ignored: a MIN variable always starts from the largest representable number (about 1.8e308), so do not print it before the first row has been processed.
MAX	The largest value. The comparison starts from the initial value, so if all values can be negative, set the initial value to a number smaller than any expected value (the default 0 would otherwise be the result).

12.1.2. The variable expression

The variable expression can take two forms:

Data source column

The name of a column of the detail's data source, for example `value`, or a column of a specific data source in the `datasourceID.column` form, for example `DataSource1.value`. Note that the column is written without the `$D{}` reference syntax. The variable is then bound to that data source (see [Section 12.1.5](#)).

Script expression

If the expression contains the `<%` and `%>` tokens, the text is evaluated like a field expression: the data source, parameter, and variable references are substituted, the embedded script is executed, and the result is converted to a number. The `Variable_Expression.ncr` demo report sums a calculated ratio this way:

```
<variable id="calculations" type="num" funcType="sum" scope="report" initValue="0">
&lt; %$D{width}/$D{height}%></variable>
```

In the Designer you simply type `<%$D{width}/$D{height}%>` into the **Expression** column; the XML escaping is done automatically.



A variable based on a script expression is not bound to any data source, so the automatic page break correction described in [Section 12.1.4](#) does not apply to it. Use a plain column expression for page totals and carried-forward balances whenever possible.

12.1.3. Reset scopes

The scope determines when the variable returns to its initial value:

Report

The variable is reset only when the report starts. It accumulates through the whole report: use it for grand totals and running balances.

Page

The variable is reset at the beginning of every page, before the page header is printed. Use it for page totals.

Group

The variable is reset after the footer of a group is printed, but only if the variable is listed in that group's **Reset variables** list (the `resetVariables` attribute of the `<group>` element). A `Group` scope variable that no group resets behaves like a `Report` scope variable.

In addition, all variables are reset when the report starts (in double pass mode, at the start of both passes), and when the engine moves to the next detail section, the variables bound to the data source of the new detail are reset, whatever their scope.



In the Designer, only `Group` scope variables are offered in the **Reset variables** list of the group dialog. The engine itself resets every variable named in the `resetVariables` attribute, regardless of its scope.

12.1.4. Evaluation timing

All variables are updated once for every data row, before the row is printed. For each row of a detail section, the engine performs the following steps:

1. If the value of a group expression has changed, the footers of the closed groups are printed (the innermost group first). The footers still see the variable values of the previous row, that is, the totals of the closed group. After each footer, the variables listed in that group's **Reset variables** list are reset.
2. All variables are updated with the current row.
3. If a new group starts, the headers of the new groups are printed. During the group header, the update made by the current row is temporarily taken back, so a group header shows the value *before* the first row of the group (0 for a freshly reset group variable).
4. The detail section is printed. The detail sees the value *including* the current row, so a `Report` scope `SUM` variable printed in the detail is a running total.

The page and report level sections see the following values:

Section	Value of the variables
Report header	The initial values (the first row is temporarily taken back).

Section	Value of the variables
Page header	The values after the page reset: Page scope variables show their initial value, Report and Group scope variables show the total of all rows printed on the previous pages (the "brought forward" value).
Detail	The values including the current row.
Group header	The values before the first row of the new group.
Group footer	The values including the last row of the group.
Page footer	The values including the last row printed on the page.
Report footer	The final values. Note that Group scope variables have already been reset by the final group footers, so use a Report scope variable for a grand total in the report footer.

The page break needs special care, because when the engine finds that a detail row does not fit on the page, the variables have already been updated with that row. In this case the engine steps back one row and takes back the last update of the variables bound to the current data source, prints the page footer, begins the new page (resetting the **Page** scope variables and printing the page header), and only then reapplies the update. As a result, the page footer of the old page does not contain the row that moved to the next page, and the row is counted in the **Page** scope variables of the new page. The same correction is applied when a group header does not fit on the page.

This correction is controlled by the **Data trimming** report option (the `dataTrimming` option element), which is on by default. If you turn it off, the page footer and the page header of the next page show values that already include the first row of the next page.



The correction can take back only the last update. With **MIN** and **MAX** variables the step back cannot restore the previous value, so these functions are not suitable for page level values. A repeated group header (**Reprint header on new page**) is printed after the update has been reapplied, so in a repeated group header a group variable already includes the first row of the new page.

12.1.5. Variables in reports with more than one detail

By default, every variable is updated for every row of every detail. A variable bound to a column of another data source is updated with that data source's current value, which is rarely what you want. If the report has more than one detail section, turn on the `updateOnlyCurrentDataSourceVariables` report option: then a variable bound to a data source column is updated only while a detail of the same data source is being processed. Variables with a script expression are always updated.

To get an alert when a variable expression cannot be evaluated (for example, a misspelled column name), turn on the `varErrorAlert` report option; otherwise an invalid variable is silently left unchanged.

12.2. Page totals and running balances

Invoices, ledgers, and long lists often need three kinds of totals on every page: the total of the current page, the running balance up to the end of the page, and the balance carried over from the previous page at the top of the next page. With the timing rules above, all three need only two variables:

- a **Page** scope **SUM** variable for the page total, printed in the page footer;
- a **Report** scope **SUM** variable for the running balance, printed in the page footer as the "carried forward" value and in the page header as the "brought forward" value.

The `PageTotalsAndBalanceInProductlist.xml` demo report prints the running balance of the `value` column in the detail, in the page footer, and in the page header. The `page_total_example.xml` demo does the same in a grouped product list, and prints the grand total in the footer of a constant (`%CONST`) group. The following example is based on these demos, extended with a page total variable. Geometry and font attributes are omitted.

```
<variables>
  <variable id="page_total" type="num" funcType="sum" scope="page" initValue="0">value</variable>
  <variable id="balance" type="num" funcType="sum" scope="report" initValue="0">value</variable>
</variables>
<pageheader id="phead">
  <label id="1">Brought forward:</label>
  <field id="2" type="num" ftype="var" formatting="true" format="f" precision="2"
    printWhen="$V{pageno} > 1">balance</field>
</pageheader>
<pagefooter id="pfoot">
  <label id="1">Page total:</label>
  <field id="2" type="num" ftype="var" formatting="true" format="f" precision="2">page_total</field>
  <label id="3">Carried forward:</label>
  <field id="4" type="num" ftype="var" formatting="true" format="f" precision="2">balance</field>
  <field id="5" type="num" ftype="sys" arg="Page: %1">pageno</field>
</pagefooter>
<details>
  <detail id="Detail1" datasource="DataSource1">
    <items>
      <field id="1" type="num" ftype="ds">ProductID</field>
      <field id="2" type="txt" ftype="ds">ProductName</field>
      <field id="3" type="num" ftype="ds" formatting="true" format="f" precision="2">value</field>
      <!-- running balance in every row -->
      <field id="4" type="num" ftype="var" formatting="true" format="f" precision="2">balance</field>
    </items>
  </detail>
</details>
```

Suppose the first page holds rows with the values 10, 20, and 30, and the next row (40) does not fit. The report then prints:

Place	page_total	balance
Page 1 header	0	0 (hidden by the print condition)
Page 1 detail rows	10, 30, 60	10, 30, 60
Page 1 footer	60	60 (carried forward)

Place	page_total	balance
Page 2 header	0	60 (brought forward)
Page 2 first detail row	40	100

The `printWhen` expression of the brought-forward field hides it on the first page, where there is nothing to bring forward. The same technique works for counts (`COUNT`) and for group level running totals: a `Group` scope variable printed in the page header shows the running total of the current group up to the end of the previous page.



In the Designer, set the **Reset scope** to `Page` for the page total, and `Report` for the balance. No group is needed for these variables. If you also need a grand total at the end of the report, print the `Report` scope variable in the report footer or in the footer of a constant group.

12.3. Group totals and nested group totals

A group total is a `Group` scope variable that is reset by its group. Since the reset happens after the group footer is printed, the footer shows the total of the whole group, and the next group starts from the initial value again. The `GroupFunc.ncr` demo, for example, sums the `col0` column by month:

```
<variables>
  <variable id="total" type="num" funcType="sum" scope="group" initValue="0">col0</variable>
</variables>
...
<group id="Group0" groupExp="$MONTH{col1}" resetVariables="total" reprintHeader="false"
startsOnNewPage="false">
  <groupheader id="GH_Group0">...</groupheader>
  <groupfooter id="GF_Group0">
    <field id="0R2HF" type="num" ftype="var">total</field>
  </groupfooter>
</group>
```

For nested groups, define one variable for each level, each with the same expression, and let every group reset only its own variable. The groups of a detail are nested in the order in which they are listed: the first group is the outermost one. When an outer group changes, all of the inner groups are closed too, so the inner footers are printed first (and reset their variables), then the outer footer. The `group_simple_totals.xml` demo uses a constant outer group for the grand total and an inner group for the subtotals:

```
<variables>
  <variable id="total_0" type="num" funcType="sum" scope="group" initValue="0">price</variable>
  <variable id="total_1" type="num" funcType="sum" scope="group" initValue="0">price</variable>
</variables>
...
<groups>
  <group id="Group0" groupExp="%CONST" resetVariables="total_0">
    <groupheader/>
    <groupfooter>
      <field id="1" type="num" ftype="var">total_0</field>    <!-- grand total -->
    </groupfooter>
```

```

</group>
<group id="Group1" groupExp="groupID" resetVariables="total_1">
  <groupheader>...</groupheader>
  <groupfooter>
    <field id="0" type="num" ftype="var">total_1</field>    <!-- subtotal of the groupID group -->
  </groupfooter>
</group>
</groups>

```

The `resetVariables` attribute is a comma-separated list, so one group can reset several variables, for example a sum and a count of the same group (`resetVariables="sum_price,count_items"`). A variable must not be listed in an outer group only, or the inner footers would show the accumulated value of the outer group.

Other useful patterns:

- A row number restarting in every group: a `COUNT` variable reset by the group and printed in the detail. The `variable-row-index.ncr` demo uses this technique.
- A group average: an `AVERAGE` variable reset by the group, printed in the group footer.
- A group total printed in the page header and footer: the `GroupTotals.xml` demo prints `Group` scope variables in the page header, the page footer, the group headers (where they show the value before the first row of the group), and the group footers.

12.4. Modifying variables

In special cases, the regular aggregation is not enough: you may want to add an extra amount to a total, subtract a value under a condition, or restart a counter in the middle of a group. For this purpose, the report engine contains a variable modifier (the `NCReportVariableModifier` class), which changes the current value of a variable directly.

The modification lines are written into the conditional formatting code of a Label or a Field (see [Chapter 31](#)). A line that starts with `$V{` is not a style definition but a variable modification, in the following form:

```
$V{variableID} operator expression
```

The available operators are:

Operator	Effect
=	Sets the variable to the value of the expression.
+=	Adds the value to the variable.
-=	Subtracts the value from the variable.
*=	Multiplies the variable by the value.
/=	Divides the variable by the value.

The expression on the right side can contain data source, parameter, and variable references, and it is

evaluated by the script engine, so it can be a constant, a reference, or a script expression such as `$D{price}*$D{qty}` or `($D{discount}>0 ? 1 : 0)`. A `COUNT` variable is set to the integer part of the result.

The modification is executed when the item is printed: each time the section that contains the item is rendered and the item's print condition (**Print only when expression is true**) is true. Therefore the print condition of the item acts as the condition of the modification. The new value is visible for the items rendered after the modifying item in the same section, and for all later sections. The variable's own aggregation continues from the modified value at the next row.

The `Variable-Modify-By-CondFormat-Demo.xml` demo report has three `Report` scope variables. The label in the detail section changes its font weight conditionally, and also adds 2 to the `v3` variable in every row:

```
<variables>
  <variable id="v2" type="num" funcType="sum" scope="report" initValue="0">col0</variable>
  <variable id="v3" type="num" funcType="sum" scope="report" initValue="0">col0</variable>
  <variable id="v1" type="num" funcType="count" scope="report" initValue="0">col0</variable>
</variables>
...
<detail id="Detail1" datasource="DataSource1">
  <items>
    <label id="1" cf="1">
      <tr>Lorem ipsum </tr>
      <cf>font-bold:{if($D{col1}>20) true; else false;}
$V{v3}+=2
</cf>
    </label>
    <field id="N7Q30" type="txt" ftype="var">v1</field>
    <field id="HTPFP" type="txt" ftype="var">v2</field>
    <field id="ROE94" type="num" ftype="var" formatting="true" format="f" precision="2">v3</field>
  </items>
</detail>
```

The `v2` and `v3` variables sum the same column, but because the label is rendered before the `v3` field, the `v3` column shows the running sum plus 2 for every row printed so far (in the n -th row, $v2 + 2 * n$).

Keep the following rules in mind:

- The `$V{` must be at the very beginning of the line, and every modification must be on its own line. The style lines and the modification lines can be mixed in any order; the style evaluation skips the modification lines.
- The operator is identified by the first of the strings `+=`, `-=`, `/=`, `*=`, and `=` that occurs in the line. If you use the `=` operator, the expression must not contain any further `=` character (for example `==`, `>=`, or `<=`); use another form, or move the condition into the print condition of the item.
- An unknown variable ID or an empty expression makes the engine skip the rest of the modification lines of the item.
- The modification is applied every time the item is printed. If the section is printed more than once (for example, a section with automatic height that did not fit on the page and is printed again on the next page), the modification is applied again. The automatic page break correction (see [Section 12.1.4](#)) does not take back a modification.

12.5. System variables

System variables are values provided by the report engine itself, such as the page number, the current date, or the name of the application. They are read-only and do not need to be defined. A system variable can be used:

- in a field with the **System variable** source type (`ftype="sys"`), entering the name as the column;
- in a field with the **Variable** source type, in an expression, or in a template, with the `$V{name}` syntax, for example `Page $V{pageno} of $V{pagecount}`.

With the `$V{}` syntax, the name is first looked up among the report's own variables, so a user variable with the same name hides the system variable. The names are case-sensitive. An unknown name returns the text `#VAR#`.

The following table lists all of the system variables recognized by the report engine.

Name	Value	Notes
<code>pageno</code> , <code>pagenum</code>	The current page number of the report.	At least 1. Restarts in every report of a batch, in every forced copy, at a group with the Reset page number option, and, if the <code>subReportResetPageNo</code> option is on, in every sub-report iteration.
<code>pagecount</code>	The total number of pages of the report.	Available only in double pass mode (see "Double pass mode") and in Text Document mode; otherwise it returns 0. If page numbering is restarted by a group, it is the number of pages counted since the last restart.
<code>gpageno</code>	The global page number.	Counted continuously across the reports of a batch (see Chapter 34) and across forced copies. Without batch mode and forced copies it equals <code>pageno</code> , unless page numbering is restarted.
<code>reportno</code> , <code>reportnum</code>	The sequence number of the current report in a batch, starting from 1.	1 if the report does not run in batch mode.
<code>reportcount</code>	The number of reports in the batch.	1 if the report does not run in batch mode.
<code>forcecopies</code>	The number of forced copies.	Set by <code>NCReport::setForceCopies()</code> ; the default is 1.
<code>currentcopy</code>	The number of the copy being rendered when forced copies are used.	Zero-based: the first copy is 0, the last is <code>forcecopies - 1</code> .
<code>currentrow</code>	The current row number of the detail.	Zero-based in normal reports: the first row of a detail is 0. It restarts in every detail section. In a group footer, it refers to the last row of the group.
<code>groupcount</code>	The number of groups of the current group level printed so far.	Meaningful only in group headers and group footers; the counter belongs to the group of the section being printed. It is not offered in the Designer's system variable list; type the name manually.

Name	Value	Notes
<code>date</code>	The current date.	<code>QDate::currentDate()</code> , read each time it is evaluated. Use the date formatting options of the field to format it.
<code>time</code>	The current time.	<code>QTime::currentTime()</code> , read each time it is evaluated.
<code>datetime</code>	The current date and time (timestamp).	<code>QDateTime::currentDateTime()</code> , read each time it is evaluated.
<code>appname</code>	The name of the report engine: <code>NCReport</code> .	Constant of the NCReport library, not of your application.
<code>applongname</code>	The long name of the report engine: <code>NCReport Report Generator</code> .	Constant of the NCReport library.
<code>appversion</code>	The version number of the NCReport library, for example <code>2.35.2</code> .	Constant of the NCReport library.
<code>appinfo</code>	The full info string of the report engine: name, version, release name (Commercial or Evaluation Version), and copyright.	Depending on the build, the text may end after the version number.
<code>appcopyright</code>	The copyright notice of NCReport.	Constant of the NCReport library.
<code>coreappname</code>	The name of the host application.	<code>QCoreApplication::applicationName()</code> , as set by your application.
<code>coreappversion</code>	The version of the host application.	<code>QCoreApplication::applicationVersion()</code> .
<code>coreorgname</code>	The organization name of the host application.	<code>QCoreApplication::organizationName()</code> .
<code>coreorgdomain</code>	The organization domain of the host application.	<code>QCoreApplication::organizationDomain()</code> .
<code>os</code>	The operating system: <code>Windows</code> , <code>Linux</code> , or <code>MacOS</code> .	Determined at compile time. Empty on other platforms.
<code>qtversion</code>	The version of the Qt library at run time.	<code>qVersion()</code> , for example <code>6.8.2</code> .
<code>lang, language</code>	The current language code of the report.	See "Internationalization". The default (no translation) language code is <code>00</code> .



The `app*` variables describe the NCReport engine. To print the name and version of your own application, set them with `QCoreApplication::setApplicationName()` and `setApplicationVersion()` and use the `coreappname` and `coreappversion` variables.

Chapter 13. Script Editor

The script editor lets you store predefined custom scripts within the report. Storing scripts in the report avoids duplicating script code assigned to report items or used anywhere else in the report. For example, it can be used efficiently for print-when expressions.

To reference a script anywhere in the report, use the `$$S{scriptId}` token, where *scriptId* is the ID you assigned to the script.



Use predefined scripts when you need complex code or use the same script expression more than once in a report. This is typically useful for multiple instances of the same print-when expression.

To define your scripts in the Designer, select **Report > Scripts...** (the equivalent tool button is on the toolbar). A dialog appears where you can add, edit, and remove scripts. To add a new script, click **[Add]** and type the script ID. To remove a script, click **[Remove]**. To use the expression builder, click **[Expression builder...]** to open the expression builder dialog.

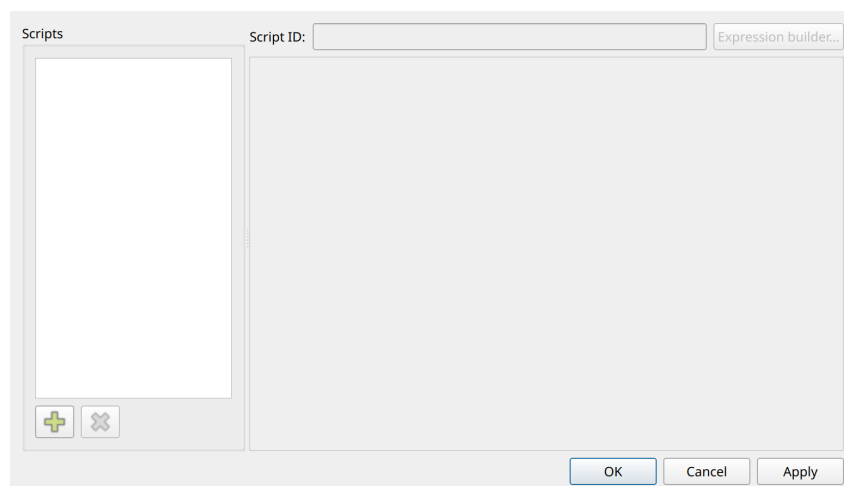


Figure 15. Script editor dialog

13.1. Script ID

The script identifier. You reference the script with this ID anywhere in the report, for example: `$$S{myscript1}`

13.2. Script Definition

Enter your code here. You can use any report data reference in the code, following the usual rules for NCReport data references.

13.3. Available Buttons

[Apply]

Applies your settings without closing the dialog.

[OK]

Applies your settings and closes the dialog.

[Cancel]

Closes the editor without saving any changes and returns to the Designer desktop.

Chapter 14. Data Formatting

You can specify formatting options for fields. Fields support the **Text**, **Numeric**, **Date**, **DateTime**, **Time**, and **Boolean** data types, many of which require formatting. You can set the format options in the Designer in the **Field settings** dialog, on the **Text**, **Numeric**, and **Date/Time** tabs, depending on the data type selected in the **Data type** combo box.

14.1. Text formats

Use the **Text** tab for special text format settings. The basic settings (font, size, alignment) can be set from the toolbars. You can specify the following parameters:

Vertical alignment

The vertical alignment of the text block: Top, Center, or Bottom.

Character length

Sets the width of the string explicitly (number of characters). If the value is shorter, it is padded with the fill character, depending on the horizontal alignment (left-aligned values are padded on the right, right-aligned values on the left). The default is **Auto** (no padding).

Rotation angle

The data can be rotated. 0 is the default position; 90 or 270 makes the text vertical.

Fill character

The character used for padding when **Character length** is set (for numbers, also when **Field width** is set).

Capitalization

Mixed case (default), All uppercase, All lowercase, Small caps, or Capitalize.

Letter spacing

Spacing between letters in pixels; the default is 0.

14.2. Numeric formats

If the **Data type** is numeric, use the **Numeric** tab for special format settings. The basic settings (font, size, alignment) can be set from the toolbars. You can specify the following parameters:

Number formatting

Enables or disables number formatting. It is turned off by default; in this case, the value is printed as it is.

Use localized settings

If this is turned on, the number is formatted according to the report's locale (QLocale), for example with localized decimal and group separators.

Blank if value equals zero

If this is turned on and the value equals zero, nothing is shown; the zero value is hidden.

Substitution string if value equals zero

Turn on this setting if you want to replace zero values with a string, for instance '- '.

Substitution string

You can specify the substitution string here.

Decimal precision

The number of decimal places. It is considered only if number formatting is turned on.

Field width

The minimum width of the number in characters (not used with localized formatting). Shorter numbers are padded with the fill character.

Format character

The number format character, as used by `QString::arg()` and `QLocale::toString()`: `f` (fixed-point, e.g. 123.45), `e` or `E` (scientific notation, e.g. 1.2345e+02), or `g` or `G` (the more concise of the two). If **Use localized settings** is on, `C` formats the number as a currency string using the report's locale.

Field source type: Data source Data type: Text

Field: col0

Build expression...

General Text Numeric **Date/Time**

Options

- ☐ Number formatting
- ☐ Use localized settings
- ☐ Blank if value equals zero
- ☐ Substitution string if value equals zero

Decimals

Decimal precision: 0

Field width: Auto

Format character: f

Substitution string:

Identify Print when HTML Dynamic position Dynamic style TOC

Item ID: 1LDDE Zone ID: 0

Function ID:

☐ Page break after printing this item

Conditional format... OK Cancel

14.3. Date Formats

Fields can, of course, display date, datetime, and time data. The format can be specified in the **Field settings** dialog on the **Date/Time** tab, in the **Date / DateTime format** box. If the format is left empty, the short date (or datetime) format of the report's locale is used. You can specify the format in two ways: statically or dynamically.

Field source type: Data source Data type: Text

Field: col0

Build expression...

General Text Numeric Date/Time

Date / DateTime format:

Identify Print when HTML Dynamic position Dynamic style TOC

Item ID: 1LDDE Zone ID: 0

Function ID:

☐ Page break after printing this item

Conditional format... OK Cancel

14.3.1. Static Date Format

The static date format is a Qt date format expression. Day and month names are localized according to the report's locale:

Expression	Output
d	The day as a number without a leading zero (1 to 31)
dd	The day as a number with a leading zero (01 to 31)
ddd	The abbreviated localized day name (e.g. 'Mon' to 'Sun')
dddd	The long localized day name (e.g. 'Monday' to 'Sunday')
M	The month as a number without a leading zero (1 to 12)
MM	The month as a number with a leading zero (01 to 12)
MMM	The abbreviated localized month name (e.g. 'Jan' to 'Dec')
MMMM	The long localized month name (e.g. 'January' to 'December')
yy	The year as a two digit number (00 to 99)
yyyy	The year as a four digit number. If the year is negative, a minus sign is prepended, making five characters.

For DateTime and Time data, the following time expressions can also be used:

Expression	Output
h	The hour without a leading zero (0 to 23, or 1 to 12 if AM/PM display is used)
hh	The hour with a leading zero (00 to 23, or 01 to 12 if AM/PM display is used)

m	The minute without a leading zero (0 to 59)
mm	The minute with a leading zero (00 to 59)
s	The second without a leading zero (0 to 59)
ss	The second with a leading zero (00 to 59)
zzz	The milliseconds with leading zeroes (000 to 999)
AP or ap	AM/PM or am/pm display

Example: `yyyy-MM-dd hh:mm:ss`

14.3.2. Durations

If the data type is **Time** and the data is an integer number that represents a duration, append a `|` character and a unit marker to the format: `s` for seconds, `z` for milliseconds, `m` for minutes, or `h` for hours. For example, the `hh:mm:ss|s` format displays the value 3725 as `01:02:05`.

14.3.3. Dynamic Date Format

The dynamic date format takes the format string from a parameter, a data source column, or a variable. Use the usual data reference syntax:

Example:

```
$D{ds.dformat} or $P{mydateformatparam}
```

The dynamic date format is recognized by the report engine and evaluated during rendering. It is up to you whether a parameter or a data source stores the dynamic date format.

14.4. Template arg() string

The **Template arg() string** option on the **General** tab of the **Field settings** dialog wraps the formatted value in a text. The string must contain the `%1` placeholder, which is replaced by the formatted value (as in `QString::arg()`). For example, the `Total: %1 USD` string prints `Total: 1250.00 USD`.

14.5. Boolean values

Boolean data is displayed as `true` or `false` (these words are translatable).

Chapter 15. Text and Item Display Options

This chapter collects the display options of the text-bearing report items that are not covered by the **Data Formatting** chapter: the content modes of HTML text items (rich text, plain text and Markdown), hiding repeated field values, breaking long rich text across pages, and rotating text.

15.1. Rich Text, Plain Text and Markdown in Text Items

The **Text** item (`<text>` element, **HTML Text Properties** dialog in the Designer) renders its content with Qt's rich text engine (`QTextDocument`). How the content is interpreted depends on two options:

Mode	plain	markdown	Content handling
Rich text (default)	-	-	The content is HTML (the subset of HTML supported by Qt's rich text engine) and is passed to the document as it is.
Plain text	true	-	The content is plain text. Without Markdown integration (see Section 15.2), each line break becomes <code>
</code> and the whole text is wrapped in a paragraph. With Markdown integration built in, the plain text is converted to HTML by the Markdown processor.
Markdown	(true)	true	The content is Markdown text and is loaded with <code>QTextDocument::setMarkdown()</code> (Qt 5.14 or later).

These modes apply to every content source of the text item: static text, file, data source column, parameter, file name from a data source and URL. The content is converted each time the item value is evaluated, so dynamic content (for example a data source column that contains Markdown) is converted record by record.

In the Designer, the options are check boxes in the **Options** group of the **HTML Text Properties** dialog:

Plain text

Turns on plain text mode (`plain="true"`). When it is checked, the WYSIWYG **Editor mode** is disabled and the dialog switches to **Source code mode**, because the content is no longer HTML.

Markdown conversion

Turns on Markdown mode (`markdown="true"`). Checking it also checks **Plain text**.

Use system font settings

(`useFont="true"`) The item's own font, text color (`forecolor`) and alignment, set with the **Text properties** toolbar, are applied as the defaults of the text document, and the document margin is set to zero. Formatting in the HTML content (for example a `<span style="...">`) overrides these defaults. When the option is off, the text color and alignment of the item are not applied; use HTML formatting in the content instead.

The radio buttons **Editor mode** and **Source code mode** select the editor used in the dialog: the

WYSIWYG rich text editor or the HTML source editor with syntax highlighting. The choice is stored in the `editorMode` attribute (`0` = editor mode, `1` = source code mode); if the attribute is missing, source code mode is used. It only affects editing, not rendering.

When plain text mode is on, the item's alignment is also saved in the report definition (`alignment` attribute); together with **Use system font settings** it becomes the default alignment of the text.

15.1.1. Text case and letter options

Capitalization, letter spacing, rotation, fill character, character length and vertical alignment are options of **Label** and **Field** items. They are set on the **Text** tab of the label and field property dialogs, which is described in the **Data Formatting** chapter (**Text formats**). The capitalization is stored in the `capital` attribute as the numeric value of `QFont::Capitalization`:

<code>capital</code>	Capitalization (Designer)
(not written)	Mixed case (default)
<code>1</code>	All uppercase
<code>2</code>	All lowercase
<code>3</code>	Small caps
<code>4</code>	Capitalize (first letter of each word in uppercase)

Text items have no capitalization option in their property dialog; in HTML content, use the CSS `text-transform` and `font-variant` properties supported by Qt's rich text engine instead.

15.2. Markdown in Text Items

Markdown is a lightweight markup language that is easier to write than HTML. NCReport supports Markdown in text items in two ways.

15.2.1. Qt's built-in Markdown support

If NCReport is built with Qt 5.14 or later, a text item with the `markdown="true"` attribute loads its content with `QTextDocument::setMarkdown()`. No additional build option is needed. In the Designer, check **Markdown conversion** in the **HTML Text Properties** dialog; this also checks **Plain text**, so the item is saved with both attributes:

```
<text id="6J00M" posX="0.000" posY="0.000" width="168.910" height="124.460"
  useFont="true" fontName="Times New Roman" fontSize="10" forecolor="#5c3566"
  plain="true" markdown="true" editorMode="1" alignment="1">### Title
- subject 1
- subject 2</text>
```

With Qt versions older than 5.14, the `markdown` attribute is ignored by the engine (the item is then handled as a plain text item), and the **Markdown conversion** check box is not available in the Designer unless Markdown integration is built in.

15.2.2. Markdown integration (MARKDOWN_INTEGRATION)

NCReport can also be built with its own Markdown processor, which is based on the *Discount* Markdown library shipped in `src/3rdparty/markdown`. To enable it, add the `markdown` extension to the `EXTENSIONS` variable in `src/config.pri` before building:

```
EXTENSIONS += designer runner consoleapp barcode preview markdown
```

The extension is not part of the default extension list. When it is enabled, the build defines the `MARKDOWN_INTEGRATION` macro and compiles the Markdown processor into the engine. The macro has two effects:

- In the engine, the conversion of **plain text** content is done by the Markdown processor: a text item with `plain="true"` is converted from Markdown to HTML instead of the simple line-break conversion.
- In the Designer, the **Markdown conversion** check box is always shown, regardless of the Qt version.

When both `markdown="true"` and `plain="true"` are set and NCReport is built with Qt 5.14 or later, Qt's built-in Markdown support takes precedence. The Discount processor is used for items that have only `plain="true"`, and for all plain text items when the Qt version is older than 5.14.

15.2.3. Example

The `markdown_test.ncr` demo report in the `reports` directory contains a text item with static Markdown content (a level 3 heading and a bulleted list) in its detail section.

15.3. Hiding Repeated Values

In lists sorted by a column, the same value often appears in several consecutive rows, for example the order date in an order list. The **Hide repeated values** option prints such a value only once: a field is left blank if its value is the same as in the previous record.

In the Designer, the option is a check box on the **General** tab of the field property dialog (**Options** group). In the report definition it is the `hideRepeated` attribute of the field:

```
<field id="3" posX="0.000" posY="0.000" width="22.490" height="5.027"  
      hideRepeated="true" type="date" ftype="ds">OrderDate</field>
```

15.3.1. How it works

The engine decides at the visibility check of the field (the same check that hides empty fields):

- The comparison uses the formatted display value of the field, that is, the text that would be printed (after number and date formatting).
- The value is compared with the value the field had the last time it was evaluated with a non-empty

value, which is normally the previous record of the section. If the two are equal, the field is treated as empty and is not printed. The current value is then stored for the next comparison.

- An empty value is never compared and does not replace the stored value.
- The stored value is cleared at the start of the report for all fields, and after every page break for the fields of the current detail section. Therefore the first record on each page always shows the value, even if it is the same as the last value on the previous page.
- The stored value is not cleared when a group changes. If a group starts with the same value as the last record of the previous group, the value is hidden.
- The option is applied only if the field has no **Print When** expression. If a **Print When** expression is set, the expression alone decides whether the field is printed, and repeated values are printed.

Because the hidden field counts as empty, in a section with automatic height a zone that contains only hidden fields is not printed and the section shrinks accordingly (see the **Zones** chapter).



The `hideRepeated` attribute is also read and written for label items, but it has no effect on them, and the Designer does not show the option in the label property dialog. Hiding repeated values is a field feature.

15.3.2. Example

The `hide_repeated_orderlist_demo.xml` demo report lists the records of the `orders` table and hides the repeated values of the `OrderDate` field.

15.4. Breaking Rich Text Across Pages

A text item with automatic height grows with its content. If the content is longer than the space left on the page, the whole section would normally move to the next page, and a text longer than a page could never be printed completely. NCReport can instead slice such a text item: it prints the part that fits on the current page and continues the rest on the following pages.

15.4.1. Requirements

Slicing works only if all of the following conditions are met:

1. The item is a **Text** (HTML text) item. Other item types cannot be sliced.
2. The item can break: **Break to page** is checked in the **HTML Text Properties** dialog (`canBreak="true"`).
3. The item has automatic height: **Automatic height** is checked (`autoHeight="true"`), so its height is calculated from the content.
4. The section has automatic height (`autoHeight="true"` on the section), because slicing is part of the zone handling of automatic-height sections.
5. Page breaks between zones are allowed in the report: **Page break is allowed between zones** is checked in the **Zone settings** of the report and page settings (the `zoneBreak` report option is `true`).

```
<options>
  ...
  <zoneBreak>true</zoneBreak>
</options>
...
<detail id="Detail1" height="28.840" autoHeight="true" datasource="text">
  <items>
    <text id="UIMNF" posX="47.625" posY="4.762" width="122.237" height="4.762"
      zoneID="1" resource="datasource" autoHeight="true" canBreak="true"
      useFont="false">text.col3</text>
  </items>
</detail>
```

15.4.2. How the text is sliced

When the engine renders a section with automatic height, it collects the visible items into zones and calculates the height of each zone. If a zone does not fit into the remaining space of the page and it contains a breakable item, the engine calculates how much of the text fits:

- The text is divided at line boundaries: the engine walks through the laid-out lines of the text document (block by block, line by line) and takes as many lines as fit into the remaining height. A line is never split.
- If at least one line fits, the zone is shrunk to the height of the fitting lines, and that part of the text is printed on the current page.
- If not even one line fits, the section is not printed on the current page; it is moved to the next page.

After the first part has been printed, the engine performs a page break (the page footer and the page header are printed as on any new page) and continues the text from the next unprinted line at the top of the new page. This repeats until the whole text has been printed, so a text can span any number of pages.

Rectangle and line items of the same zone that have automatic height are redrawn around each printed part of the sliced text, so a frame drawn around a text item follows the text onto the following pages.



The sliced parts are painted only in the graphical outputs (preview, printer, PDF, SVG and image). The HTML, Excel and text outputs do not paginate the text this way.

15.4.3. Example

The `htmltext_slice_demo.xml` demo report prints long rich text values from a static text data source. Its detail section has automatic height and contains a breakable text item with automatic height, and the report has the `zoneBreak` option turned on.

15.5. Rotating Items

The following report items and objects support rotation:

Item	XML attribute	Notes
Label, Field	<code>rotation</code> (integer, degrees)	Set with Rotation angle on the Text tab of the property dialog (0-360 in the Designer). Described below.
Barcode	<code>rotation</code> (degrees)	Rotation by the Zint library: 0, 90, 180 or 270. See the Barcodes chapter.
Cross-tab cell	<code>rotation</code> (real number, degrees)	Rotation of the cell text in the cell style of the cross-table. See the Cross-Tab Tables chapter.
Whole output	-	<code>NCReport::setRotationAngle()</code> rotates the output pages; the valid angles are 90, -90, 180 and 270. See the Report Outputs chapter.

Text (HTML text), image, shape, chart and table items cannot be rotated.

15.5.1. Rotating labels and fields

A label or field with a non-zero `rotation` value is drawn rotated by the given angle in degrees (clockwise):

```
<label id="0" posX="0.000" posY="0.000" width="31.750" height="31.750"
      fontName="Liberation Sans" fontSize="10" rotation="90">Text rotated by 90 degrees</label>
<field id="WAXL7" posX="60.325" posY="1.587" width="47.625" height="49.212"
       resource="datasource" fontName="Arial" fontSize="10" rotation="270"
       type="txt" ftype="ds">name</field>
```

The engine draws rotated text in a rectangle whose width and height are swapped compared to the item: the height of the item becomes the available line length, and the width of the item becomes the available height of the text. Design rotated items accordingly; for example, a vertical caption needs a tall, narrow item. Alignment and word wrapping are applied within the swapped rectangle.

- **90**: the text runs from top to bottom (rotated clockwise).
- **270**: the text runs from bottom to top (rotated counterclockwise).



The rotated text is positioned inside the item box only for the angles 90 and 270. For other angles, including 180 and arbitrary angles, the engine does not translate the text into the item rectangle, so the text appears outside the item. Check the result in the preview before using other angles.

15.5.2. Limitations

- Rotation is applied only in the graphical outputs: preview, printer, PDF, SVG and image. The HTML, Excel and text outputs ignore the rotation.
- The automatic height calculation of labels and fields measures the text horizontally and does not take the rotation into account. Use a fixed item size for rotated text.
- The Designer accepts rotation angles from 0 to 360 for labels and fields. If you edit an item whose

definition contains a negative angle (for example one written by hand), the value is reset to 0 when you click **[OK]** in its property dialog.

15.5.3. Examples

The [rotation_demo.xml](#) demo report contains labels rotated by 90, 270 and 180 degrees. The [lbl270.ncr](#) demo report shows a label and a data field rotated by 270 degrees, a typical layout for vertical captions on labels.

Chapter 16. Images

The Image item renders bitmap or vector (SVG) graphics in a report. An image can be stored statically inside the report definition, or it can be loaded at run time from a file, from a URL, from a data source column or from a report parameter. This makes the image item suitable both for fixed content such as company logos and for dynamic, record-dependent content such as product photos.

16.1. Adding an Image Item

In the Designer, select the **Image** tool from the toolbar or the Tools menu, then click in the section where you want to place the item. The new item is created with a default size and its property dialog opens. When no image is loaded (for example, when the image comes from a data source), the Designer displays a hatched placeholder labelled **IMAGE (...)** with the source text in parentheses.

The **Image properties** dialog has the following groups:

Source

The **Item source type** and the source text (file name, data source column, URL or parameter ID, depending on the source type). For static and file sources, a file browser button and a preview button are available.

Format

Image format, horizontal and vertical alignment, **Pin to right**, and the **HTML width** / **HTML height** options.

Scale

The aspect ratio mode and the transformation (smoothing) mode.

Preview

A thumbnail preview of the selected image.

Advanced Options

The common item options: **Print When** expression, HTML options, item ID, function ID and zone ID.

16.2. Image Sources

The source type defines where the image comes from. In the report XML it is stored in the `resource` attribute of the `<image>` element, and the element text holds the source text. If the attribute is missing, the source is static.

Source type (Designer)	resource value	Element text and behavior
Static	<code>(none)</code> or <code>static</code>	The image is embedded in the report definition as Base64-encoded data. When the report is saved, raster images are always written as Base64-encoded PNG. In SVG format, the SVG XML text is embedded instead.
File	<code>file</code>	The path of an image file. The file is loaded when the report definition is read. If the file name contains <code>\$</code> tokens (for example <code>\$D{ds.column}</code> or <code>\$P{param}</code>), the file name is evaluated at run time, when the item is refreshed.
File name from data source	<code>filefromds</code>	A data source column expression (for example <code>col1</code> or <code>ds.filename</code>). The column value is used as the image file name for every record.
Data source	<code>datasource</code>	A data source column that holds the image data itself, for example a BLOB column, a Base64 text column, or an SVG XML text column (depending on the image format).
URL	<code>url</code>	A URL. The image is downloaded through the built-in HTTP client.
URL from data source	<code>urlfromds</code>	A data source column expression whose value is the URL to download for every record.
Parameter	<code>param</code>	The ID of a report parameter that provides the image (see Section 16.2.2).



Images that depend on a data source (data source, file name from data source and URL from data source) are reloaded only when the column value differs from the previous one, so consecutive records that refer to the same image do not load it again.

16.2.1. File names

File names, whether static or coming from a data source, go through the NCReport file name evaluation. The following placeholders are replaced in a file name:

Placeholder	Replaced by
{TEMP} or %TEMP%	The system temporary directory
{HOME} or %HOME%	The user's home directory
{CURR}	The current working directory
{NCREPORT_DIR}	The value of the <code>NCREPORT_DIR</code> environment variable
{ROOT}	The root directory set by <code>NCReport::setRootDir()</code>
{VAR0} ... {VAR9}	The value of the application properties <code>VAR0</code> ... <code>VAR9</code> (set with <code>qApp->setProperty()</code>)

In addition:

- If the file name contains no `{` or `%` characters, the application properties `PATH0` ... `PATH9` are used as search paths: if a file with the same name exists in one of these directories, that file is used.
- A relative file name that does not exist is resolved relative to the directory of the report definition file.
- File names that start with `http` are not modified.

Example of a file name built from two data source columns:

```
<image id="9KF7T" posX="20.637" posY="0.000" width="15.875" height="14.287"
  resource="file" aspectRatio="keep" format="0"
  alignment="1">${d1.filepath}/${d2.filename}</image>
```

16.2.2. Images from parameters

With the **Parameter** source type, the element text is a parameter ID. How the parameter value is interpreted depends on the image format:

- **Native/Binary:** the parameter can hold a `QPixmap` object (a `QImage` if the engine is built with the `nopixmap` build mode) or a string containing an image file name.
- **Base64 encoded:** the parameter holds the Base64-encoded image data as a string.
- **SVG format:** the parameter holds the SVG XML text.

```
NCReport *report = new NCReport(this);
report->setReportFile("invoice.ncr");
report->addParameter("logo", QPixmap(":/images/logo.png"));
```

16.3. Image Formats

The image format tells the engine how to interpret the source data. It is stored in the `format` attribute; the engine accepts both the numeric and the textual form.

Format (Designer)	format value	Description
Native/Binary	0 or <code>bin</code>	Raw image data or an image file in any format supported by Qt's image readers. The Designer's file dialog offers <code>.png</code> , <code>.jpg</code> , <code>.jpeg</code> , <code>.gif</code> , <code>.xpm</code> , <code>.bmp</code> , <code>.pbm</code> , <code>.ppm</code> , <code>.tiff</code> and <code>.svg</code> files.
Base64 encoded	1 or <code>b64</code>	Base64-encoded image data. With a file source, the file must contain Base64 text; with a data source, the column value is decoded from Base64 before loading.
SVG format	2 or <code>svg</code>	SVG XML content, rendered as vector graphics with Qt's SVG renderer.



Use the **Base64 encoded** format when images are stored as text in a database or in an XML/text data source, for example when an image is delivered in a JSON or XML document.

16.3.1. SVG handling

When the **SVG format** is selected, the item keeps the SVG XML text and renders it with `QSvgRenderer` instead of converting it to a bitmap:

- The SVG is scaled from its default size to the item size according to the aspect ratio mode, so it remains sharp in PDF and printer output.
- SVG content can come from a static (embedded) source, a file, a data source column containing SVG XML text, or a parameter.
- In the Designer, the SVG image is drawn in the item only for static and file sources.
- If the engine is built without SVG support (`NCREPORT_NO_SVG`), SVG images are not rendered.



URL sources are always loaded as raster image data, regardless of the selected format.

16.4. Scaling and Transformation

An image is always scaled to fit the item rectangle. The way it is scaled is controlled by two options.

Aspect ratio (`aspectRatio` attribute)

- **Keep aspect ratio** (`keep`, default): the image is scaled as large as possible inside the item rectangle while preserving its aspect ratio.
- **Keep aspect ratio by expanding** (`expand`): the image is scaled to cover the item rectangle while preserving its aspect ratio. Selecting this mode also turns on the item's automatic height (`autoHeight="true"`).
- **Ignore aspect ratio** (`ignore`): the image is stretched to fill the item rectangle exactly.

Transformation (`transformMode` attribute)

- **Off (fast)** (`fast`, default): the image is drawn directly into the target rectangle. This is the fastest method.

- **Smooth transformation (slow)** (`smooth`): the image is first rescaled with smooth (bilinear) filtering and then drawn. This gives better quality, especially when images are scaled down, at the cost of rendering speed.

These options correspond to Qt's `Qt::AspectRatioMode` and `Qt::TransformationMode` values.

16.5. Alignment

When the scaled image is smaller than the item rectangle (which happens with **Keep aspect ratio**), the horizontal and vertical alignment determine where the image is placed inside the rectangle:

- Horizontal alignment: Left, Center, Right
- Vertical alignment: Top, Center, Bottom

The default is left/top. In the XML, the alignment is stored in the `alignment` attribute as the numeric value of the combined `Qt::Alignment` flags, for example `36` (`Qt::AlignHCenter` | `Qt::AlignVCenter`) or `34` (`Qt::AlignRight` | `Qt::AlignVCenter`). The attribute is omitted for the default left/top alignment.

The **Pin to right** option (`pinToRight="true"`) moves the image so that its right edge is aligned to the right page margin.

16.6. HTML Options

HTML width and **HTML height** (`htmlWidth`, `htmlHeight` attributes) are used when the image is converted to HTML in text document print mode. The image is then embedded as an `<img>` tag with Base64 PNG data, and these values become its `width` and `height` attributes. The horizontal alignment determines the alignment of the surrounding block.

16.7. Output Support

Images are painted in all graphical outputs: printer, PDF, SVG, image output and print preview. For HTML-based outputs, the image is converted by the output class itself.

16.8. XML Examples

A static, embedded PNG image (Base64 data shortened):

```
<image id="2" zValue="2" posX="0.000" posY="6.350" width="28.787" height="20.955"
  aspectRatio="keep" transformMode="smooth" format="1"
  alignment="1">iVBORw0KGGoAAAANSUhEUgAA...</image>
```

An image whose file name comes from a data source column, centered in the item:

```
<image id="11771" posX="57.150" posY="0.000" width="55.562" height="11.112"
  resource="filefromds" aspectRatio="keep" format="0"
```

```
alignment="36">col0</image>
```

A Base64-encoded image stored in a data source column:

```
<image id="N7XDB" posX="16.087" posY="0.000" width="6.350" height="6.350"
  resource="datasource" aspectRatio="keep" transformMode="smooth"
  format="1" alignment="1">col0</image>
```

An image downloaded from a URL stored in a data source:

```
<image id="N7Q30" posX="149.225" posY="0.000" width="20.637" height="9.525"
  resource="urlfromds" aspectRatio="keep" format="0">col1</image>
```



The NCReport distribution contains several demo reports for images in the `reports` directory, for example `ImageAlignment.xml`, `ImageAlignmentFromDataSourceFile.xml`, `ImageAlignmentFromDataSourceBase64.xml`, `ImagesByFileNameFromDataSource.ncr`, `ImagesByURLFromDataSource.ncr`, `ImageByComplexFilename.ncr` and `base64image.xml`.

Chapter 17. Barcodes

NCReport renders barcodes with the Barcode item. Barcode generation is based on the Zint barcode library, which is bundled with the NCReport source code (`src/3rdparty/zintlib`, Zint version 2.12.0). Thanks to Zint, the item supports nearly one hundred symbologies, from classic linear barcodes such as EAN-13 and Code 128 to 2D symbols such as QR Code, Data Matrix, PDF417 and Aztec Code.



Barcode support is a build-time feature. It is enabled by the `barcode` extension in `config.pri`, which is part of the default build configuration (`EXTENSIONS += ... barcode ...`). Without it, the barcode item is not able to encode barcodes.

17.1. How Barcodes Are Rendered

When the barcode value or any barcode option changes, the item passes the data to Zint, which produces the barcode in SVG format. By default the SVG is generated in memory. If the engine is built with the `barcodetempfile` extension, Zint writes the SVG into a temporary file instead (into the system temporary directory), which is then read back. This mode is intended for platforms where in-memory generation is not suitable, such as some ARM builds.

The generated SVG is then painted by one of the two available renderers:

Native (SVG) renderer

The default. The SVG is rendered by Qt's `QSvgRenderer`. The barcode is scaled into the item rectangle while keeping its aspect ratio, and positioned according to the item alignment.

Fast mode renderer

An alternative renderer that parses the Zint SVG output and draws its elements directly with `QPainter`.

Barcodes are painted in the printer, PDF, SVG, image and preview outputs. If Zint reports an error (for example, invalid characters for the selected symbology), the Designer shows the error message inside the item rectangle; at run time the barcode is not drawn and the error message is written to the standard error output.

17.2. Adding a Barcode Item

In the Designer, select the **Barcode** tool from the toolbar or the Tools menu and click in the section where you want to place the barcode. A new barcode item is created with the sample value `503102187159` and the default symbology EAN (Zint ID 13), and the **Barcode properties** dialog opens.

The dialog box is titled "Barcode" and is divided into several sections. At the top, there are four input fields: "Item source type" (set to "Data source"), "Barcode type" (set to "EAN (European Article Number)"), "Data source column" (set to "col0"), and "Static barcode data" (set to "50310218715"). Below these is a tabbed interface with "Appearance" and "Options" tabs. The "Appearance" tab contains settings for "Height ratio" (20 X), "Whitespace" (Auto), "Rotation" (0°), "Renderer mode" (Native), "Border type" (No border), "Border width" (0 X), "Foreground color" (Black), and "Background color" (White). There is also a checkbox for "Show barcode text" which is checked. Below the tabs are fields for "Item ID" (2), "Function ID" (empty), and "Zone ID" (0). A checkbox for "Page break after printing this item" is also present. At the bottom right are "OK", "Cancel", and "Apply" buttons.

The dialog contains the following parts:

Barcode

Item source type, barcode type (symbology), the data source column / parameter / expression field, and the static barcode data.

Appearance tab

Height ratio, whitespace, rotation, **Show barcode text**, border type, border width, foreground color and background color.

Options tab

Symbology-specific options (error correction level, version, number of columns, encoding mode, structured primary message, Option1-3) and the renderer mode.

Advanced Options

The common item options: **Print When** expression, HTML options, item ID, function ID and zone ID.

17.3. Setting the Symbology

The symbology is selected in the **Barcode type** combo box, which lists all symbologies supported by the bundled Zint library, sorted by name. In the report definition, the symbology is stored in the `barcodeType` attribute as the numeric Zint symbology ID:

```
<barcode ... barcodeType="58" ...>
```

For backward compatibility, the values `EAN`, `EAN13` and `EANX` are also accepted and mapped to Zint's EAN symbology (ID 13).

The following table lists commonly used symbologies with their Zint IDs:

Zint ID	Symbology (name in the Designer)	Type
3	2 of 5 Interleaved	Linear
8	Code 39	Linear
13	EAN (European Article Number) — EAN-8, EAN-13 and add-ons	Linear
14	EAN + Check Digit	Linear
16	GS1-128	Linear
18	Codabar	Linear
20	Code 128	Linear
25	Code 93	Linear
29	GS1 DataBar Omnidirectional	Linear
34	UPC-A	Linear
37	UPC-E	Linear
40	USPS (U.S. Postal Service) POSTNET	Postal
52	Pharmazentralnummer (PZN)	Linear
55	PDF417	Stacked
57	MaxiCode	2D
58	QR Code	2D
60	Code 128 (Suppress subset C)	Linear
69	ISBN	Linear
70	Royal Mail 4-State Customer Code	Postal
71	Data Matrix (ECC200)	2D
72	EAN-14	Linear
75	NVE-18 (SSCC-18)	Linear
84	MicroPDF417	Stacked
85	USPS Intelligent Mail (OneCode)	Postal
89	ITF-14	Linear
90	Dutch Post KIX Code	Postal
92	Aztec Code	2D
97	Micro QR Code	2D
115	DotCode	2D
116	Han Xin (Chinese Sensible) Code	2D
130	EAN Composite	Composite
145	Rectangular Micro QR Code (rMQR)	2D



The complete list of symbology IDs can be found in the bundled Zint header file `src/3rdparty/zintlib/backend/zint.h` (the `BARCODE_*` constants), and the list of symbologies offered by NCReport in `ZintLib::getAllSymbols()` in

```
src/3rdparty/zintlib/zintlib.cpp.
```

17.4. Barcode Value: Static or Dynamic

The data to encode can be static or dynamic. The **Item source type** combo box offers the following source types for barcodes:

Source type	resource value	Description
Static	<i>(none)</i>	The value typed into Static barcode data . It is stored in the <code>code</code> attribute.
Data source	<code>datasource</code>	The value of a data source column. The element text holds the column expression, for example <code>col0</code> or <code>barcodeDS.col0</code> .
Parameter	<code>param</code>	The value of a report parameter. The element text holds the parameter ID.
Template	<code>tpl</code>	A template text with embedded data source, parameter and variable tokens (for example <code>ORD-#{orders.id}</code>), evaluated at run time.
Script expression	<code>exp</code>	A script expression; its result is the barcode value.

For dynamic sources, the barcode is regenerated only when the value differs from the previous one.



The `code` attribute is always written to the report definition. For dynamic barcodes it simply contains the last value used in the Designer; it is not used at run time once the dynamic value has been evaluated.

17.5. Barcode Options

The following table summarizes the options of the barcode item, the corresponding XML attributes and their defaults. Attributes that hold the default value are not written to the report file.

Option (Designer)	XML attribute	Default	Description
Barcode type	<code>barcodeType</code>	13	Zint symbology ID, see above.
Height ratio	<code>heightRatio</code>	50	The symbol height passed to Zint. A value of 0 is treated as 50. Because the barcode is scaled into the item rectangle with a fixed aspect ratio, this value controls the height-to-width proportion of linear barcodes. (<code>heightFactor</code> is accepted as an older alias when reading.)
Whitespace	<code>whiteSpace</code>	0 (Auto)	Width of the horizontal quiet zone (Zint <code>whitespace_width</code>).
Rotation	<code>rotation</code>	0	Rotation angle in degrees. Zint accepts 0, 90, 180 and 270; other angles result in an "Invalid rotation angle" error.

Option (Designer)	XML attribute	Default	Description
Show barcode text	<code>showText</code>	<code>true</code>	Shows or hides the human-readable text below the barcode.
Border type	<code>borderType</code>	No border	No border, Bind (boundary bars above and below) or Box (box around the symbol).
Border width	<code>borderWidth</code>	<code>0</code>	Width of the border or boundary bars. Written only when a border type is set.
Foreground color	<code>foreColor</code>	black	Color of the bars/modules, as a color name such as <code>#000080</code> .
Background color	<code>backColor</code>	white	Background color of the symbol.
Error correction level (PDF417 and QR Code)	<code>secure</code>	<code>0</code>	If greater than 0, it is passed to Zint as <code>option_1</code> , which is the error correction (security) level for symbologies such as QR Code and PDF417.
Structured primary message (Maxicode and Composite)	<code>primary</code>	<i>(empty)</i>	The primary message used by MaxiCode structured modes and by composite symbologies.
Option1, Option2, Option3	<code>opt1, opt2, opt3</code>	<code>0</code>	Passed directly to Zint's <code>option_1</code> , <code>option_2</code> and <code>option_3</code> symbol fields. Their meaning depends on the symbology; for example, <code>option_2</code> sets the symbol version (1-40) for QR Code and the number of data columns for PDF417. <code>opt1</code> is ignored when the error correction level is set, and <code>opt3</code> is not used for PDF417.
Renderer mode	<code>renderer</code>	<code>0</code> (Native)	<code>0</code> = native SVG renderer, <code>1</code> = fast mode renderer.
<i>(XML only)</i>	<code>alignment</code>	horizontally centered	Position of the scaled barcode inside the item rectangle (numeric <code>Qt::Alignment</code> value). Applies to the native renderer.



The **Version (QR Code)**, **Number of columns (PDF417)** and **Encoding mode** fields of the Options tab are saved to the report (`vers`, `cols` and `mode` attributes), but in the current engine source they are not passed to the Zint encoder. To set the QR Code version or the number of PDF417 columns, use **Option2**; to set the MaxiCode mode, use **Option1** (Zint reads the MaxiCode mode from `option_1`). The `scale` attribute is also read and written, but it does not affect the generated barcode.

For the exact meaning of `option_1`, `option_2` and `option_3` for a given symbology, consult the Zint documentation.

17.6. XML Examples

A static EAN-13 barcode, rotated by 270 degrees:

```
<barcode id="9BUNB" zValue="0" posX="97.790" posY="11.430" width="33.020" height="34.290"
barcodeType="13" code="503102187159" rotation="270"></barcode>
```

A Code 128 barcode whose value comes from a data source column:

```
<barcode id="1" zValue="1" posX="45.720" posY="0.000" width="124.460" height="24.130"
  resource="datasource" barcodeType="60" code="xxxxxx"
  heightRatio="50">col2</barcode>
```

A QR Code with error correction level 2, version 4 (set through Option2), without human-readable text, drawn in dark blue:

```
<barcode id="QR1" zValue="2" posX="10.000" posY="0.000" width="25.000" height="25.000"
  barcodeType="58" code="https://ncreportsoftware.com"
  secure="2" opt2="4" showText="false" foreColor="#000080"
  renderer="0"></barcode>
```



Barcode demo reports are available in the `reports` directory of the NCReport distribution, for example `barcodes.xml`, `barcodes-2.xml`, `barcodes_code128b.xml`, `barcodes_qrcode.xml`, `dynamic_barcodes.xml` and `BarcodeRotate2.ncr`.

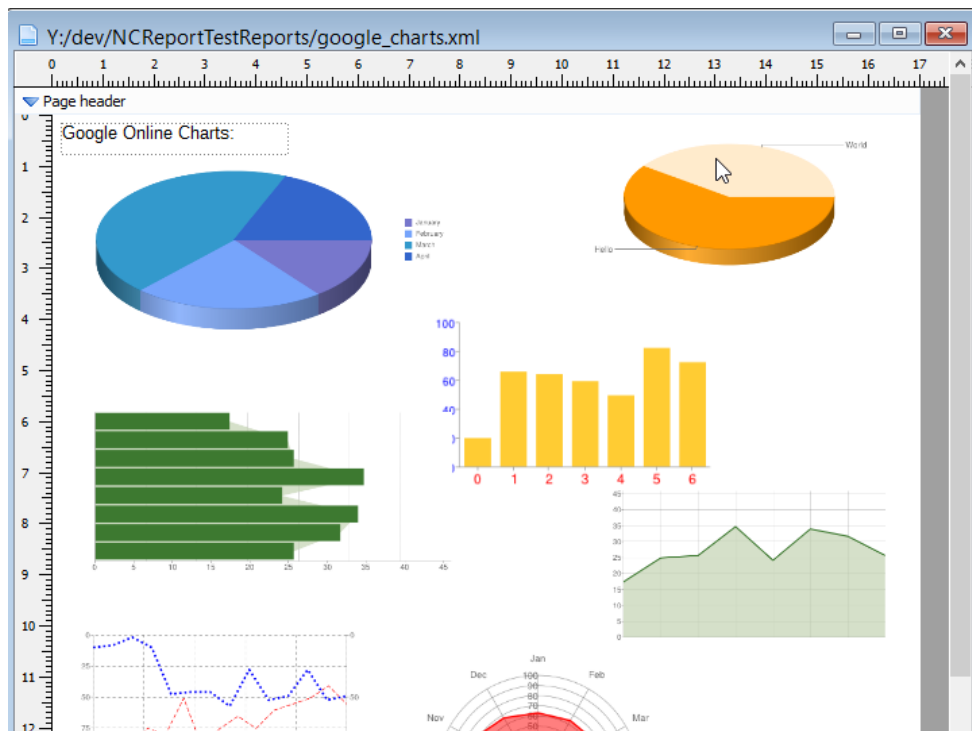
Chapter 18. Charts and Graphics

NCReport has no single built-in "chart" object that covers every need. Instead, it offers several ways to put charts and other graphical content into a report:

- **Online chart images:** an Image item with a URL source can display charts generated by a web chart service (see [Section 18.1](#)).
- **Chart item:** an optional, Highcharts-based chart item that renders a chart from report data through a web engine (see [Section 18.2](#)).
- **Graph (Custom) item:** content painted by your own C++ renderer class, the most flexible solution for charts (see [Section 18.3](#)).
- **Graphics View item:** renders the contents of an existing `QGraphicsView` widget of your application (see [Section 18.4](#)).

18.1. Online Chart Images

Because the Image item can load images from a URL, any web service that returns a chart as a PNG or other image can be used to embed charts in a report. The image is downloaded by the built-in HTTP client when the item is loaded. The chart parameters are encoded in the URL itself. The URL can be static (**URL** source type, `resource="url"`), or it can be read from a data source column (**URL from data source** source type, `resource="urlfromds"`), so the URL, and thus the chart, can be built from report data, for example by an SQL query.



The demo reports `google_charts.xml` and `google_charts2.xml` show this technique with Google's image chart URLs. A static online chart image looks like this:

```
<image id="1" zValue="1" resource="url" posX="96.630" posY="1.534" width="72.089" height="39.879"
```

```
aspectRatio="keep" format="bin">http://chart.apis.google.com/chart?chs=500x200&chd=t:60,40
&cht=p3&chl=Hello|World</image>
```

In `google_charts2.xml` the chart URL comes from a data source column:

```
<image id="0" zValue="0" resource="urlfromds" posX="0.000" posY="7.938" width="168.275" height="87.312"
aspectRatio="keep" transformMode="smooth" format="0">DataSource2.col0</image>
```



Charts loaded this way depend on the availability of the remote service and require network access when the report runs. Remember to escape the `&` characters in URLs as `&` in the XML file.

For all other image options, see the Images chapter.

18.2. Chart Item (Highcharts Integration)

The Chart item (`<chart>` element, implemented by `NCReportChartItem` and `NCReportHighChartsManager`) renders charts with a JavaScript charting library running in a Qt web engine. The item fills an HTML template with data taken from report data sources, loads the result into an off-screen web page, and uses the SVG text produced by the page as the chart image.



The Chart item is an optional feature without official support. It is compiled only when the engine is built with the `highcharts` extension together with the `webkit` or `webengine` extension in `config.pri`. The implementation uses the Qt Script module (`QScriptEngine`), which is not available in Qt 6. The item has no property dialog in the Designer; its settings are edited in the report XML.

18.2.1. How it works

1. The item text contains a JavaScript snippet that defines a global `reportData` object.
2. Series data are filled in from the report data sources (see below), and the updated `reportData` object is serialized back to script.
3. The template file is copied to a result file (by default `chart.html` in the system temporary directory). While copying, the content of the template's `<script id="report-data">` element is replaced with the generated `reportData` script; all other elements and scripts are copied unchanged.
4. The result file is loaded into a `QWebEnginePage` (or `QWebPage` with WebKit). When loading has finished, the plain text content of the page is taken as the chart SVG. The template must therefore produce the chart's SVG markup as the text of the page.
5. The SVG is drawn into the item rectangle, keeping its aspect ratio. If the chart is not ready within 15 seconds, the item reports a "Timeout error when generating chart." error.

The template file name is stored in the `template` attribute. If the file does not exist at the given path, the engine looks for a file with the same name in the current directory, in the report's template

directory, in `templates/` and `../templates/`, and in the `templates` folder of the user's home directory.

18.2.2. Feeding data into the chart

The `reportData.resultSet` object controls how data are loaded. There are two ways:

Data source per series

Each element of `reportData.resultSet.seriesList` may have a `datasource` property and a `categorysource` property in the form `dataSourceID.columnName`. All records of the referenced column are loaded into the series' `data` array (as numbers) or `category` array (as strings).

Dedicated data source

If `reportData.resultSet.datasource` contains a data source ID, the engine builds the whole `seriesList` from that data source. The column names of the data source determine the structure:

- `category`, `data` and `series` columns: one series is created for each distinct `series` value; each record adds a `data` value and a `category` label to its series. An optional `group` (or `groupname`) column sets the series' `group` property.
- An additional `category_key` column: records are sorted by the category key, a first series named `category` holds the category labels, and missing data points are filled with `null`.
- Any other column layout is treated as pivoted data: every data source column becomes a series named after the column, with one data point per record.

A minimal skeleton of the item's `reportData` script, showing the properties read by the engine; the other properties of the object (chart options for the template, for example) are passed to the template unchanged:

```
var reportData = {
  resultSet: {
    seriesList: [
      { datasource: "salesDS.amount", categorysource: "salesDS.month" }
    ]
  }
};
```

18.2.3. Output

In printer and PDF output, the chart SVG is drawn through an intermediate pixmap; in SVG, image and preview output it is rendered directly as SVG.

18.3. Graph (Custom) Item

The Graph item (also called Custom item, `<graph>` element) delegates painting to a C++ class that you implement and register in your application. Your class receives a `QPainter`, the target rectangle and the item's data, and can draw anything: bar or line charts, gauges, diagrams, or other QPainter-drawn content. This is the recommended way to produce charts that are fully under your control and that

print in vector quality.

18.3.1. Item settings

The **Graph/Custom item properties** dialog has the following fields:

Item resource

Static or **Data source**.

Class ID

The ID of the renderer class that paints the item (`classID` attribute).

Datasource column

With the **Data source** resource, the data source column whose value is passed to the renderer for every record.

Graph/Item static definition

With the **Static** resource, a fixed text passed to the renderer.

The text passed to the renderer is the `itemdata` argument of the renderer's `paintItem()` method. Its format is entirely up to you, for example a comma-separated list of values, as in the `graph_demo.xml` demo report:

```
<datasource id="data" type="txt" openRole="begin" source="static" columnDelimiter="tab" hasColumnHeader="true">ID Description GraphData
0 Graph 1. 95,63,84,96,20,34,50
1 Graph 2. 57,33,79,51,80</datasource>
...
<graph id="3" zValue="3" resource="datasource" classID="graph01"
  posX="0.000" posY="6.350" width="169.862" height="28.575">GraphData</graph>
```

Graph items can also store named parameters instead of a single definition text. In this mode the element has the `hasParameters="true"` attribute and `<par>` child elements; string parameter values that contain `$` tokens are evaluated before each rendering. The renderer can read the parameters through its `item()` pointer.

18.3.2. Renderer classes

Renderers are subclasses of `NCReportGraphRenderer`. The class `NCReportAbstractItemRendering`, used in older examples, is an obsolete subclass kept for backward compatibility. The main virtual methods are:

`paintItem(QPainter*, NCReportOutput*, const QRectF& rect, const QString& itemdata)`

Pure virtual. Paints the item content into `rect`.

`paintEditorItem(QPainter*, const QRectF& rect, const QString& text)`

Paints the item in the Designer. If no renderer is registered for the item, the Designer shows a placeholder labelled `CUSTOM DRAWING (...)`.

`calculateSizeMM() / calculateHeightMM()`

Return the item size or height in millimeters when the item's automatic height is enabled.

`dataSourceID()`

Returns the ID of a report data source; the engine then makes that data source available to the renderer through `dataSource()`.

`isValidLocation(int sectionType)`

Restricts the section types in which the item may be placed.

The renderer for an item is looked up first by the item's numeric type ID (`typeId` attribute) and then by its class ID. Set the renderer's ID with `setId()` (a string or an integer) and register it with `NCReport::addGraphItemRenderer()`; the older `addItemRenderingClass()` method is obsolete but still available:

```
NCReport *report = new NCReport(this);
MyChartRenderer *renderer = new MyChartRenderer();
renderer->setId("graph01");           // must match the item's Class ID
report->addGraphItemRenderer(renderer);
```

The renderer is called only for graphical outputs (printer, PDF, SVG, image and preview). A complete renderer implementation is shown in the "Custom items in NCReport" section of the NCReport API chapter; the `GraphDemo`, `SqlGraphDemo` and `CustomItem` demo applications in `src/demos` are working examples.



Applications that embed the Designer can register the same renderers in the Designer main window (`addGraphItemRenderer()`), so the custom content is visible while designing. Custom toolbar actions for graph items with a given type ID can be added with `addCustomToolAction()` or `addGraphItemAction()`, and a custom property dialog can be provided through `NCRDGraphItemDialogFactory`.

18.4. Graphics View Item

The Graphics View item (`NCReportGraphicsViewItem`, `<graphicsview>` element) prints the content of a

`QGraphicsView` widget that exists in your application. This lets you print charts or drawings that are already displayed on screen using the Qt Graphics View framework, without writing a renderer class.

In the Designer, select the **Graphics View** tool and place the item in a section. In the **GraphicsView item settings** dialog, set the **Class ID**, which identifies the view object. In design mode the item is shown as a placeholder labelled `QGraphicsView Content [classID]`.

At run time, register the view pointer under the same ID through the report definition:

```
NCReport *report = new NCReport(this);
report->setReportFile("chartreport.ncr");
report->reportDef()->addGraphicsView("salesChart", ui->graphicsView);
```

```
<graphicsview id="GV1" zValue="0" posX="0.000" posY="0.000" width="120.000" height="80.000"
classID="salesChart"/>
```

When the item is painted, the view is temporarily resized to the size of the item rectangle and rendered with `QGraphicsView::render()`, keeping its aspect ratio; afterwards the original widget size is restored. The item is static: it shows the current content of the view and is painted only in graphical outputs (printer, PDF, SVG, image and preview).

Chapter 19. Zones

This feature is available since version 2.2.0. Zones are virtual bands within a report section. Every report item has a **Zone ID** property. Items with the same zone ID form a horizontal zone, a virtual band inside the section, much like a group. When the section's **Automatic height** option is enabled, the report engine renders the zones one after another, in the order of their zone IDs. If a zone is empty (for example, because the **printWhen** expression of every item in the zone returns false), the zone is not printed and the section shrinks accordingly. This makes it easy to design sections whose parts appear only under certain conditions, without leaving empty gaps in the output.



Zones are evaluated only in sections with the **Automatic height** option turned on. In fixed-height sections, items are printed at their designed positions.

To set the zone ID of a report item, use the **Identify** tab of the item's property dialog. To assign the same zone ID to several items at once, select the items and choose **Edit > Set zone ID....**

Figure 16. Zone ID in property dialog

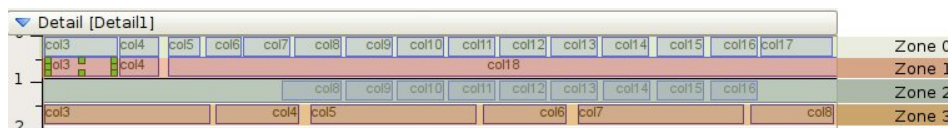


Figure 17. Zones in Design mode



Zones are not visible in design mode. The zone regions are determined only by the zone IDs of the report items.

19.1. Zone settings

Two report-level options influence how zones are rendered. They can be set in the Designer under **Report › Report and page settings...**, in the **Zone settings** panel of the **Sections** tab:

Zone spacing

Vertical spacing, in millimeters, added between the rendered zones. The default is 0. (Saved as `zoneSpacing` in the report definition.)

Page break is allowed between zones

If enabled, a section may break to the next page between its zones, so a section with many zones does not have to fit on one page as a whole. If disabled (the default), the section is printed only if all of its visible zones fit in the remaining space; otherwise, the whole section moves to the next page. (Saved as `zoneBreak` in the report definition.)

19.2. Zones example

To see how zones work, try the `zone_demo.xml` demo report.

Chapter 20. Dynamic data driven size and position

This feature makes it possible to manage the position and size of report items dynamically, driven by a data source, parameter, variable, or even a script expression.

For example, you can store the x and y coordinates of objects in the data source, so that the positions are controlled by the data. If you use graphical items such as rectangles or lines, you can even generate simple bar charts this way, with the bar heights coming from the data.

To set up dynamic position and size management for an item in the Designer, open the item's settings dialog by double-clicking the report item. In the item settings dialog, choose the **Dynamic position** tab in the bottom property panel.

20.1. Dynamic position and size settings

You can set the following properties. If you leave an option empty (this is the default), the setting is turned off and ignored. The values are always measured in millimeters and are relative to the coordinate system of the item's section.

- **X pos** The source of the item's x position.
- **Y pos** The source of the item's y position.
- **Width** The source of the item's width.
- **Height** The source of the item's height.
- **Offset mode** If this option is enabled, the evaluated positions are offsets relative to the item's original (designed) position; otherwise, they are absolute values.

Each property can be a constant number or any expression that results in a number: a data source, parameter, or variable reference such as `$D{x}` or `$P{width}`, or a script expression between `<% %>` markers, for example `<%%$D{h}*2%>`. The expression is evaluated every time the item is rendered.

20.2. Examples

The `DataDrivenPositions.ncr` and `DataDrivenHeights.ncr` demo reports show how data driven positions

and sizes work.

Chapter 21. Dynamic data driven shape style

Shapes such as lines, rectangles, and ellipses can also have a dynamic style. You can control the line width, line color, and fill color dynamically, driven by a data source, parameter, variable, or even a script expression.

To set the dynamic style of a shape item in the Designer, open the item's settings dialog by double-clicking an existing item. In the item settings dialog, choose the **Dynamic style** tab in the bottom property panel.

21.1. Dynamic style settings

You can set the following properties. If you leave an option empty (this is the default), the setting is turned off and ignored.

- **Fill color** The source of the item's background (fill) color.
- **Line color** The source of the item's line color.
- **Line width** The source of the item's line width, measured in millimeters.

Each property can be a constant value or any expression: a data source, parameter, or variable reference such as `$D{color}`, or a script expression between `<% %>` markers. The expression is evaluated every time the item is rendered. The result of a color expression must be a color name that Qt's `QColor` recognizes, for example `#ff0000` or `red`. The result of the line width expression must be a number.

21.2. Example

The `DataDrivenHeights.ncr` demo report uses dynamic fill color and line width values that come from the data source, for example `$D{color}` and `$D{lw}`.

Chapter 22. Page Breaks

Normally, paging is done automatically by the report engine, but there are several ways to force a page break, either statically or based on a logical expression evaluated while the report runs. The options are described below.

22.1. Detail page break condition

A logical expression can be specified in the **Detail settings** dialog (**Report > Details and grouping...**), on the **General** tab, in the **Page break** panel, as the **Break condition (logical expression)**. If the current result of the expression is true, a page break is inserted before the detail section is printed.

22.2. Detail starts on new page

In the same **Page break** panel of the **Detail settings** dialog, the **Starts on new page** option makes the detail start on a new page. This is useful when a report contains more than one detail section and a detail should not continue on the page of the previous one.

22.3. Group page break options

If you define a group for a detail section, there are additional page break options for the group. They can be found in the **Page break** panel of the group settings dialog (**Report > Details and grouping...**, then [**Groups...**]):

Break condition (logical expression)

If the current result of the expression is true, a page break is inserted before the group header section is printed.

Always starts on new page

Each new group (that is, every change of the group expression's value) starts on a new page.

Reset page number

The page numbering restarts from 1 when a new group starts. This option is typically combined with **Always starts on new page**, for example, when every customer's invoice is printed in one report run.

The **Header and Footer** panel of the same dialog also contains the **Reprint header on new page** option. If it is enabled and a group continues on the next page, the group header is printed again at the top of the new page.

22.4. Report item page break

Any report item can cause a page break if you turn on the item's **Page break after printing this item** option. In the Designer, you can find this option on the **Identify** tab of the item's property dialog

(**Report › Edit selected item...**). If this option is turned on, the report engine breaks the page after the item is rendered. Within a section, the report items are rendered in the order in which they appear in the report definition, so you can control which items are printed before the page break.

22.5. Section options related to paging

The following options can be set in the section settings dialog (**Report › Edit selected section...**). Each of them is available only for the section types listed:

Page break after this section

Report header only. If it is turned on, the page breaks after the report header is printed.

Starts on new page

Report footer only. The report footer is printed on a new page.

Break to next page if singular

Group header only. Protects the group header from being left alone at the bottom of a page: if the group header and the following detail do not fit in the remaining space, the group header moves to the next page. This protection takes effect only in double pass mode.

Anchor to page footer

Group footer and report footer only. The section is printed at the bottom of the page, directly above the page footer.

22.6. Batch and sub-report page breaks

In batch mode, a page break is inserted between the reports by default (see [Chapter 34](#)). For sub-report iteration, the **Subreports start on new page** report option controls whether each sub-report starts on a new page (see [Chapter 28](#)).

Chapter 23. Text Document printout mode

The Text Document mode allows you to render and print multi-page rich texts based on HTML and `QTextDocument`. The text source can be a file or any data source, so the text can be static, dynamic, or even a static template filled with dynamic data. In this mode, the report engine collects the HTML content of the report items into one text document, which is then broken into pages by the Qt rich text engine. The result is a multi-page HTML text, correctly broken into pages and ready for printing. In Text Document mode, the `pagecount` system variable can be used without any additional setting, because the number of pages is known before the pages are rendered.

23.1. Steps of usage

To create a text document printout report in the Designer:

- Select **Report > Report and page settings....** In the dialog that appears, set the **Report type** combo box on the **General** tab to **Text document**.
- Add a Text item to the Detail section. Set the text's properties in its property dialog. The text may come from any source, as usual.
- Specify the report's page header and footer, if needed. They are printed on every page.

23.2. How the document is built

- The texts of the report header, the report footer, and the detail sections are joined into one document, in this order. If a detail has a data source, its content is added once for every data record.
- Besides Text items, Image items are also included in the document. Other item types are ignored in the detail sections.
- The print-when expressions of the sections and items are evaluated as usual.
- The horizontal position and the width of the first Text item (if they differ from the defaults) determine the left margin and the width of the printed text block. The height of the text block is the page height minus the margins, the page header, and the page footer.



The text of a Text item can also come from your application: set the source type of the item to **Internal document** and pass your `QTextDocument` object to the report with the `NCReport::setTextDocument()` function before running the report.

23.3. Text Document printout report example

To see how it works, try the `textdocument_printout.xml` demo report. It prints a long Qt class documentation HTML file.

Chapter 24. Label and Multi-Column Reports

Printing labels, such as address labels, product labels or barcode stickers on pre-cut label sheets, is a common reporting task. In a label sheet, the same layout is repeated in a grid: several labels side by side in columns, and several rows on each page. NCReport supports this with a dedicated report type, the **label** report.

24.1. Report Types

Every report definition has a report type, stored in the `type` attribute of the root `<NCReport>` element. The possible values are:

Report type (Designer)	type value	Description
Report	<code>report</code>	The default, normal paged report with headers, footers, details and groups.
Text document	<code>textdoc</code>	Text document print mode (see the Text Document printout mode chapter).
Label	<code>label</code>	Label sheet mode: the detail section is printed repeatedly in a grid of labels.

In the Designer, the report type is selected in **Report > Report and page settings**, on the **General** tab, with the **Report type** combo box.

The screenshot shows the 'Report and page settings' dialog box with the 'General' tab selected. The 'Report type' dropdown is set to 'Report'. Other settings include Report name: 'Untitled', File encoding: 'utf8', Background color: 'white', Resolution: 'Auto', and Default Font: 'Arial'. The 'Working mode' section has 'Data trimming' checked.

24.2. Label Settings

The label grid is defined on the **Label** tab of the **Report and page settings** dialog. The settings are stored in the `<options>` element of the report definition:

Setting (Designer)	XML element	Description
Direction	<code>labelDirection</code>	The order in which labels are filled: horizontal or vertical (see Section 24.4.1).
Label size: Width	<code>labelWidth</code>	Width of one label in millimeters.
Label size: Height	<code>labelHeight</code>	Height of one label in millimeters.
Gaps: Horizontal gap	<code>labelHGap</code>	Space between two adjacent labels in a row, in millimeters.
Gaps: Vertical gap	<code>labelVGap</code>	Space between two rows of labels, in millimeters.
Dimension: Number of rows	<code>labelRowCount</code>	Number of label rows on one page.
Dimension: Number of columns	<code>labelColumnCount</code>	Number of label columns on one page.

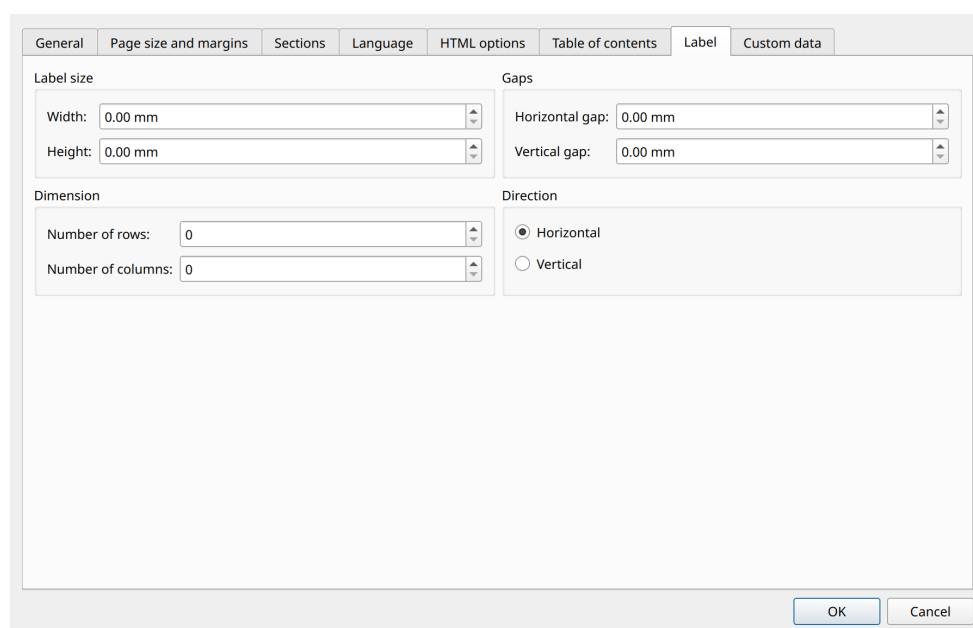


Figure 18. The Label tab of the Report and page settings dialog

The position of the first label is determined by the top and left page margins (**Page size and margins** tab), so the margins must match the top and left edges of the label sheet.

When you close the dialog with **Label** report type selected, the Designer checks the settings:

- The label width and height must be greater than zero.
- The number of label rows and columns must be set.
- Header and footer sections (page header, page footer, report header, report footer) are not allowed in label mode.

24.3. Designing the Label

A label report uses a single detail section, which represents one label. In label mode, the Designer displays the detail section with the size of one label (label width x label height), so you can design the label exactly as it will appear on the sheet. Assign a data source to the detail section as in any other report: each data source record produces one label. Any report items can be placed on the label, such

as fields, text, images and barcodes.

24.4. How Labels Are Generated

In label mode the report engine works differently from normal report processing:

1. Only the first detail section is processed, and it must have a data source assigned.
2. Rendering starts at the top-left corner of the printable area (left margin, top margin).
3. For each data source record, the detail section is rendered once at the current label position, then the position moves to the next label of the grid.
4. When all labels of the page (rows x columns) have been used and there are more records, a new page is started and the grid is filled again from the first label.
5. Processing stops after the last record.

24.4.1. Fill direction

Horizontal

Labels are filled row by row: from left to right across all columns, then the next row starts at the left margin. The horizontal step is label width + horizontal gap, the vertical step is label height + vertical gap.

Vertical

Labels are filled column by column: from top to bottom down all rows, then the next column starts at the top margin.

24.5. Planning a Label Sheet

The label settings can usually be taken directly from the specification of the label sheet. Make sure that the grid fits on the page:

```
left margin + columns x label width + (columns - 1) x horizontal gap <= page width
top margin + rows x label height + (rows - 1) x vertical gap <= page height
```

For example, an A4 sheet (210 x 297 mm) with 3 x 7 labels of 63.5 x 38.1 mm, 2.5 mm horizontal gap and no vertical gap needs a left margin of 7.25 mm and a top margin of 15.15 mm: $7.25 + 3 \times 63.5 + 2 \times 2.5 = 202.75$ mm (leaving 7.25 mm on the right) and $15.15 + 7 \times 38.1 = 281.85$ mm (leaving 15.15 mm at the bottom).

The corresponding report options:

```
<NCReport version="2.32.1" name="Address labels" type="label">
  <options>
    <orientation>portrait</orientation>
    <pagesize width="210.000" height="297.000">A4</pagesize>
    <topmargin>15.15</topmargin>
```

```

<bottommargin>15.15</bottommargin>
<leftmargin>7.25</leftmargin>
<rightmargin>7.25</rightmargin>
...
<labelColumnCount>3</labelColumnCount>
<labelRowCount>7</labelRowCount>
<labelHGap>2.5</labelHGap>
<labelVGap>0</labelVGap>
<labelWidth>63.5</labelWidth>
<labelHeight>38.1</labelHeight>
<labelDirection>horizontal</labelDirection>
</options>
...
<details>
  <detail id="Detail1" height="38.100" datasource="addresses">
    <items>
      ...
    </items>
  </detail>
</details>
</NCReport>

```



The demo report `label_test.xml` in the `reports` directory is a label report that prints a frame, a title and a data-driven EAN barcode on every label.

The label settings can also be set from code through the report definition's page options, for example:

```

NCReportPageOption opt = report->reportDef()->option();
opt.setReportType(NCReport::Label);
opt.setLabelSize(QSizeF(63.5, 38.1));
opt.setLabelColumnCount(3);
opt.setLabelRowCount(7);
opt.setLabelHorizontalGap(2.5);
opt.setLabelVerticalGap(0);
opt.setLabelDirection(NCReport::Horizontal);

```

24.6. Multi-Column Reports

The report options also contain two settings intended for multi-column reports:

columns

Number of columns.

columnspace

Space between columns in millimeters.

These values are read from and written to the report definition (`NCReportPageOption::columnCount()` and `columnSpacing()`), and they appear in every report file, usually as `<columns>0</columns>` and `<columnspace>0</columnspace>`.



In the current version of the report engine, the `columns` and `columnspace` options are not used during report generation, and the Designer has no input fields for them. Setting

them does not make the details flow into multiple columns.

To print detail records side by side in several columns, use the **label** report type: define the "column" as a label with the desired width and height, set the number of label columns and rows, and choose the fill direction. **Vertical** direction fills each column from top to bottom before moving to the next column, which corresponds to the newspaper-style flow of a classic multi-column report; **horizontal** direction produces a card or catalog style layout. Keep in mind that label mode prints only the detail section: header and footer sections and groups are not processed.



The `group_demo_2columns.ncr` demo report is not a multi-column layout example: it demonstrates grouping by an expression built from two data columns.

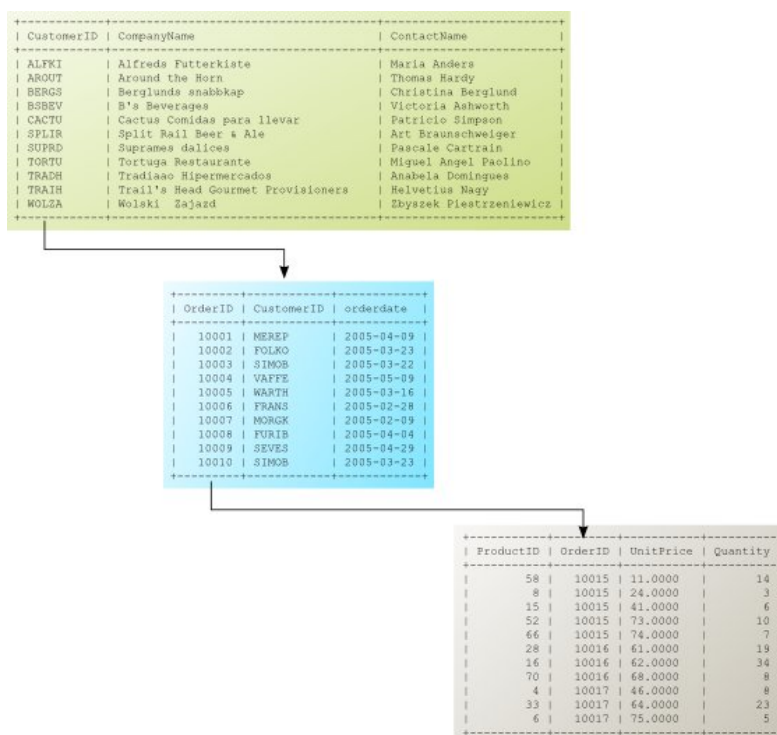
Chapter 25. Data Relation System

This feature is also known as the sub-query or sub-data source system. Databases almost always contain master/detail relations between tables. When you define reports for typical documents such as invoices or orders, at least one header record and its related detail data are used, linked by a primary key and a foreign key. The goal of the data relation system is to update child data sources at runtime, row by row, driven by a parent data source. This works through an ID column that is the primary key of the parent and the foreign key of the child. Data source relations are very useful for SQL data sources, where the data is fetched from database tables with an SQL command, and for Item Model data sources, where you manage the data source content from code.



The data source relation system currently works only for the SQL data source and the Item Model data source. The other data source types are not supported by this feature.

The following example shows a three-level parent/child structure.



The next sections show how to define the data sources of a master/detail relation, using a three-level data source relation as an example.

25.1. Defining a parent data source

You can add the master data source in the data source settings dialog. In the Designer, select **Report > Data sources...** and add a new SQL data source. Set the **Opening/running role** to **Beginning of the report**. This means that the query is executed only once, at the beginning of the report. Type the data source ID, set the connection properties, and edit the SQL query in the SQL editor text box.

This is the master query of our Northwind database example (data source ID: `customers`), which queries the customers:

```
SELECT customers.CustomerID, customers.CompanyName
FROM orders
INNER JOIN customers ON orders.CustomerID=customers.CustomerID
WHERE OrderDate between '2005-03-01' and '2005-03-31'
GROUP BY CustomerID
```

25.2. Defining child data sources

In the same dialog, add two more data sources to the parent/child structure. For the first one, add a new SQL data source again and set its **Opening/running role** to **Child datasource (subquery)**. This means that the query is executed repeatedly, every time the next master record is processed. Type the data source ID (`orders`) and set the connection properties. Then type the **Parent datasource** ID, which is the ID of the previously defined parent data source (`customers`).

Warning

The **Parent datasource** ID is case sensitive. It must exactly match the ID of an existing parent data source.

Edit the SQL query in the SQL editor text box. This is the first child query. It queries the order headers of a customer within a date range:

```
SELECT OrderID, CustomerID, EmployeeID, OrderDate, ShipName
FROM orders
WHERE CustomerID='$D{customers.CustomerID}'
AND OrderDate between '2005-03-01' and '2005-03-31'
ORDER BY OrderID
```

As you can see, the data relation is managed by a data reference expression: `$D{customers.CustomerID}`, which inserts the key value of the parent data source into the SQL command.

Next comes the second child data source (`items`), the third level of the relation. Set its **Opening/running role** to **Child datasource (subquery)** as well, and type the **Parent datasource** ID, which is the ID of its parent data source (`orders`). Edit the SQL query in the SQL editor text box. This query retrieves the order items of the current order:

```
SELECT OrderID, orderitems.UnitPrice, Quantity, Itemno,
products.productname, orderitems.UnitPrice*Quantity as Value
FROM orderitems INNER JOIN products ON orderitems.productID = products.productID
WHERE OrderID=$D{orders.OrderID}
ORDER BY Itemno
```

At this level, the data relation is managed by the `$D{orders.OrderID}` data reference expression. The parent key is always evaluated, and the query is executed whenever the parent key changes, that is, whenever the report processor moves to another parent row.

25.3. Setting up the detail section

In this step, assign the appropriate data source to the Detail section. To do that, select **Report > Details and grouping...** (or use the tool button on the toolbar) to open the **Detail settings** dialog. Select the root parent data source of the hierarchy, which in our example is `customers`.



Since version 2.13, the root parent data source must always be assigned to the Detail section. (In earlier versions, the lowest-level child data source had to be assigned; see "Changes in version 2.13" below.)

25.4. Designing the report

After defining the data sources and assigning the root data source to the Detail, add the appropriate groups to the Detail as well, using the **[Groups...]** button. As usual, each data source level corresponds to a group level.

Add the other report sections and report items and set their alignment. The following figure shows the ready-to-run report. The figures show the `list_of_orders_complex.xml` example report, which was made with the pre-2.13 relation system and can now be found in the `reports/obsolete` directory. The `DataSourceRelationTest.xml` demo report uses the same data sources with the current relation system.

The screenshot displays the NCRReport Designer interface for a report titled "List of orders". The report structure is as follows:

- Page header:** Contains the title "List of orders" and a "datetime" field.
- Group header [Detail1.G_OUT]:** Contains the text "Outmost Group Header" and a "tg.xdata" field.
- Group header [Detail1.Group_customer]:** Contains the text "customers.Custo" and "customers.CompanyName".
- Group header [Detail1.Group_order]:** Contains a table with columns: "ItemNo", "Product name", "Quantity", "Unit price", and "Total". The table also includes fields for "Order ID: orders.Or", "Order Date: ers.OrderDate", and "Quantity".
- Detail [Detail1]:** Contains a table with columns: "ItemNo", "ProductName", "Quantity", "UnitPrice", and "Value".
- Group footer [Detail1.Group_order]:** Contains the text "Order total:" and "OrderTotal".
- Group footer [Detail1.Group_customer]:** Contains the text "customers.CustomerID" and "CustomerTotal".
- Group footer [Detail1.G_OUT]:** Contains the text "Outmost Group Footer" and a "tg.xdata" field.
- Page footer:** Contains the text "appinfo" and a "pagenc" field.

Figure 19. Sub-query report example in Designer

The preview of the example report looks like this:

List of orders 2011-04-30T21:25:15

*This report demonstrates the **subquery** usage. In other words it's named also "**child query**" which means the often applied parent-child data **relation**.*

Outmost Group Header ALFKI

ALFKI **Alfreds Futterkiste**

ALFKI / Germany Order ID: **10692** Order Date: **2005-03-31**

Itemno	Product name	Quantity	Unit price	Total
1	Singaporean Hokkien Fried Mee	43	63,00	2 709,00
				Order total: 2 709,00

ALFKI / Germany Order ID: **10835** Order Date: **2005-03-03**

Itemno	Product name	Quantity	Unit price	Total
1	Queso Manchego La Pastora	55	59,00	3 245,00
2	Escargots de Bourgogne	13	77,00	1 001,00
				Order total: 4 246,00

ALFKI total: 6 955,00

ANTON **Antonio Moreno Taqueraa**

ANTON / Mexico Order ID: **10507** Order Date: **2005-03-11**

Itemno	Product name	Quantity	Unit price	Total
1	Rassle Sauerkraut	46	43,00	1 978,00
2	Laughing Lumberjack Lager	12	48,00	576,00
				Order total: 2 554,00

ANTON / Mexico Order ID: **10677** Order Date: **2005-03-26**

Itemno	Product name	Quantity	Unit price	Total
1	Boston Crab Meat	31	26,00	806,00
2	Rassle Sauerkraut	2	33,00	66,00
				Order total: 872,00

ANTON / Mexico Order ID: **10856** Order Date: **2005-03-16**

Itemno	Product name	Quantity	Unit price	Total
1	Iphoh Coffee	19	2,00	38,00
2	Perth	14	42,00	588,00
				Order total: 626,00

ANTON total: 4 052,00

Figure 20. Result of a sub-query report example

25.5. Changes in version 2.13

The data source relation feature was extended in version 2.13 and became a better supported, general feature. It now also works with Item Models. Reports created with the old sub-query/relation system are not compatible with the new version of the data source relation function.



Reports that use the sub-query function and were created with an earlier version of NCReport must be upgraded.

Changes in the function (make these changes in old reports):

- The detail's data source must be the root parent data source.
- All fields and expressions must include their data source ID, i.e. `datasource.column`.

To use the new data source relation system, follow these rules:

- A data source relation is defined simply by setting the **Child datasource (subquery)** role and specifying the parent data source ID (as usual).

- One parent can have only one child (a one-to-many relation).
- You can specify the **Primary key column index** of the parent data source. If a primary key column is defined for the parent data source, you can use the `{PK}` or `{ID}` placeholder in the child data source query; it is replaced with the current key value of the parent. (This works in SQL data sources only.)
- When you assign a data source relation to a detail section, always set the root parent data source on the detail. In earlier versions, the last child data source had to be set, but this is outdated since 2.13.
- For non-SQL data sources, handle the data source updates with the `dataSourceUpdateRequestFKey(const QString& dataSourceID, const QString& foreignKeyValue)` signal of the `NCReport` object, which passes the current foreign key value. (The `dataSourceUpdateRequest(const QString& dataSourceID, const QString& data)` signal passes the evaluated data source specification instead.)
- Use `!$D{datasource.isEmpty()}` as the print-when expression of a detail section to hide empty child data.

Chapter 26. Double pass mode

Double pass mode is a report option that changes how the report engine runs. When double pass mode is enabled, the report is processed twice. The two runs are called the primary (test) pass and the secondary (real) pass. In the test pass, the report is rendered to an invisible "null" output, so nothing is printed, but the engine learns the final layout, including the total number of pages. The real pass then renders the report to the actual output, already knowing the results of the test pass.

Why is double pass mode necessary? In normal (single pass) mode, the report engine simply runs the report from beginning to end without any look-ahead, so information such as the total number of pages is not available while the pages are being rendered.



The `pagecount` system variable always returns zero in single pass (normal) mode. If you need the `pagecount` system variable, you have to enable double pass mode.

Double pass mode has the following additional effects:

- The **Break to next page if singular** option of group headers takes effect only in double pass mode (see [Chapter 22](#)).
- With endless paper, the height of the output is adjusted to the final position measured in the test pass.

Since the report is processed twice, double pass mode increases the running time of the report. Enable it only when you need it. Double pass mode applies to normal reports only; in Text Document mode, the page count is available anyway.

26.1. Setting double pass mode

The double pass option is one of the report options that are saved in the report definition (as the `doublePass` option). To enable or disable it in the Designer, select **Report > Report and page settings...** and turn on the **Double pass mode** option in the **Working mode** panel of the **General** tab. For more information, see "Setting up page and report options" in the Designer part.

26.2. Example using the pagecount variable

Use the `$V{pagecount}` expression in a field in expression or template mode, or in a text item in template mode.

Expression mode example: `$V{pagecount}`

Template mode example: `Page $V{pageno} of $V{pagecount}`

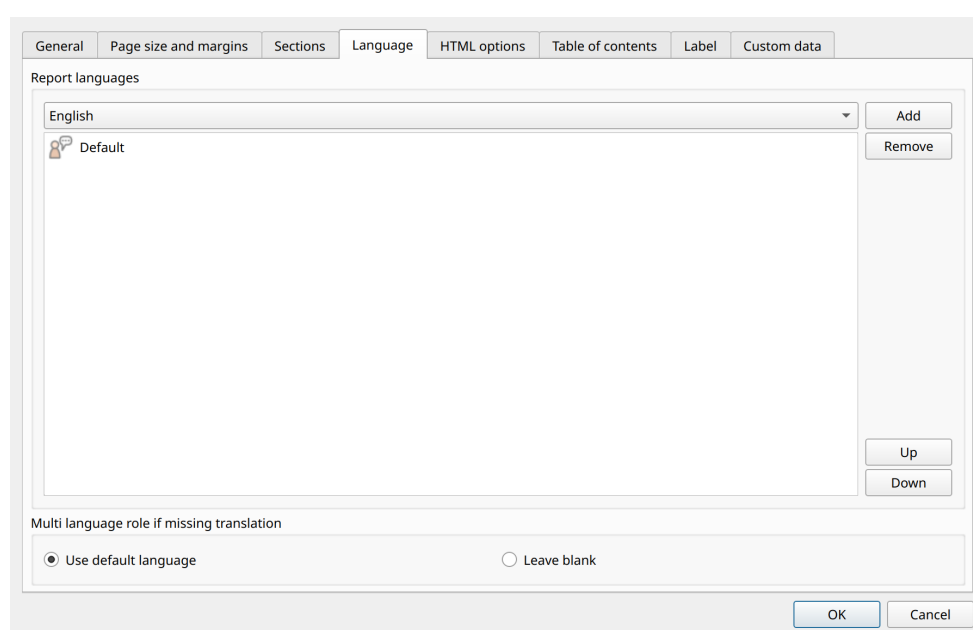
The `sqlite_productlist_grouped_doublepass.xml` demo report uses double pass mode.

Chapter 27. Internationalization

Since version 2.5, reports can support multiple languages. This is an important aspect of international applications. The goal of this feature is to allow fields and labels to store multiple texts, one for each predefined language.

27.1. Adding languages

- To set the languages, select **Report > Report and page settings...** and choose the **Language** tab in the dialog.
- To add more languages, select a language in the combo box and add it to the language list with the **[Add]** button.



Leave the **Default** language first in the list. It represents the original language of the report.

- Set the **Multi language role if missing translation** option. It determines what happens if a label or field has no translation for the current language: if you choose **Use default language**, the default text appears; if you choose **Leave blank**, the label or field is not printed.

27.2. Adding translations of Fields or Labels

Insert a Field or Label item as usual. The property dialog contains a tab for each language defined previously. Type the translations into the appropriate language tabs. An empty translation tab means a missing translation.

The screenshot shows the NCReport Designer interface. At the top, there are tabs for 'Label [en]', 'Label [de]', and 'Label [hu]'. Below these is a text box containing the word 'Invoice'. The main panel has two tabs: 'General' and 'Text'. The 'Text' tab is active, showing settings for 'Background mode' (set to 'Transparent'), 'Background color' (set to 'White'), and 'Elide mode' (set to 'Off'). Below these are 'Options' with checkboxes for 'Word wrapping', 'Word wrap anywhere', 'Automatic height', 'Background only in Designer', 'Clickable in Preview', 'Pin to left', and 'Pin to right'. At the bottom, there are tabs for 'Identify', 'Print when', 'HTML', 'Dynamic position', 'Dynamic style', and 'TOC'. The 'Identify' tab is active, showing 'Item ID' (1), 'Function ID' (empty), and 'Zone ID' (0). There is also a checkbox for 'Page break after printing this item' and buttons for 'Conditional format...', 'OK', and 'Cancel'.

27.3. Setting up the current language

The current language of the report can be set both in design mode and at runtime. In the Designer, select **Report > Report language** or the **[Languages]** tool button on the toolbar, and choose the language you want from the submenu.

To set the language from application code, use the `setCurrentLanguage( const QString& langcode )` function, where **langcode** is the two-letter ISO language code.

27.4. Setting up the language

```
NCReport* report = new NCReport(parent);
report->setCurrentLanguage("de");
```

To set the language when running reports from the command line, use the `-l` (or `--language`) command line option followed by the two-letter language code.

27.5. Setting up the current language from command line

```
NCReport -f myreport.ncr -l de
```

The `multi-language-invoice-test.xml` demo report shows a multi-language report.

Chapter 28. Sub-Report iteration

The **Sub-Report** feature is a repeated report generation process that iterates over a dedicated data source, whose opening role is **Subreport iteration**. This approach is similar to the classic sub-report model, but it supports only one level. It is particularly useful when a complex report or a multi-detail report needs to be repeated for different data records. The dedicated (master) data source is the source of the repetition: the whole report, including the report header and footer, is processed once for every record of this data source.

The Sub-Report function is a convenient way to create simple one-to-many relationship reports. Note that the other data sources must not be set as **child data sources** of the master data source; the report uses the sub-report data source to repeat the reports.

28.1. Sub-Report data source

To define the data source on which the iteration is based, add a data source to the report as usual and set its **Opening/running role** to **Subreport iteration**.

28.2. Reference to master data source

You can place any reference to the master data source in the SQL queries of the other data sources. For example:

```
SELECT product.name, product.code FROM product WHERE id=SD{master.id}
```

For non-SQL data sources, such as the Item Model data source, you can use the signal/slot mechanism. Use the following signal of the NCReport object to update the data of your data source:

```
signals:
    void dataSourceUpdateRequest(const QString& dataSourceID, const QString& data);
```



All data sources except the master data source (and the data sources with the **When group change** role) are updated at the beginning of every sub-report cycle. This function is similar to a parent/child relation, but not the same.

28.3. Sub-Report options

The following report options, found in **Report > Report and page settings...** on the **General** tab in the **Working mode** panel, affect sub-report iteration:

Subreports start on new page

Each sub-report cycle starts on a new page.

Subreport resets page number

The page numbering restarts from 1 for each sub-report cycle.

The `subreport.xml` and `SubreportOrders.ncr` demo reports show the sub-report iteration in practice.

Chapter 29. Table View Rendering

The Table View item is a report item for rendering `QTableView` tables with full WYSIWYG print support, or for rendering the content of an item model. The function was originally designed for printing tables that look the same as an existing `QTableView` widget on the screen. The Table View item follows the formatting of the original `QTableView` widget, and the cells get their display appearance information from the table's item model. Some basic table settings, such as the header background, line type, and line color, are currently fixed.

29.1. Adding TableView item

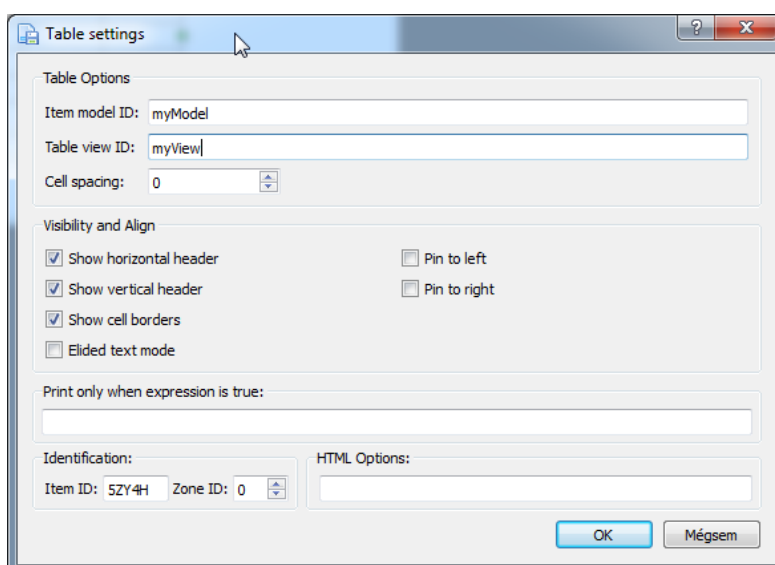
To add a Table View item to a section in the Designer, select the **[Table View]** tool button or the corresponding item in the **Insert** menu. After the cursor changes to a crosshair, click in the section where you want to place the item. The **Table View** item is created and its settings dialog appears.



It is strongly recommended to add the **Table View** to a Detail section, not to a header or footer. Since the table may fill the available space both horizontally and vertically, no other report items should be added to this section. The Table View has its own data assignment, so it is independent of the detail's data source. The detail's data source should be a dedicated one with a single record; a one-row static text data source is a good choice.

In the **Table settings** dialog, specify the same IDs that you use when passing the table view and the model to the report from code. The report engine identifies the objects by these IDs.

29.2. Table View Dialog



The dialog options are as follows:

- **Table view ID** Identifies the `QTableView` object you want to render.
- **Item model ID** Identifies the item model of the table.

- **Cell spacing** Spacing value for the cells. Currently it has no effect.
- **Cell renderer ID** and **Cell renderer column** Assign a custom cell renderer to a column (see "Custom Cell Content" below).
- **Decimal precision** The number of decimal places for floating-point data cells. If it is set to **Auto**, the values are printed unformatted.
- **Font size** The font size of the table, or **Auto**.
- **Force aspect ratio** If enabled, the table is always scaled to the width of the item.
- **Show horizontal header** If enabled, the horizontal table header appears.
- **Show vertical header** If enabled, the vertical table header appears.
- **Alternate row background** If enabled, every second row gets a light gray background.
- **Auto row height** Automatic row height calculation.
- **Show cell borders** Shows or hides the cell borders.
- **Elided text mode** When this option is enabled, multi-line text is not rendered in full; only the first line is shown, followed by an ellipsis.
- **Pin to left** The table is automatically aligned to the left margin.
- **Pin to right** The table is automatically aligned to the right margin.
- **Hidden rows** and **Hidden columns** Comma-separated lists of the row and column indexes that should not be printed.

29.3. Setting the object references

Use the following API functions to pass the QTableView object and its model to NCReport. You have to set the appropriate IDs to identify the objects, because multiple objects can be assigned to one NCReport object. This is needed only when running the report, not at design time.

```
NCReport* report = new NCReport( this );
report->addItemView( ui->tableView, "myView");
report->addItemModel(ui->tableView->model(), "myModel");
```

29.4. Example

The following example shows a QTableView widget on the screen and the same table in the print preview. The table is filled with test data, including images.

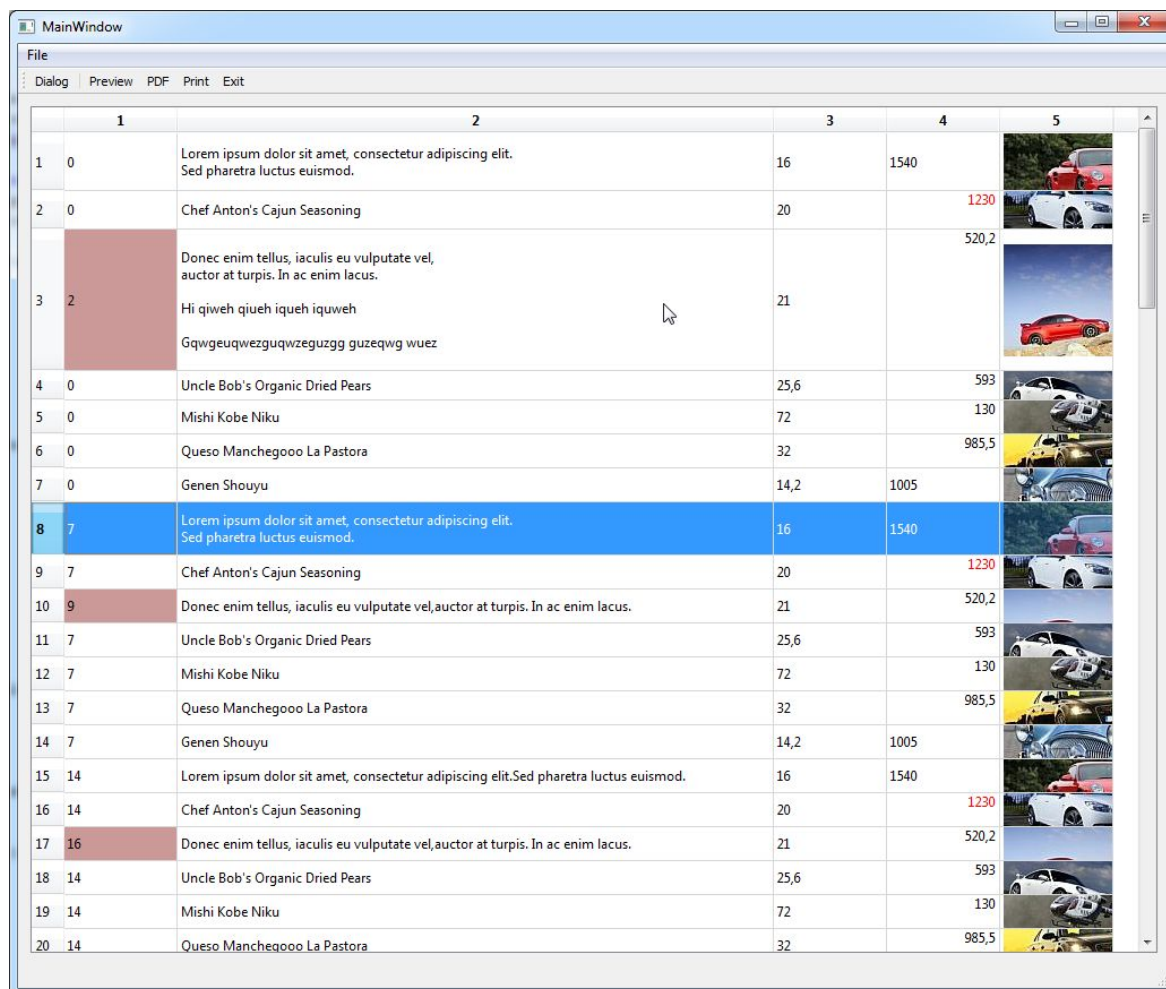


Figure 21. `QTableView` widget

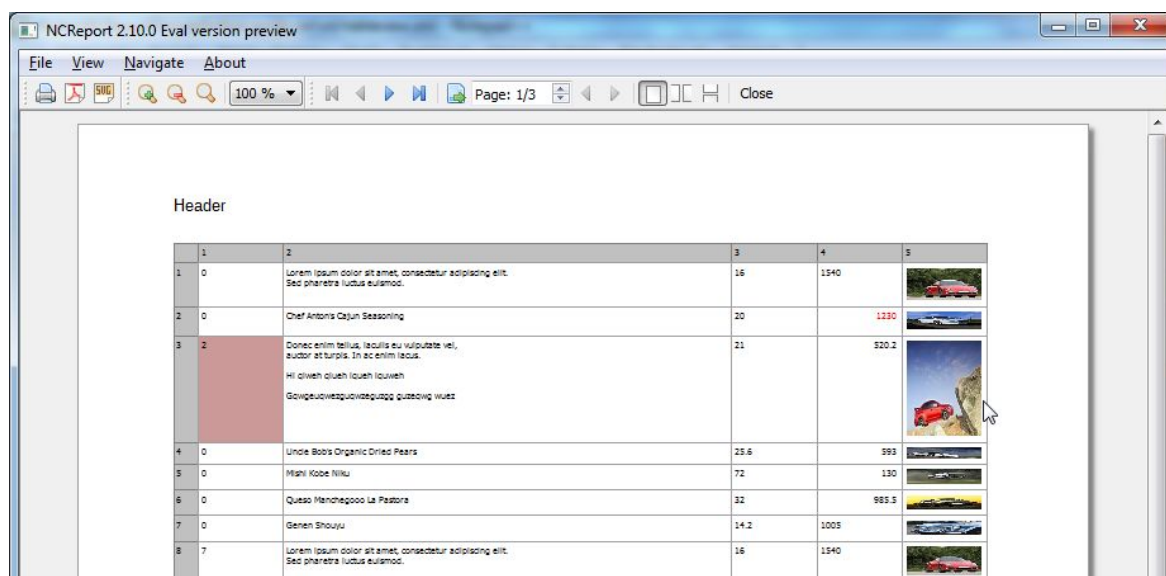


Figure 22. `QTableView` table in print preview

29.5. Printing Item Model Based Table without QTableView

It is also possible to render a table without having an existing `QTableView` widget (or any other item view). In this case, you have to set at least the item model ID that you assign from code; the table view

ID can be left empty. Without an item view, the report engine tries to calculate the optimal cell width and height based on the content.

Example code:

```
QStandardItemModel model(4, 4);
for (int row = 0; row < model.rowCount(); ++row) {
    for (int column = 0; column < model.columnCount(); ++column) {
        QStandardItem *item = new QStandardItem(QString("row %0, column %1").arg(row).arg(column));
        model.setItem(row, column, item);
    }
}
NCReport* report = new NCReport( this );
report->setReportFile("mytablereport.ncr");
report->addItemModel(&model, "myModel");
report->runReportToPdfWriter("outputfile.pdf");
```

29.6. Custom Cell Content

It is even possible to print custom-defined cells (columns) in tables, for example graphical content. To achieve this, implement a subclass of the graph renderer API, whose base class is `NCReportGraphRenderer`. The renderer is assigned to a column of the table by the **Cell renderer ID** and **Cell renderer column** settings of the table item (internally, these are set by the `NCReportTableItem::setCellGraphRenderer(const QString id, int column)` function). When a data cell of that column is painted, the renderer's `paintItem()` function is called, and the `index()` function of the renderer returns the model index of the current cell.

Example implementation of a custom cell/column renderer class:

```
#include "ncreportgraphrenderer.h"

class MyCustomCellRenderer : public NCReportGraphRenderer
{
public:
    MyCustomCellRenderer();
    virtual ~MyCustomCellRenderer() override;

    virtual void paintItem( QPainter* painter, NCReportOutput* output, const QRectF& rect, const QString& itemdata ) override;
};

#include "mycustomcellrenderer.h"
#include <QPainter>

MyCustomCellRenderer::MyCustomCellRenderer() {}

MyCustomCellRenderer::~MyCustomCellRenderer() {}

void MyCustomCellRenderer::paintItem(QPainter *painter, NCReportOutput *output, const QRectF &rect, const QString &itemdata)
{
    Q_UNUSED(output)
    Q_UNUSED(itemdata)

    MyLogoPainter cp;
    cp.setMyProperty(50);
```

```
cp.setState(MyLogoPainter::Basic);

painter->save();
cp.paint(painter, rect.toRect());
painter->restore();
}
```

To register your renderer with the report, use the `addGraphItemRenderer()` API function:

```
MyCustomCellRenderer *renderer = new MyCustomCellRenderer();
renderer->setId("mycell");
...
NCReport* report = new NCReport( this );
report->addGraphItemRenderer(renderer);
```

In the table settings dialog, set the same renderer ID and the column index:



29.7. Handle progress signal of table rendering

You can connect the progress signal of the table rendering to a slot in your application. This can be useful when printing large tables. The signal is emitted row by row:

```
NCReport::dataRowProgress(int row)
```

Chapter 30. Cross-Tab Tables

Reports often contain tables or data presented in a tabular layout. Sometimes it is necessary to rotate the results so that the data source columns are displayed as rows and the data source records as columns. This is known as creating a pivot table, a cross-tab report, or rotating data. In cross-tab tables, the data source records are represented as horizontal columns, and the cross-tab rows are printed as data source columns. Tables often include horizontal and/or vertical summarization as well.

	Jan	Feb	Mar	Apr	May	June	Total
Income	19,80	23,30	35,70	43,90	28,70	30,50	181,90
Expense	20,10	19,80	18,50	18,60	19,60	21,20	117,80
Assets	10,00	8,00	16,00	17,00	17,30	14,10	82,40
Liability	47,30	36,60	54,10	31,80	42,90	53,20	265,90
Total	97,20	87,70	124,30	111,30	108,50	119,00	648,00

A cross-tab table has its own data source: a dedicated data source must be defined in the report for each table. When the report generator renders cross-tab tables, they behave as follows:

- Horizontally expandable: Table columns represent data source records. If the table is wider than the available space, it continues in a new table below.
- Vertically expandable: Each row represents a data column of the specified data source, and the table can break across multiple pages.

30.1. Table Structure

Cross-tab tables are composed of cells, each with its own function depending on its location. The primary elements of tables are the rows and columns. The following figures show the cross-tab row and column structure along with the names of their functions:

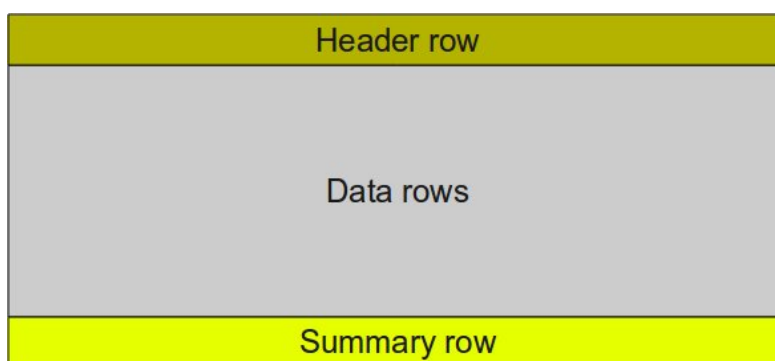


Figure 23. Table Rows



Figure 24. Table Columns

0	1	1	1	2
3	4	4	4	5
3	4	4	4	5
3	4	4	4	5
3	4	4	4	5
3	4	4	4	5
6	7	7	7	8

Figure 25. Cell Structure

The cell structure of cross-tab tables is as follows:

Cell Function	Description
0	Corner header
1	Column header
2	Side summary header
3	Row header
4	Data
5	Side summary data
6	Bottom summary header
7	Bottom summary data
8	Cross summary data

30.2. Using Cross-Tab Tables in the Designer

To add a cross-tab table to a report, select the Cross table item on the toolbar or in the **Insert** menu.

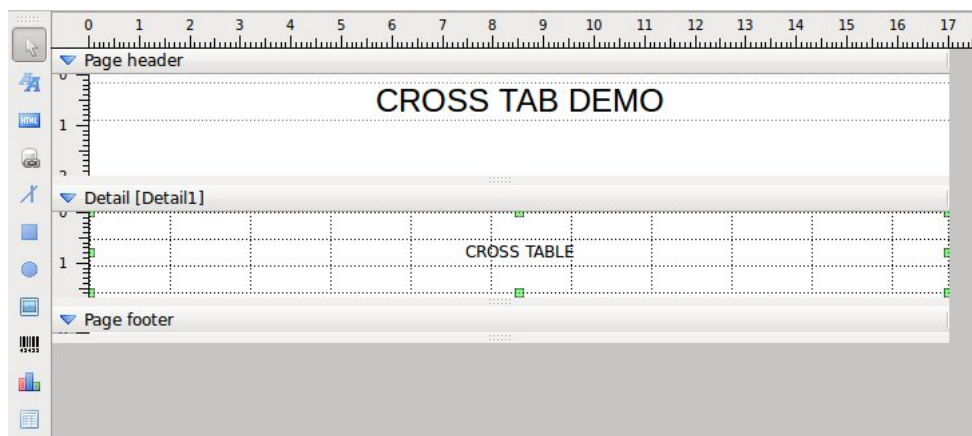


Figure 26. Cross-tab in Designer

This creates a new cross-tab object in the selected section at the clicked position and opens the **Cross tab settings** dialog. In this dialog, you can set all of the table's properties.

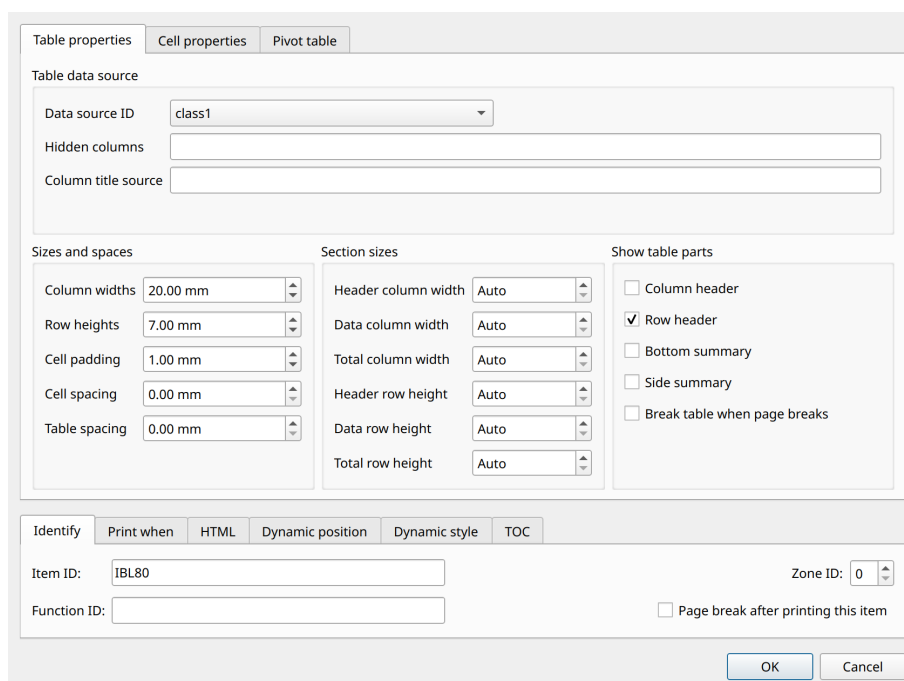


Figure 27. Cross-tab settings dialog

The dialog is divided into the following tabs: **Table properties**, **Cell properties**, and **Pivot table**. The **Print only when expression is true** option is at the bottom of the dialog. If a logical expression is defined, the table is shown or hidden based on the result of the expression.

30.2.1. Table Level Properties

- Table data source:
 - Data source ID: ID of the data source of the table. The selected data source should be unique and independent of the data source of any detail, because cross-tab tables have their own data processing.
 - Hidden columns: Comma-separated list of data source columns that should not be shown in

the table.

- Column title source: The data column that contains the column header titles. If it is not specified, the column numbers appear.
- Sizes and spaces: General sizes of cross-tab table elements.
 - Column widths: General width of the columns (or **Auto**)
 - Row heights: General height of the table rows (or **Auto**)
 - Cell padding: Gap inside the cells (internal cell margin)
 - Cell spacing: Spacing between the cells
 - Table spacing: Spacing between the tables when the table breaks into a new line
- Section sizes: Sizes of cross-tab table sections.
 - Header column width: Width of the header (left, first) column
 - Data column width: Width of the data columns
 - Total column width: Width of the total (summary) column, usually the last, rightmost column
 - Header row height: Height of the header (first) row
 - Data row height: Height of the data rows
 - Total row height: Height of the total (bottom summary) row, usually the last row of the table
- Show table parts: Switches to enable or disable specified table parts.
 - Column header: Shows or hides the column header
 - Row header: Shows or hides the row header
 - Bottom summary: Shows or hides the summary row
 - Side summary: Shows or hides the side summary column
 - Break table when page breaks: If enabled, the table can break between its rows at a page break

30.2.2. Cell Level Properties

The cell properties apply to the cells of the selected function (see the cell structure above).

The screenshot shows the 'Pivot table' tab in the NCReport properties dialog. The 'Borders' sub-tab is active, displaying settings for the four sides of the cell. All borders are checked as 'Enabled', set to 'Solid' style, '0.10 mm' width, and '#000000' color. Below this, the 'Identify' sub-tab is visible, showing 'Item ID' as 'IBL80' and 'Function ID' as an empty text box. A 'Page break after printing this item' checkbox is located to the right of the 'Function ID' field. 'OK' and 'Cancel' buttons are at the bottom right.

The following groups of settings are available for each cell function:

- **Data:** The data type of the cell (Text, Numeric, Date, DateTime, or Boolean), a static cell text, and a format arg string.
- **Style:** Font, font size, text color, rotation, and bold, italic, and underline switches.
- **Align:** Horizontal and vertical alignment.
- **Background:** Background mode (transparent or opaque), fill pattern, and background color.
- **Number format:** Formatted number, localized formatting, blank if zero, format code, field width, fill character, and precision. These options work like the numeric format options of fields (see the "Data Formatting" chapter).
- **Borders:** Top, right, bottom, and left borders of the cell.

30.2.3. Pivot Table Properties

On the **Pivot table** tab, the cross-tab item can be switched to pivot table mode with the **Create Pivot table** option. In this mode, the table is not a simple rotation of the data source: the values are aggregated by two data columns, one of which provides the rows and the other the columns of the table.

- **Row source:** The data column whose distinct values form the rows of the pivot table.
- **Column source:** The data column whose distinct values form the columns of the pivot table.
- **Value source:** The data column that contains the values to be aggregated.
- **Aggregate function:** Count, Summary, Average, Standard deviation, Minimum, or Maximum.
- **Auto sort header data:** Sorts the row and column header values in ascending order (compared as text).

The `PivotTableDemo.xml` and `PivotTableDemo2.xml` demo reports show the pivot table mode in practice. For cross-tab examples, see the `crosstabtest*.xml` and `crosstab_school.xml` demo reports.

Chapter 31. Conditional Formatting

This feature enables dynamic, data-driven text styles in reports, based on the current value of any data source column, parameter, variable, or script expression. Conditional formatting is available for Labels and Fields only. (HTML Text items can be styled dynamically in a different way: by embedding data references or script expressions in the HTML code in template mode.)

The format definition is a block of code with style tags and values or expressions, similar to CSS code. A style tag and its value/expression are separated by a colon, and each line contains one style definition. A value can be a constant, a data source, parameter, or variable reference, or a script expression. Script expressions must be enclosed in curly braces: if the value starts with `{`, it is evaluated as a script.

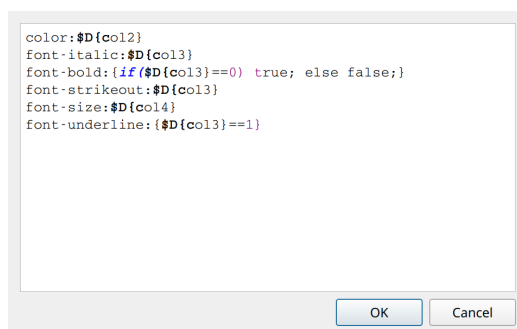
31.1. Dynamic Style Tag Symbols

Tag symbol	Description	Examples
<code>color:</code>	Text foreground color	<code>color:#ff0000, color:\${D}{ds.color}, color:{if(\${D}{ds.price}>500) "#ff0000"; else ""};</code>
<code>background-color:</code>	Text background color	<code>background-color:#ff0000, background-color:\${D}{ds.bgcolor}</code>
<code>font-family:</code>	Font family name	<code>font-family:Arial, font-family:\${D}{ds.font}</code>
<code>font-bold:</code>	Font bold on/off	<code>font-bold:true, font-bold:\${D}{ds.isBold}</code>
<code>font-italic:</code>	Font italic on/off	<code>font-italic:true, font-italic:\${D}{ds.isItalic}</code>
<code>font-weight:</code>	Font weight integer value. Higher value results in bolder text.	<code>font-weight:50, font-weight:\${D}{ds.fweight}</code>
<code>font-underline:</code>	Font underline on/off	<code>font-underline:true, font-underline:\${D}{ds.isUnderline}</code>
<code>font-size:</code>	Font size in points. Integer value.	<code>font-size:12, font-size:\${D}{ds.size}</code>
<code>font-strikeout:</code>	Font strikeout on/off	<code>font-strikeout:true, font-strikeout:\${D}{ds.fstrikeout}</code>
<code>letter-spacing:</code>	Text letter spacing value. A greater value results in wider spacing.	<code>letter-spacing:1.5, letter-spacing:\${D}{ds.letterspacing}</code>
<code>capitalization:</code>	Rendering option that the text font applies. Integer value from 0 to 4. Equals the QFont::Capitalization enumeration property.	<code>capitalization:\${D}{ds.cap}</code>

31.2. Editing Style Code in Designer

To define conditional text formatting for a Label or a Field, click the **[Conditional formatting...]** button at the bottom of the item property dialog. The conditional format code dialog appears. Type or

paste the format code, following the syntax rules, then click [**OK**] to save the code.



NOTE

A style tag and its value must be on one line. A style definition split across multiple lines is not evaluated. Since the first colon separates the tag from its value, the value itself must not contain a colon.

31.3. Default Style

If a condition (script or data) returns an empty value, the default style is applied. The default style settings are the ones you set statically in the report, as usual.

31.4. Examples

The `CondFormatDemo.xml`, `CondFormatDemo_JS.xml`, and `CondFormatDemo2.xml` demo reports show conditional formatting in practice.

Conditional format code can also modify the value of variables. See [Section 12.4](#).

Chapter 32. Report Outputs

The report engine renders the report onto an *output*. The output decides what the result is: pages in a preview window, a printed document, a PDF file, image or SVG files, an HTML page, an Excel compatible HTML file or a text file built from a template. The same report definition can be run to any of these outputs without changes.

Every output is a subclass of the abstract `NCReportOutput` class. You can either use one of the one-step `NCReport::runReportTo...`() functions, which create, configure and delete the right output object for you, or create an output object yourself, assign it with `NCReport::setOutput()` and start the report with `NCReport::runReport()`.

32.1. Output types

The following table lists the output classes of the engine and the matching one-step functions of the `NCReport` class.

Output	Class	One-step function
Preview	<code>NCReportPreviewOutput</code>	<code>runReportToPreview()</code> , <code>runReportToShowPreview()</code>
Preview dialog (Qt / modern)	<code>NCReportQtPreviewOutput</code>	<code>runReportToQtPreview()</code> , <code>runReportToModernPreview()</code>
Printer	<code>NCReportPrinterOutput</code> (<code>NCReportPrinterOutputV21</code>)	<code>runReportToPrinter()</code> , <code>runReportToPrinterNoDialog()</code>
PDF (QPrinter based)	<code>NCReportPdfOutput</code>	<code>runReportToPdf()</code>
PDF (QPdfWriter based)	<code>NCReportPdfWriterOutput</code>	<code>runReportToPdfWriter()</code>
SVG	<code>NCReportSvgOutput</code>	<code>runReportToSVG()</code>
Image	<code>NCReportImageOutput</code>	<code>runReportToImage()</code>
HTML	<code>NCReportHtmlOutput</code>	<code>runReportToHtml()</code> (or <code>runReportToHTML()</code>)
Excel (HTML based)	<code>NCReportExcelHtmlOutput</code>	<code>runReportToExcelHtml()</code>
Text (template based)	<code>NCReportTextOutput</code>	<code>runReportToText()</code>

The `NCReportOutput::OutputType` enumeration also contains values such as `PostScript`, `Xml`, `Odf`, `ExcelXml`, `PlainText` and `Custom`. Of these, only `PostScript` has an implementation, and only in builds with Qt 4 (see [Section 32.7](#)). The `Null` type is used internally by `NCReportNullOutput` for the first pass of double pass reports.



The `runReportTo...`() functions do not return a result. After they return, check the result with `NCReport::hasError()` and `NCReport::lastErrorMsg()`.

32.2. Common output options

The `NCReportOutput` base class provides the following options for every output type.

setFileName(const QString&)

The name of the output file for file based outputs.

setPageRanges(const QString&)

Restricts the output to the given pages. The expression is a comma-separated list of page numbers and ranges, for example `1-3,5,8-10`. When the report is run with `NCReport::runReport()`, the page range expression of the `NCReport` object is always passed to the output and replaces the output's own setting, so set the page ranges with `NCReport::setPageRanges()`.

setPageRange(int from, int to), setPageFrom(int), setPageTo(int)

Sets the page range by its first and last page.

setCopies(int)

The number of copies (printer output).

setShowPrintDialog(bool)

Whether the print dialog appears before printing (printer output).

setParentWidget(QWidget*)

The parent widget of dialogs opened by the output.

setAutoDelete(bool)

Whether the output object is deleted by the `NCReport` object. A preview output that is handed to a preview window must not be auto-deleted.

setResolution(int dpi)

The resolution of outputs that support it (printer, PDF).

setRotationAngle(int)

Rotates the output. `NCReport::setRotationAngle()` passes the angle to the outputs created by the one-step functions.

setBatchModePageBreakBetweenPages(bool)

Controls the page break between the reports of a batch. `NCReport` has a function with the same name.

hasError(), lastError()

The error state of the output.

32.3. Running the report with an output object

The one-step functions cover the most common needs. When you need more control, create the output object, configure it and run the report with `runReport()`:

```
NCReport *report = new NCReport();
report->setReportFile("products.ncr");
```

```

NCReportOutput *output = new NCReportPdfWriterOutput("products.pdf");
report->setOutput(output);           ①
report->setPageRanges("1-3");       ②

bool ok = report->runReport();
if (!ok || report->hasError())
    qWarning() << report->lastErrorMsg();

```

- ① The previous output object is deleted.
- ② The page ranges are applied to the output by `runReport()`.

32.4. Preview output

`NCReportPreviewOutput` records the rendered pages in memory. It does not display anything by itself: after the report has run, the output object is passed to an `NCReportPreviewWindow`, which shows the pages and lets the user print them or save them to other formats.

`runReportToPreview()` runs the report into a new preview output (with auto delete disabled), and `runReportToShowPreview( QWidget* parent = nullptr, bool maximized = false )` does the same and then opens the preview window:

```

// one step: run and show the preview window
report->runReportToShowPreview(this);

// or step by step
report->runReportToPreview();
if (!report->hasError()) {
    NCReportPreviewWindow *pv = new NCReportPreviewWindow();
    pv->setReport(report);
    pv->setOutput(static_cast<NCReportPreviewOutput*>(report->output()));
    pv->setAttribute(Qt::WA_DeleteOnClose);
    pv->exec();
}

```

Setting the report object with `setReport()` makes printing from the preview window run the report again to the printer, instead of printing the lower-quality preview pages. See the *Previewing the report* section of the *Using the NCReport API* chapter for details.

32.5. Preview dialogs: Qt print preview and modern preview

`NCReportQtPreviewOutput` is a printer based output that renders the report into a preview dialog.

`runReportToQtPreview( QWidget* parent )`

Shows Qt's standard `QPrintPreviewDialog` (extended with a [**Close**] button).

`runReportToModernPreview( int dpi, const QString& outputFileName, QWidget* parent )`

Shows NCReport's own preview dialog. If `dpi` is 0, the print resolution of the report is used.

Both functions open the dialog modally and delete the output when it is closed.

```
report->runReportToQtPreview(this);  
// or  
report->runReportToModernPreview(0, "products", this);
```

32.6. Printer output

`NCReportPrinterOutput` prints the report through a `QPrinter`. The printer is created with a print quality (`NCReportPrinter::PrinterMode: ScreenResolution`, `PrinterResolution` or `HighResolution`) and an optional resolution. If the print dialog is enabled, it is shown in `initialize()`, and cancelling it stops the report.

`printer()`

Returns the `QPrinter` object, so you can, for example, set the printer name or the document name.

`setCopies( int )`

Sets the number of copies on the printer.

`customizePrinter( QPrinter* )` signal

Emitted while the printer is initialized, so the application can change printer settings. When the one-step functions are used, the signal is forwarded by `NCReport::customizePrinter()`.

The one-step functions are:

```
// copies, show dialog, parent widget, printer name, document name  
report->runReportToPrinter(1, true, this);  
  
// print to a named printer without dialog  
report->runReportToPrinterNoDialog(2, "HP_LaserJet", "Invoice");
```

The print quality used by `runReportToPrinter()` and `runReportToPdf()` can be set with `NCReport::setPrintQuality()`.

`NCReportPrinterOutputV21` is the printer output of NCReport version 2.21, kept for compatibility. It has the same interface. When the library is built with the `p21` build mode (which defines `NCREPORT_PRINTER_V21`), `runReportToPrinter()` uses this class instead of `NCReportPrinterOutput`.

32.7. PDF output

There are two PDF outputs, built on two different Qt classes.

`NCReportPdfOutput`

A subclass of the printer output that prints into a `QPrinter` set to PDF format. It uses the print quality and resolution of the report, like the printer output. Its run function is `runReportToPdf(const QString& filename, bool overwrite = true, PdfVersion version = Pdf1_4, const QString& creator = QString())`. In builds with Qt 4 the same class can also produce PostScript (`runReportToPostScript()`); with Qt 5 and later `runReportToPostScript()` does nothing.

NCReportPdfWriterOutput

Renders directly into a `QPdfWriter` (Qt 5 and later). It can write to a file or to any `QIODevice` (for example a `QBuffer` in memory), and it supports more document settings: `setDpi( int )` (the default of the class is 120 dpi, the default of `runReportToPdfWriter()` is 600 dpi), `setPdfVersion( NCReport::PdfVersion )`, `setCreator( const QString& )` (default: NCReport), `setTitle( const QString& )` (default: the report name) and `setXmpMetaData( const QByteArray& )` (applied with Qt 5.15 or later). This is the recommended way of PDF generation; the Designer also uses it with Qt 5 and later.

The PDF version (`NCReport::PdfVersion`) can be `Pdf1_4` (default), `PdfA1b` or `Pdf1_6`. When `overwrite` is `false` and the file already exists, or when the file is not writable, the report is not run and an error is set.

```
// QPdfWriter based PDF, 300 dpi, PDF/A-1b
report->runReportToPdfWriter("report.pdf", 300, true, NCReport::PdfA1b,
                             "My Application", "Monthly report");

// PDF into memory
QBuffer buffer;
buffer.open(QIODevice::WriteOnly);
report->runReportToPdfWriter(&buffer);

// QPrinter based PDF
report->runReportToPdf("report.pdf");
```

32.8. SVG output

`NCReportSvgOutput` writes every page into a separate SVG file at 96 dpi. The page files are named after the given file name with a page suffix: running to `report.svg` produces `report_p1.svg`, `report_p2.svg`, and so on. The SVG title is the report name.

```
report->runReportToSVG("/tmp/report.svg");
```

32.9. Image output

`NCReportImageOutput` renders every page into a `QImage` and saves the pages as image files. The image format is determined by the extension of the file name (for example `.png` or `.jpg`), and the pages are saved as `<name>_p1.<ext>`, `<name>_p2.<ext>`, and so on.

When the report is in endless (continuous) mode, the output produces a single image instead, cut to the height of the rendered content and saved with the given file name. `setMaximumHeight( int )` sets the maximum height of this image in pixels (default: 20000). The rendered images can also be accessed from code with `page( int )` and, in endless mode, `successiveResult()`.

```
report->runReportToImage("/tmp/report.png"); ①
```

① The pages are saved as `report_p1.png`, `report_p2.png`, and so on.

32.10. Markup language outputs: HTML and Excel

`NCReportMarkupLanguageOutput` is the common base class of the text based markup outputs. It collects the items of every rendered section, arranges them into table rows and cells, and writes the result as one document, in the encoding of the report. Subclasses only have to implement how the document, the sections, the rows and the cells are written (`writeHeader()`, `writeFooter()`, `writeSectionStart()`, `writeRowStart()`, `cellStarts()` and so on), which makes it a good starting point for custom markup outputs.

Its options are:

setStrategy(SpreadStrategy)

How sections are converted to tables. `SectionOneTable` (default): each section is translated as one table. `SectionMultiTables`: sections are always separated into multiple tables.

setStyleSheetFile(const QString&)

If empty (default), the generated CSS styles are embedded in the document. Otherwise the styles are written to the given CSS file and the document links to it. The special value `@` means a CSS file with the same base name as the output file.

32.10.1. HTML output

`NCReportHtmlOutput` generates an HTML file. Images, barcodes and charts are embedded as PNG images. With `setImageMode( NCReportHtmlOutput::File )` they are saved into a `<filename>_files` directory next to the HTML file instead of being embedded as Base64 data (`NCReportHtmlOutput::Base64`, the default).

The HTML document can be customized in the report definition:

htmlBodyStyle

Page option (*Page settings* dialog, *HTML options*, *Body style options*). CSS declarations for the `body` element. The marker `{DEFAULT}` (the default value) inserts the width, default font and margins of the report page.

htmlHead

Page option (*General head options*). Text inserted into the `<head>` element, for example additional `<meta>` or `<link>` tags.

htmlOptions

Item and section property. Additional CSS declarations for the table cell of the item or for the section.

HTML page options in the report definition

```
<options>
...
<htmlBodyStyle>{DEFAULT}background-color:#f0f0f0;</htmlBodyStyle>
<htmlHead>&lt;meta name="author" content="NCReport"&gt;</htmlHead>
```

```
</options>
```

`runReportToHtml( const QString& filename, int strategy = 0, const QString& cssFile = QString() )` runs the report to HTML. `strategy` 0 means `SectionOneTable`, any other value `SectionMultiTables`.

```
report->runReportToHtml("report.html");           ①
report->runReportToHtml("report.html", 1, "report.css"); ②

// or with an output object
NCReportHtmlOutput *html = new NCReportHtmlOutput();
html->setFileName("report.html");
html->setStrategy(NCReportHtmlOutput::SectionMultiTables);
html->setImageMode(NCReportHtmlOutput::File);
report->setOutput(html);
report->runReport();
```

- ① Styles are embedded in the HTML file.
- ② Multiple tables (`SectionMultiTables`), styles in an external CSS file.

32.10.2. Excel HTML output

`NCReportExcelHtmlOutput` is a variant of the HTML output that produces HTML markup in the format Microsoft Excel opens as a worksheet (`application/vnd.ms-excel` content type). All pages are stored in one file; the Designer offers the `.xls` extension for it.

```
report->runReportToExcelHtml("report.xls");
```

32.11. Text output

`NCReportTextOutput` generates any kind of text file (plain text, CSV, HTML, XML, ...) from a text template, without graphical rendering. The template file defines the text of each report section. The template syntax is described in the *General TEXT output* chapter.

```
report->runReportToText("customers.csv", "templates/customers_csv.txt");

// or with an output object
NCReportTextOutput *txt = new NCReportTextOutput();
txt->setFileName("customers.csv");
txt->setTemplateFile("templates/customers_csv.txt"); // or setTemplate( templateText )
report->setOutput(txt);
report->runReport();
```

32.12. Custom outputs

To create your own output, subclass `NCReportOutput` and implement the pure virtual functions `initialize()`, `cleanup()`, `begin()`, `end()` and `newPage()`. Items are painted with the painter returned by `painter()`; for special rendering, `renderItem()`, `renderSection()`, `sectionStarts()` and `sectionEnds()` can be reimplemented. If `initialize()` returns `false`, the report is not run.

32.13. Selecting the output in the Designer

In the Designer, the **Run report** dialog lists the available outputs: *New preview*, *Old preview*, *Qt print preview*, *Printer*, *PDF file*, *SVG file(s)*, *Image file(s)*, *HTML file*, *Excel file* and *Text file*. For file based outputs the Designer asks for the output file name. The dialog also has a page range field and a *Run reports in batch mode* option for running all opened reports as a batch.

The outputs map to the engine as follows: *New preview* runs `runReportToModernPreview()`, *Old preview* `runReportToShowPreview()`, *PDF file* `runReportToPdfWriter()` (with Qt 5 and later), *HTML file* `runReportToHTML()` and so on.

Some output settings are taken from the Designer preferences:

Printer

Printer DPI (used by the new preview), *Default printer name* and *Print without printer dialog*.

HTML Output

Sections build from one table / Sections build from multiple tables (the HTML strategy) and *CSS file*.

32.14. Selecting the output in the command line tool

The command line tool selects the output with the `-o (--output)` option. Its values are `preview` (default), `print`, `pdf`, `svg`, `html`, `image` and `excel`. File based outputs need the output file name in `-of (--output-file)`.

Option	Meaning
<code>-o, --output</code>	Output type: <code>preview</code> , <code>print</code> , <code>pdf</code> , <code>svg</code> , <code>html</code> , <code>image</code> , <code>excel</code>
<code>-of, --output-file</code>	Output file name
<code>-n, --printer</code>	Name of the target printer
<code>--copies</code>	Number of copies for the printer output
<code>--nodialog</code>	Print without the printer dialog
<code>-hs, --htmlstrategy</code>	HTML strategy: <code>1</code> = one table per section (default), any other value = multiple tables
<code>-css, --css-file</code>	External CSS file for the HTML output

```
NCReport -f products.ncr -o pdf -of products.pdf
NCReport -f products.ncr -o html -of products.html -hs 2 -css products.css
NCReport -f products.ncr -o print -n HP_LaserJet --copies 2 --nodialog
```

See the *Command line client* chapter for all options.

Chapter 33. General Text Output

Text output is a very powerful feature of NCReport. It can generate various kinds of text output, such as HTML, XML, or plain text. Text output requires an additional text template, so besides the report definition, you also have to specify the template file before running the report.



Text output is generated very quickly, because the data is substituted directly into the text template without any graphical rendering.

33.1. Text template manager tags

The following tags are available when you create a text template. Each pair of begin and end tags delimits the template of a report section. The tags are written as standard HTML comments, and each tag must be at the beginning of a separate line. The text between the tags may contain data source, parameter, variable, and field references (`{D{}}`, `{P{}}`, `{V{}}`, `{F{}}`), which are replaced with their current values every time the section is processed.

33.2. Text template tags

Tag keyword	Description
<code><!-- BEGIN {DH} --></code>	Document header begins. The document is the text output itself; the document header is written once, at the beginning of the output (for example, the HTML head part).
<code><!-- END --></code>	Section ends
<code><!-- BEGIN {DF} --></code>	Document footer begins. It is written once, at the end of the output (for example, the closing part of an HTML document).
<code><!-- END --></code>	Section ends
<code><!-- BEGIN {PH} --></code>	Page header section begins.
<code><!-- END --></code>	Section ends
<code><!-- BEGIN {PF} --></code>	Page footer section begins.
<code><!-- END --></code>	Section ends
<code><!-- BEGIN {RH} --></code>	Report header section begins.
<code><!-- END --></code>	Section ends
<code><!-- BEGIN {RF} --></code>	Report footer section begins.
<code><!-- END --></code>	Section ends
<code><!-- BEGIN {D.DetailID} --></code>	Detail section begins. The section is identified by the detail ID.
<code><!-- END --></code>	Section ends

<code><!-- BEGIN {GH.DetailID.GroupID} --></code>	Group header section begins. The section is identified by both the detail ID and the group ID.
<code><!-- END --></code>	Section ends
<code><!-- BEGIN {GF.DetailID.GroupID} --></code>	Group footer section begins. The section is identified by both the detail ID and the group ID.
<code><!-- END --></code>	Section ends

33.3. Examples

The following example shows a typical usage of a text template:

```
<!-- BEGIN {DH} -->
SIMPLE TEXT REPORT OUTPUT
<!-- END -->
<!-- BEGIN {PH} -->
Customer ID      Name      Address
-----
<!-- END -->
<!-- BEGIN {D.Detail1} -->
${D{custid}}      ${D{custname}}      ${D{address}}
<!-- END -->
```

33.4. Running a report to text output

To run a report to text output from code, use the `runReportToText()` function, which takes the output file name and the template file name:

```
NCReport *report = new NCReport();
report->setReportFile( "customers.ncr" );
report->runReportToText( "customers.txt", "customers_template.txt" );
```

For more information about the text output class, see [Section 32.11](#).

Chapter 34. Batch Report Mode

Batch report mode enables running multiple reports into a single output. The member reports are joined in a specified order and run one after another as if they were one report: the result is a single print job, a single PDF file, or a single preview document. Each member report keeps its own report definition, so the reports may use completely different data sources, sections, and layouts.

Typical use cases:

- Printing documents that belong together in one step, for example an invoice, a delivery note, and a cover letter.
- Creating one PDF file from several independent reports.
- Combining a summary report with one or more detailed reports.

34.1. How batch mode works

Batch mode is enabled as soon as at least one report is added to the batch of an `NCReport` object (`isBatchMode()` returns true in this case). When the report is run, the engine processes the reports of the batch one by one, in the order they were added, using the same output object. The output is initialized when the first report starts and finalized when the last report ends; the reports in between are rendered into the same output.

- By default, every report starts on a new page. Call `setBatchModePageBreakBetweenPages(false)` to disable the page break between the reports.
- Page numbering restarts for every member report: the `pageno` system variable shows the page number within the current report.
- The `gpageno` system variable returns the global page number, counted across all the reports of the batch.
- The `reportno` system variable returns the number of the current report in the batch (starting from 1), and the `reportcount` system variable returns the total number of reports in the batch.



When the QPrinter-based PDF output (`NCReportPdfOutput`) is used, batch mode is supported only if all reports have the same page settings; otherwise the result is incorrect.

34.2. Batch mode API

The batch-related functions of the `NCReport` class are the following:

`addReportToBatch( const QString& reportXML )`

Adds a report to the batch. The string is either a report definition XML string or a report file name. A string that ends with `.xml` or `.ncr` is treated as a file name.

`addReportToBatch( const QString& reportXML, const QHash<QString,QVariant>& parameters )`

Adds a report to the batch together with the parameters that are added to the report when it runs.

`addReportToBatch( NCReport* report )`

Adds an already configured `NCReport` object to the batch. When this report runs, its report source, parameters, item models, string lists, and data sources are used.

`setReportBatch( const QStringList& )`

Sets the whole batch at once from a list of report file names or XML strings.

`clearBatch()`

Removes all reports from the batch.

`isBatchMode()`

Returns true if the batch contains at least one report.

`setBatchModePageBreakBetweenPages( bool )`

Specifies whether a page break is inserted between the reports. The default is true.

`batchReportRunning( int reportNumber )`

Signal emitted before each report of a string-based (file name or XML) batch starts.



Don't mix the two kinds of batches. If any `NCReport` object has been added to the batch with `addReportToBatch( NCReport* )`, only the object batch is run, and the reports added as file names or XML strings are ignored.

34.3. Example

The following example runs three report files into one PDF file. The second report gets its own parameter. For further details, see the [Section 38.18](#) section in the "Using the NCReport API" chapter.

```
NCReport *report = new NCReport();

report->clearBatch();
report->addReportToBatch( "/home/anywhere/report1.xml" );

QHash<QString,QVariant> params;
params.insert( "category", 2 );
report->addReportToBatch( "/home/anywhere/report2.xml", params );

report->addReportToBatch( "/home/anywhere/report3.xml" );

report->runReportToPdfWriter( "batch.pdf" );

if ( report->hasError() )
    qWarning() << report->lastErrorMsg();

delete report;
```

34.4. Running batch reports in the Designer

The **Run report** dialog of the Designer has a **Batch mode** panel that lists the report files of all open documents. If **Run reports in batch mode** is checked, all of the listed reports are run in batch mode into the selected output. The option is available only when more than one report document is open. Since the reports are run from their files, save your changes before running the batch.

The `batch_demo_report1.xml`, `batch_demo_report2.xml`, and `batch_demo_report3.xml` demo reports in the `reports` directory of the NCReport package are designed for trying out batch mode.

Chapter 35. Special Detail Sections

Detail sections have some special features that can be very useful at times. Certain special details can be rendered at a non-standard location, or repeated. These features are the sub-section, the odd/even page content, and the repeated detail.

35.1. Sub (Detail) Sections

Sometimes a single report section (detail) is not enough to hold all of the content, and one or more additional sections would be useful for adding more items. Without zones, a section is the smallest band area in which a group of items is kept together on a page. A sub-section is an additional content area that is linked to its parent section. The parent can be a detail, a group header, or another section. The sub-section is always rendered below its parent. With the **printWhen** option of the section, the sub-section can be hidden or shown conditionally, which makes the report design very flexible.

35.2. Adding a Sub-Section

To add a sub-section, select **Report > Details and grouping...** and add a new Detail section. Normally, a detail section needs an assigned data source, but a sub-section does not. Set the **Parent section ID** on the **Special** tab to the ID of the section you want to link to.

The screenshot shows the 'Details' dialog box with the 'Special' tab selected. On the left, a list titled 'Details' contains 'Detail1'. To the right of the list are icons for adding (+), deleting (X), and moving (up/down arrows). The main area on the right is divided into sections: 'Options' with 'Secondary data source' and 'Parent section ID' fields; 'Alternating rows' with 'Alternating mode' set to 'Disabled', a checked 'Use default background color' checkbox, and a 'Background color' field set to '#dcdcdc'; and 'Repeating' with a 'Repeat detail' field followed by 'times'. At the bottom are buttons for 'Groups...', 'OK', 'Cancel', and 'Apply'.

The sub-section appears directly after its parent section, regardless of the other details.

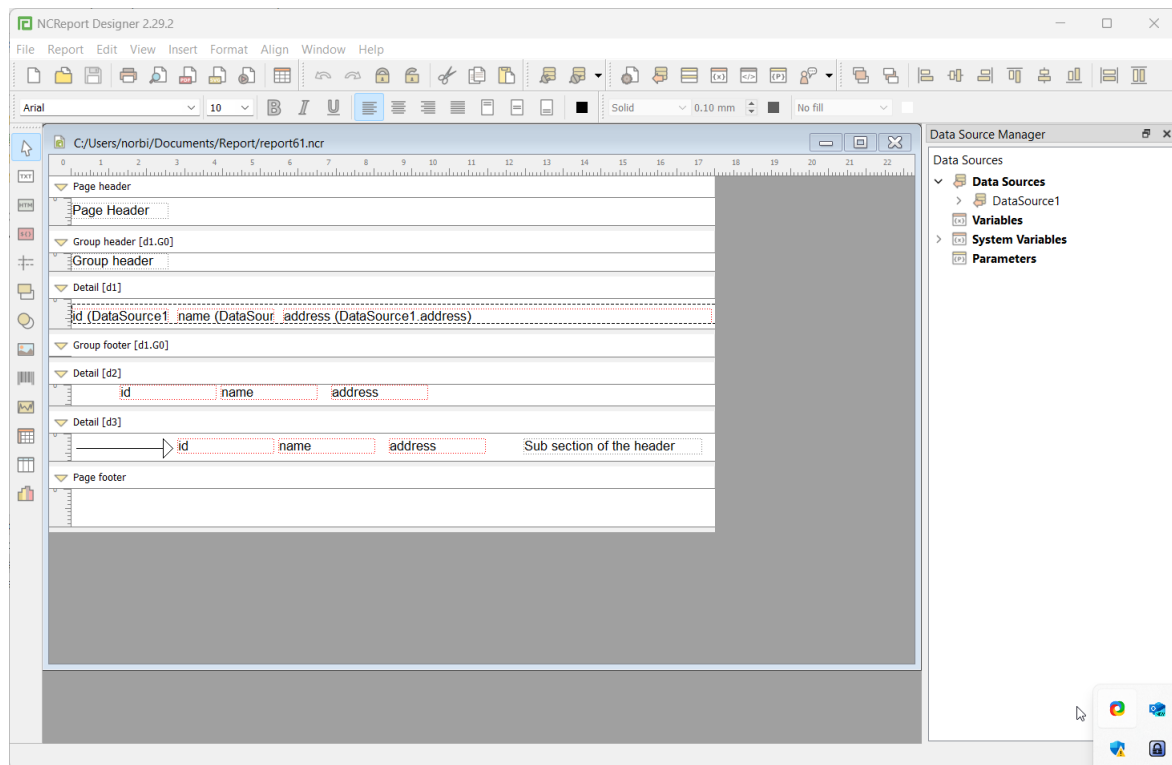
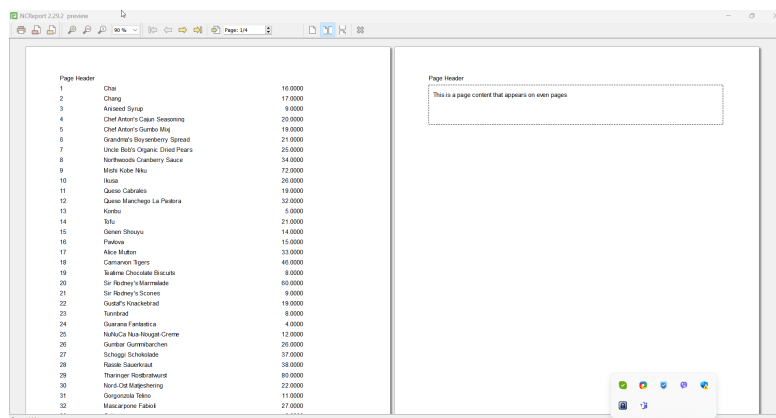


Figure 28. Example of sub-sections

35.3. Odd / Even Pages

The odd/even page feature is a special use case of the sub-section. It makes it possible to create a report with alternating odd and even pages, where different content is inserted on every odd or every even page. For this purpose, you can specify a dedicated detail section that represents the content of the odd pages or the even pages.



To create odd or even page content, add a new detail to your report and set one of the following identifiers as the **Parent section ID** on the **Special** tab of the **Detail settings** dialog. It is recommended to assign a dummy, one-row data source to such details, to ensure that their content is printed only once per page.

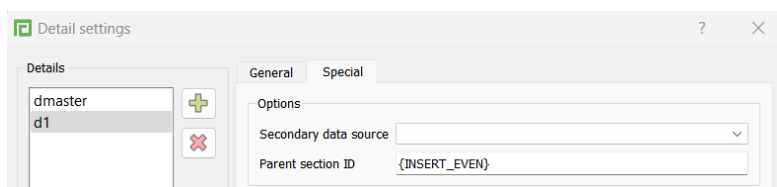
```
{INSERT_EVEN} or {INSERT_ODD}
```

{INSERT_EVEN}

Content that is inserted on every even page.

{INSERT_ODD}

Content that is inserted on every odd page.



35.4. Repeated detail by constant or dynamic value

By default, every detail section is printed once per data source row, but sometimes the detail content needs to be repeated several times, for instance when the report is intended for printing tickets, barcodes, or labels. To set how many times the detail is repeated, use the **Repeat detail ... times** field on the **Special** tab of the **Detail settings** dialog. You can set either a constant integer value or an expression that results in an integer. The feature works for normal detail sections only.

Example: 5 or `$P{numberofdetails}`, or `$D{num}` ...

Repeating

Repeat detail times

Page Header		
1	Chai	16.0000
1	Chai	16.0000
1	Chai	16.0000
1	Chai	16.0000
1	Chai	16.0000
2	Chang	17.0000
2	Chang	17.0000
2	Chang	17.0000
2	Chang	17.0000
2	Chang	17.0000
3	Aniseed Syrup	9.0000
3	Aniseed Syrup	9.0000
3	Aniseed Syrup	9.0000
3	Aniseed Syrup	9.0000
3	Aniseed Syrup	9.0000

Chapter 36. User Functions

User functions let the host application take part in report generation with its own C++ code. A user function is an object of a class derived from `NCReportUserFunction`, registered in the report under a string ID. Any report item can refer to a user function by this ID; the engine then calls the function every time the item is refreshed, and the function can modify the item before it is printed: change its text, color, font or other properties, based on report data or on any information available in the application.

User functions are useful when a rule is too complex, too slow or simply not practical to express as a script expression, or when it depends on application state that is not available to the report as data, such as business logic in C++, user permissions or runtime settings.

36.1. The `NCReportUserFunction` Class

The public interface of the class, declared in `ncreportuserfunction.h`, is:

```
class NCReportUserFunction
{
public:
    NCReportUserFunction();
    virtual ~NCReportUserFunction();

    enum UserAccess { Invisible=0, ReadOnly, ReadWrite };

    void setUserAccess( UserAccess access );
    UserAccess userAccess() const;

    virtual void updateValue( NCReportItem* item, NCReportOutput* output,
                             NCReportEvaluator* evaluator ) = 0;
};
```

`updateValue()`

The only method you must implement. Its arguments are:

- `item`: the report item being refreshed. Use `item->objectType()` (for example `NCReportItem::Label`, `NCReportItem::Field`, `NCReportItem::Image`) to check the item type before casting it to the concrete item class.
- `output`: the current report output.
- `evaluator`: the report evaluator. Its public `evaluate(QString&)` method replaces data source (`$D{...}`), parameter (`$P{...}`) and variable (`$V{...}`) tokens in a string with their current values, so the function can read the current record.

`setUserAccess()` / `userAccess()`

Control how the **Function ID** field is presented in the Designer's item property dialogs when the item refers to this function: `ReadWrite` (the default) leaves the field editable, `ReadOnly` makes it read-only, and `Invisible` hides it.

36.2. When the Function Is Called

The report engine refreshes every item of a section each time the section is rendered, by calling the item's `updateValue()` method; this happens before the item's **Print When** expression is evaluated. At the end of this step, if the item has a function ID and a user function is registered with that ID, the engine calls the user function's `updateValue()` with the item.

The item's own value evaluation runs before the user function for field, text and barcode items, so the function sees the item already updated for the current record. For field items, note that the displayed text is formatted from the field value during this evaluation; a user function is therefore best suited to changing the field's appearance (colors, font) rather than its value. For static text (label) items, the printed text can be replaced with `setText()`.

36.3. Assigning a Function to an Item

Each report item has a **Function ID** property, found among the **Advanced Options** at the bottom of the item property dialogs (next to the item ID and zone ID). In the report definition it is stored in the `funcID` attribute:

```
<field id="F1" zValue="0" posX="0.000" posY="0.000" width="30.000" height="4.762"
      resource="datasource" type="num" ftype="ds" funcID="lateHighlight">days_late</field>
```

User functions are not called from script expressions or from data source expressions; they are attached to items only through the function ID. If no function is registered with the given ID, the property is ignored.

36.4. Implementing and Registering a User Function

The following example changes the text color of a field to red when the order belonging to the current record is late. The number of late days is read from the current record of the `orders` data source.

Declaration and implementation:

```
#include "ncreportuserfunction.h"
#include "ncreportfielditem.h"
#include "ncreportevaluator.h"

class LateOrderHighlighter : public NCReportUserFunction
{
public:
    void updateValue(NCReportItem* item, NCReportOutput* output,
                    NCReportEvaluator* evaluator) override
    {
        Q_UNUSED(output)
        if (item->objectType() != NCReportItem::Field)
            return;

        QString daysLate(QString{"${orders.days_late}");
        evaluator->evaluate(daysLate); ①

        NCReportFieldItem* field = static_cast<NCReportFieldItem*>(item);
```

```

        field->setForeColor(daysLate.toInt() > 0 ? Qt::red : Qt::black);
    }
};

```

① The data reference is replaced by the current value.



The same item object is used for every record. If the function changes a property under some condition, it should also set it back in the other case, as the example does with the black color.

Registration with `NCReport::addUserFunction()`:

```

LateOrderHighlighter lateHighlight;    ①

NCReport *report = new NCReport(this);
report->setReportFile("orders.ncr");
report->addUserFunction("lateHighlight", &lateHighlight);
report->runReportToPreview();

```

① The function object must live while the report runs.

The report does not take ownership of the function object, so you are responsible for keeping it alive while the report runs and for deleting it afterwards.

36.4.1. Using user functions in the Designer

If your application embeds the NCReport Designer, register the functions in the Designer main window with `NCRDMainWindow::addUserFunction()`. The functions registered there are passed to the report engine when a report is run from the Designer, so the preview shows their effect:

```

LateOrderHighlighter lateHighlight;
window.addUserFunction("lateHighlight", &lateHighlight);

```

Custom toolbar actions created with `addCustomToolAction()` accept an optional user function ID as their last argument; items created with such an action get this function ID automatically. Combined with `setUserAccess(NCReportUserFunction::ReadOnly)` or `Invisible`, this lets you provide predefined item types whose function binding cannot be changed by report designers.

36.5. Script-Based Functions

User functions are a C++ mechanism. Reports can also contain reusable logic written in script, which requires no application code:

JavaScript functions in expressions

A script expression (for example, a field with the **Script expression** source type) can define JavaScript functions and call them in the same expression. The `Javascript_Function_Demo.ncr` demo report pads a number with leading zeros this way:

```
function pad(n, width, z)
{
  z = z || '0';
  n = n + '';
  return n.length >= width ? n : new Array(width - n.length + 1).join(z) + n;
}

pad(${DataSource1.col2},6)
```

The data source token is replaced with the current value before the script runs.

Stored scripts

Script code used in several places can be stored in the report with the Script Editor and referenced with the `$$scriptId` token. See the Script Editor chapter.

Data source and value functions

Data source expressions support built-in functions such as `rowCount()`, `isAvailable()` or `MID(n,m)` (see the `FunctionsDemo.ncr` demo report). These are described in the Data/Script Expressions chapter.

Part IV: Command Line Tool

This part describes the command line tool of the report engine, which runs reports without a graphical user interface.

Chapter 37. Command line client

The NCReport engine is also available as a command line client executable. It can run a report definition file to a printer, to the preview window, or to a file output without writing any code, which makes it useful for testing reports and for running reports from scripts.

37.1. Running the command line executable

To run the command line engine, use the `NCReport` executable in the `bin` directory of the installation:

```
NCReport [options]
```

If the executable is started without any options, it prints the list of available options and exits.

Example: running a report definition file to a PDF file with a parameter:

```
NCReport -f ../reports/gettingstarted_demo.xml -o pdf -of result.pdf -par firstname,Robert
```

37.2. Command line options

-?, --help

Displays the help text and exits.

-v, --version

Displays the NCReport version and exits.

-f, --report-file <filename>

Name of the report definition (XML) file to run.

-q, --sql-driver <driver>

Qt SQL driver name for the database connection, for example QDB2, QIBASE, QMYSQL, QOCI, QODBC, QPSQL, QSQLITE, or QTDS. The default is QMYSQL.

-h, --host <hostname>

Database host name or IP address. The default is localhost.

-u, --user <username>

Database user login name.

-p, --password <password>

Database user login password.

-d, --database <dbname>

Name of the database to use. A database connection is opened only if this option is specified.

-pt, --port <port>

Database port number. If not specified, the default port of the driver is used.

-c, --connection-id <id>

SQL connection name (ID). It must match the connection ID of the SQL data sources in the report. The default is `conn`. If the ID is `default`, the connection is added as the default database connection.

-cs, --connect-string <string>

Joined connection string in `<user>/<password>@<host>` format. It sets the user name, password, and host name in one step.

-co, --connect-option <opt>

Connection option string.

-o, --output <output type>

The output type: `print`, `preview`, `pdf`, `svg`, `html`, `excel`, or `image`. Use `print` to send the report to a printer, or `preview` to show the report in a preview window. The `excel` output generates an Excel-compatible HTML file. The default output is `preview`.

-of, --output-file <filename>

Output file name. Required for the file-based output types (`pdf`, `svg`, `html`, `excel`, and `image`). The default is `test.pdf`.

-n, --printer <printername>

Name of the target printer.

--copies <1..50>

Number of copies when the output is a printer.

--nodialog

Runs the report to the printer without showing the printer dialog.

-dbid, --report-db-id <id>

ID number of the database record that contains the report definition (XML) text. This option and `--report-file` must not be used together.

-par, --add-parameter <parametername>,<value>

Adds a custom parameter to the report. Specify the name and the value of the parameter separated by a comma, for example `firstname,Robert`. The parameter can be referenced as `${parametername}` in the report definition. The option can be used more than once to add several parameters.

-l, --language <code>

Two-letter international language code of the current report language. Works only when the specified language is defined in a multilingual report.

-td, --template-dir <dir>

Sets the default template directory used for additional files, such as chart templates.

-hs, --htmlstrategy <1|2>

HTML output generation strategy: **1** = each section is rendered as one table (default); **2** = sections are always rendered as separate tables.

-css, --css-file <filename>

Style sheet file to be generated for the HTML output. If not set, the style sheet is included in the target HTML file.



In the current version of the console application, the `--connect-option` and `--report-db-id` options are accepted, but they are not applied when the report runs; the report definition is always loaded from the file specified by `--report-file`.

Part V: Using the NCReport API

This part shows how to use the NCReport API in your own Qt/C++ application: running reports, using different data sources and outputs, extending the engine, and monitoring the report process.

Chapter 38. Using the NCReport API

This chapter shows how to use the report API from a C++ (Qt) application: how to create an NCReport object and how to run reports with it. As described earlier, the NCReport system consists of two parts: the report renderer library and the report designer GUI application. The report engine can, of course, be used without the Designer.

To call NCReport from your application, you first have to integrate NCReport into your project. There are several ways to do so:

- Add the entire source code to your project and build it together with your application.
- Use the NCReport engine as a shared library. To use the NCReport library like any other library in your project, you need to declare it in your project file. For more information, see the "Declaring Other Libraries" chapter of the Qt **qmake** manual.
- Link the NCReport library statically into your project. For more information, see the "Declaring Other Libraries" chapter of the Qt **qmake** manual.

38.1. Project file settings

Add at least the following lines to your .pro file. The library names follow the naming used by the NCReport build (**NCReport** for release and **NCReportd** for debug builds; on Windows, the major version number **2** is appended). If NCReport was built with barcode support, the Zint barcode library must also be linked:

```
NCREPORT_LIBPATH = /home/ncreport/lib

CONFIG(release, debug|release) {
    win32|win64: LIBS += -L$$NCREPORT_LIBPATH -lNCReport2 -lzint2
    unix:!macx:  LIBS += -L$$NCREPORT_LIBPATH -lNCReport -lzint
    else:macx:   LIBS += -framework NCReport
}
CONFIG(debug, debug|release) {
    win32|win64: LIBS += -L$$NCREPORT_LIBPATH -lNCReportd2 -lzintd2
    unix:!macx:  LIBS += -L$$NCREPORT_LIBPATH -lNCReportd -lzintd
    else:macx:   LIBS += -framework NCReportd
}
INCLUDEPATH += /home/ncreport/include
...
```

38.2. Initializing the NCReport class

This section covers the fundamental steps that most users take when creating and using the NCReport class. The steps are presented in the suggested order.

38.3. Include directives

To include the definitions of the module's classes, use the following directives:

```
#include "ncreport.h"
#include "ncreportoutput.h"
#include "ncreportpreviewoutput.h"
#include "ncreportpreviewwindow.h"
```

38.4. Creating the NCRReport class

Create the report object just as you would any other QObject-based class:

```
NCRReport *report = new NCRReport();
```

If the object was created earlier and is passed as a parameter to the method in which you use it, initialize the report by calling the `reset()` method. If you create a new object in the current scope, you don't need to call `reset()`.

```
report->reset();
```

`reset()` also has an overloaded version that lets you choose exactly what is cleared: renderers, parameters, user functions, string lists, batch reports, and (optionally) custom data sources.

38.5. Connecting to an SQL database

An SQL connection is required only when your data source uses an internal database connection. An internal connection is an already existing database connection that is established before running the report. Reports can also use an external connection (one defined in the report itself). Other data source types don't require a database connection.

This example code shows a typical SQL database connection with error handling:

```
QSqlDatabase defaultDB = QSqlDatabase::addDatabase("QMYSQL", "myconn" );

if ( !defaultDB.isValid() ) {
    QMessageBox::warning( 0, "Report error", QObject::tr("Could not load database driver.") );
    delete report;
    return;
}
defaultDB.setHostName( "host" );
defaultDB.setDatabaseName( "database" );
defaultDB.setUserName( "user" );
defaultDB.setPassword( "password" );

if ( !defaultDB.open() ) {
    QMessageBox::warning( 0, "Report error", QObject::tr("Cannot open database:") + defaultDB.lastError
    ().databaseText() );
    return;
}
```

38.6. Setting the report's source

The report source is the way NCReport obtains the XML report definition; in other words, it is the source of the report definition data. In most cases report definitions are opened from a file, but the `NCReportSource::ReportSourceType` enumeration also provides the `Database`, `Http`, `Ftp`, `Custom`, and `String` source types. A report definition can also be passed directly as an XML string with the `setReportDefXml()` function.

In this example, we use a file as the report source:

```
report->setReportFile( fileName );
```

This is equivalent to the following code:

```
report->setReportSourceType( NCReportSource::File );  
report->reportSource()->setFileName( fileName );
```

38.7. Adding parameters

If your report uses parameters, you have to add the parameter value(s) to the NCReport object before running the report. To create and add a parameter, call:

```
report->addParameter( "id", "value" );
```

where the ID is a QString and the value is a QVariant.

38.8. Running the report

Now we are ready to run the report and catch the error message if an error occurs. There are two ways to start the report engine.

38.9. Running the report in one step

This is the simplest mode, but it offers fewer configuration options. The `runReportTo...`() functions create the appropriate output object internally and run the report. They do not return a value; check the result with `hasError()` and `lastErrorMsg()` afterwards.

```
// run report to printer  
report->runReportToPrinter(1, true, parent);  
// run report to PDF file  
report->runReportToPdf( "file.pdf" );  
// or  
report->runReportToPdfWriter( "file.pdf" ); ①  
// run report to SVG files  
report->runReportToSVG( "file.svg" );  
// run report to preview output  
report->runReportToPreview();
```

```
// run report to QPrintPreview dialog
report->runReportToQtPreview( parent );

bool error = report->hasError();
QString err = report->lastErrorMsg();
```

① The recommended way of creating PDF files.

Further one-step functions are available for other output formats, for example `runReportToImage()`, `runReportToHtml()`, `runReportToExcelHtml()`, `runReportToPostScript()`, and `runReportToText()`.

When `runReportToPreview()` is used, the preview window is not shown automatically: you also have to create and show an `NCRReportPreviewWindow` (see "Previewing the report" below). Alternatively, `runReportToShowPreview( parent )` runs the report and shows the preview window in one step.

38.10. Running the report in customized mode

This mode allows more flexible configuration. First the output object has to be initialized; after that, the report is ready to run.

38.11. Initializing the report's output

The next step is to create and specify the report's output. As the rendering target, NCRReport uses a class derived from the `NCRReportOutput` base class. There are predefined classes for the most commonly used outputs:

- `NCRReportPrinterOutput` - printer
- `NCRReportPreviewOutput` - NCRReport's own print preview
- `NCRReportQtPreviewOutput` - Qt print preview dialog
- `NCRReportPdfOutput` - PDF file through QPrinter
- `NCRReportPdfWriterOutput` - PDF file through QPdfWriter
- `NCRReportSvgOutput` - SVG file(s)
- `NCRReportImageOutput` - image file(s)
- `NCRReportHtmlOutput` and `NCRReportExcelHtmlOutput` - HTML file and Excel-compatible HTML file
- `NCRReportTextOutput` - template-based text output

For a detailed description of the output types and their options, see [Chapter 32](#).

To define the desired output, use code similar to this:

```
NCRReportOutput *output=0;

if ( rbPreview->isChecked() ) {
    output = new NCRReportPreviewOutput();
    output->setAutoDelete( false );
    report->setOutput( output );
} else if ( rbPrinter->isChecked() ) {
```

```

    output = new NCReportPrinterOutput();
    output->setCopies(1);
    output->setShowPrintDialog( true );
    report->setOutput( output );
} else if ( rbPdf->isChecked() ) {
    QString fileName = QFileDialog::getSaveFileName(this, tr("Save PDF File"), "report.pdf", tr("Pdf files (*.pdf)"));
    if ( fileName.isEmpty() ) {
        delete report;
        return;
    } else {
        output = new NCReportPdfOutput();
        output->setFileName( fileName );
        report->setOutput( output );
    }
}
}

```

38.12. Running the report

Now we are ready to run the report and catch the error message if an error occurs:

```

QApplication::setOverrideCursor(QCursor(Qt::WaitCursor));
report->runReport();
bool error = report->hasError();
QString err = report->lastErrorMsg();
QApplication::restoreOverrideCursor();

```

38.13. Previewing the report

If `NCReportPreviewOutput` is specified as the report's output, the preview window is not shown automatically. After the report engine has finished successfully, we need to create an `NCReportPreviewWindow` object for previewing. The following code shows how to do this. It is recommended to check for errors first, before showing the preview window:

```

if ( error ) {
    QMessageBox::information( 0, "Report error", err );
} else {
    if ( rbPreview->isChecked() ) {
        //-----
        // PRINT PREVIEW
        //-----
        NCReportPreviewWindow *pv = new NCReportPreviewWindow();
        pv->setReport( report );
        pv->setOutput( (NCReportPreviewOutput*)output );
        pv->setWindowModality(Qt::ApplicationModal );
        pv->setAttribute( Qt::WA_DeleteOnClose );
        pv->exec();
    }
}

```



Do not delete the output object after it has been added to the `NCReportPreviewWindow` object. The preview window deletes its output object when it is destroyed.



For the best performance and output quality, do not delete the NCReport object until the preview window is closed, and add the report object to the preview window with the `setReport()` function.

```
pv->setReport( report );
```

Once this is done, printing from the preview produces the original printout quality, since it runs the report again instead of printing the lower-quality preview pages.



Since version 2.8.4, the preview window can be shown in dialog mode, just like a QDialog. The `NCReportPreviewWindow::exec()` function shows the preview window and keeps the application event loop running during the preview. This is useful when you use a locally defined report object, because the report object will not be deleted until the user closes the preview window.

38.14. Deleting the report object

After running the report, you may want to delete the report object. Note that the preview window requires the NCReport object to exist. If `NCReportPreviewWindow::exec()` is used, the code following it is not executed until the preview window is closed.

```
delete report;
```

38.15. Using other data sources

NCReport allows you to use non-SQL data sources such as QString-based text, a QStringList, an item model, or a custom-defined data source. Depending on the data source type, data may come from a file, a report parameter, or another source. For a general description of data source types, location types, and opening roles, see [Chapter 10](#).

38.15.1. QString-based text data source

NCReport allows you to use QString text as the simplest data source. Each text row represents one data record (rows are separated by the line feed character), and the data columns are separated by a specified delimiter character. To identify columns in fields, use the column number as a reference in the report, as follows:

`0` for the 1st column, `1` for the 2nd column, etc., or `col0` for the 1st column, `col1` for the 2nd column, etc.

To use text data sources, add a Text data source to your report in the Designer. You have the following options:

- Store static text in the report definition. To do so, select the Static location type in the Designer and

add the static text to the edit box, using the specified delimiter character.

- Use an existing text file. To do so, select the File location type in the Designer and specify the name of the text file you want to use. You must also specify the delimiter character.
- Pass the text to NCReport as a report parameter. To do so, first add the text as a parameter to NCReport with the `addParameter()` function. Then select the Parameter location type in the Designer and specify the ID of the parameter you added. You must also specify the delimiter character.

Example of adding QString text to NCReport as a parameter, using the Tab character as the column delimiter:

```
NCReport report;

QString data;

data += "1 \tChai \t16.0000\t1\t1540\t0\n";
data += "2 \tChang \t17.0000\t1\t 874\t0\n";
data += "3 \tAniseed Syrup \t 9.0000\t1\t1687\t0\n";
data += "4 \tChef Anton's Cajun Seasoning \t20.0000\t1\t1230\t0\n";
data += "5 \tChef Anton's Gumbo Mixj \t19.0000\t2\t1900\t0\n";
data += "6 \tGrandma's Boysenberry Spread \t21.0000\t2\t 520\t0\n";
data += "7 \tUncle Bob's Organic Dried Pears \t25.0000\t3\t 540\t0\n";
data += "8 \tNorthwoods Cranberry Sauce \t34.0000\t3\t 120\t0\n";
data += "9 \tMishi Kobe Niku \t72.0000\t3\t 130\t0\n";
data += "10 \tIkura \t26.0000\t3\t2247\t0\n";
data += "11 \tQueso Cabrales \t19.0000\t4\t 741\t0\n";
data += "12 \tQueso Manchego La Pastora \t32.0000\t4\t 512\t0\n";
data += "13 \tKonbu \t 5.0000\t4\t1470\t0\n";
data += "14 \tTofu \t21.0000\t4\t 978\t0\n";
data += "15 \tGenen Shouyu \t14.0000\t4\t1005\t0\n";

report.addParameter( "data1", data );
```

38.15.2. QStringList data source

NCReport also allows you to use a QStringList as a data source. Each list item represents one data record, and the data columns are separated by a specified delimiter character.

To identify columns in fields, use the column number as a reference in the report, as follows:

`0` for the 1st column, `1` for the 2nd column, etc., or `col0` for the 1st column, `col1` for the 2nd column, etc.

To use a `QStringList` data source, add a **StringList** data source to your report in the Designer. You can only use the **Static** location type, since a `QStringList` can be passed to NCReport in only one way: with the `addStringList()` function. You have to specify an ID for the list, which identifies it.

Example of using a QStringList as a data source:

```
NCReport report;

QStringList list;
```

```
list << "24|Renate Moulding|Desert Hot Springs,CA|1|2008-01-01";
list << "78|Alfred Muller|Miami Beach, FL|1|2008-01-03";
list << "140|Angela Merkel|Munchen, Germany|1|2008-01-07";
list << "139|Bob Larson|Dallas, TX|0|2008-01-20";

report.addStringList( list, "s10" );
```

38.15.3. Item Model data source

The Model/View architecture is a very useful feature of Qt. NCRReport allows you to use a data source based on `QAbstractItemModel`. First you have to create your item model. Each model row represents one data record. To identify columns in fields, use the column number as a reference in the report, as follows:

0 for the 1st column, 1 for the 2nd column, etc., or `col0` for the 1st column, `col1` for the 2nd column, etc.

To use an Item Model data source, add an Item Model data source to your report in the Designer. You can only use the Static location type; other location types are undefined. In your code, add the item model to NCRReport with the `addItemModel(...)` function. You have to specify an ID that identifies the model, and the item model data source in the Designer must have the same ID.

Example of using an item model as a data source:

```
QStandardItemModel *model = new QStandardItemModel( 2, 4 );
QStandardItem *item =0;

// -----
item = new QStandardItem();
item->setData( 1, Qt::EditRole );
model->setItem( 0, 0, item);

item = new QStandardItem();
item->setData( "Chai", Qt::EditRole );
model->setItem( 0, 1, item);

item = new QStandardItem();
item->setData( 16.0, Qt::EditRole );
model->setItem( 0, 2, item);

item = new QStandardItem();
item->setData( 1540.0, Qt::EditRole );
model->setItem( 0, 3, item);

// -----
item = new QStandardItem();
item->setData( 2, Qt::EditRole );
model->setItem( 1, 0, item);

item = new QStandardItem();
item->setData( "Chef Anton's Cajun Seasoning", Qt::EditRole );
model->setItem( 1, 1, item);

item = new QStandardItem();
item->setData( 20.0, Qt::EditRole );
model->setItem( 1, 2, item);
```

```

item = new QStandardItem();
item->setData( 1230.0, Qt::EditRole );
model->setItem( 1, 3, item);

report.addItemModel( model, "model1" );

```

38.16. Custom data sources

NCReport allows you to create your own data source by subclassing the [NCReportDataSource](#) abstract class. This way you can use anything as data for NCReport. You only have to implement the pure virtual methods of the class. To add your data source object to NCReport, use the [addCustomDataSource\(\)](#) function. The following example (based on the [TestDataSource](#) class of the NCReport runner application) demonstrates how to define and use a custom data source class:

38.16.1. Declaration

```

#include "ncreportdatasource.h"
#include <QDate>

struct TestData {
    int id;
    QString name;
    QString address;
    bool valid;
    QDate date;
};

class TestDataSource : public NCReportDataSource
{
    Q_OBJECT
public:
    TestDataSource( QObject *parent=0 );
    ~TestDataSource();

    void addData( const TestData& data );

    bool open();
    bool close();
    bool first();
    bool last();
    bool next();
    bool previous();
    bool hasNext();
    bool seek( int index );
    int size() const;
    bool isValid() const;
    QVariant value( const QString& column, bool* ok=0, int nQTDatRole=-1 ) const;
    QVariant value( int column, bool* ok=0, int nQTDatRole=-1 ) const;
    bool read( NCReportXMLReader* );
    bool write( NCReportXMLWriter* );
    int columnIndexByName( const QString& columnname ) const;
    int columnCount() const;
    QString columnName( int column ) const;
private:
    QList<TestData> list;
};

```

38.16.2. Implementation

```

TestDataSource::TestDataSource(QObject * parent) : NCReportDataSource( parent )
{
    setDataSourceType(Custom);
    setLocationType(Static);
    recno() =0;
}

TestDataSource::~~TestDataSource()
{
}

bool TestDataSource::open()
{
    if ( list.isEmpty() ) {
        error()->setError( tr("No data in TestDataSource data source") );
        return false;
    }
    recno() =0;
    setOpened(true);
    return true;
}

bool TestDataSource::close()
{
    recno() =0;
    setOpened(false);
    return true;
}

bool TestDataSource::next()
{
    recno()++;

    if ( recno() >= list.count() ) {
        recno()--;
        flagEnd() = true;
        return false;
    }
    flagBegin() = false;
    return true;
}

bool TestDataSource::previous()
{
    if ( recno() <= 0 ) {
        recno() = 0;
        flagBegin() = true;
        return false;
    }
    recno()--;
    return true;
}

bool TestDataSource::first()
{
    recno()=0;
    return true;
}

bool TestDataSource::last()
{

```

```

    recno() = list.count()-1;
    return true;
}

bool TestDataSource::hasNext()
{
    return ( recno() < list.count()-1 );
}

bool TestDataSource::seek( int index )
{
    if ( index == -1 )
        return true; ①
    if ( index < list.count() ) {
        recno() = index;
        return true;
    }
    recno() = list.count()-1;
    return false;
}

int TestDataSource::size() const
{
    return list.count();
}

bool TestDataSource::isValid() const
{
    return true;
}

QVariant TestDataSource::value(const QString &column, bool* ok, int nQTDataRole ) const
{
    int col = columnIndexByName(column);
    if ( col < 0 ) {
        if (ok)
            *ok = false;
        return QVariant();
    }
    return value( col, ok, nQTDataRole );
}

QVariant TestDataSource::value( int column, bool*, int ) const
{
    QVariant v;
    switch (column) {
        case 0: v = list.at(recno()).id; break;
        case 1: v = list.at(recno()).name; break;
        case 2: v = list.at(recno()).address; break;
        case 3: v = list.at(recno()).valid; break;
        case 4: v = list.at(recno()).date; break;
    }
    return v;
}

int TestDataSource::columnIndexByName(const QString &columnname) const
{
    for ( int i=0; i<columnCount(); i++ ) {
        if ( columnname == columnName(i) )
            return i;
    }
    return -1;
}

```

```

int TestDataSource::columnCount() const
{
    return 5;
}

QString TestDataSource::columnName(int column) const
{
    switch (column) {
        case 0: return "id";
        case 1: return "name";
        case 2: return "address";
        case 3: return "valid";
        case 4: return "date";
    }
    return QString();
}

bool TestDataSource::read(NCReportXMLReader *)
{
    return true;
}

bool TestDataSource::write(NCReportXMLWriter *)
{
    return true;
}

void TestDataSource::addData(const TestData & data)
{
    list.append( data );
}

```

① An index of `-1` keeps the current record.

38.16.3. Using the TestDataSource class

Now let's use the `TestDataSource` class. To use it, add a Custom data source to your report in the Designer. You can only use the **Static** location type for this data source. Specify the same data source ID both in the Designer and in your code, using the `setID()` function.

```

NCReport report;

TestDataSource *ds = new TestDataSource();
ds->setID("cds0");

TestData d1;
d1.id = 123;
d1.name = "Alexander Henry";
d1.address = "HOT SPRINGS VILLAGE, AR";
d1.valid = true;
d1.date = QDate(2008,01,10);
ds->addData( d1 );

TestData d2;
d2.id = 157;
d2.name = "Julius Coleman";
d2.address = "Coronado, CA";
d2.valid = false;
d2.date = QDate(2008,01,12);
ds->addData( d2 );

```

```
TestData d3;
d3.id = 157;
d3.name = "Peter Moulding";
d3.address = "San francisco, CA";
d3.valid = true;
d3.date = QDate(2008,01,07);
ds->addData( d3 );

report.addCustomDataSource( ds );
```

38.17. Custom items in NCReport

Custom items are powerful members of the report system. The custom item feature enables you to render special, custom-defined content in reports. The typical use case is rendering graphs or similar content.

- Add a Graph (Custom) item to your report in the Designer and specify the size and location of the object. Specify the class ID of the item. This ID is used for identification.
- Subclass the `NCReportGraphRenderer` class and implement its `paintItem()` method.
- Set the ID of your object with the `setID()` function.
- Add your renderer object to NCReport with the `addGraphItemRenderer()` function.



Older versions used the `NCReportAbstractItemRendering` base class and the `addItemRenderingClass()` function. Both still exist for backward compatibility, but they are obsolete; use `NCReportGraphRenderer` and `addGraphItemRenderer()` in new code.

For chart rendering, see also [Chapter 18](#). To modify report items from code while the report runs, see [Chapter 36](#).

Here is an example of a custom item class:

38.17.1. Declaration

```
#include "ncreportgraphrenderer.h"

class TestItemRendering : public NCReportGraphRenderer
{
public:
    TestItemRendering();
    ~TestItemRendering();

    void paintItem( QPainter* painter, NCReportOutput* output, const QRectF& rect, const QString& itemdata );
};
```

38.17.2. Implementation

```
#include "testitemrendering.h"
#include "ncreportoutput.h"
```

```

#include <QPainter>
#include <QColor>

TestItemRendering::TestItemRendering()
{
}

TestItemRendering::~TestItemRendering()
{
}

void TestItemRendering::paintItem(QPainter * painter, NCReportOutput* output, const QRectF & rect, const
QString & itemdata)
{
    switch ( output->outputType() ) {
    case NCReportOutput::Printer:
    case NCReportOutput::Pdf:
    case NCReportOutput::Preview:
        break;
    default:
        return;
    }

    const int numcols = 10;
    const int cw = qRound(rect.width()/numcols);
    painter->setPen( Qt::NoPen );
    int ch=0;
    QColor color;
    color.setAlpha( 128 );

    for ( int i=0; i < numcols; ++i ) {
        if ( i%3 == 0 ) {
            color.setRgb(0xAAAAFF);
            ch = qRound(rect.height()*0.8);
        } else if ( i%2 == 0 ) {
            color.setRgb(0xAAFFAA);
            ch = qRound(rect.height()*0.4);
        } else {
            color.setRgb(0xFFAAAA);
            ch = qRound(rect.height()*0.6);
        }
        painter->setBrush( QBrush(color) );
        painter->drawRect( rect.x()+i*cw, rect.y()+qRound(rect.height())-ch , cw, ch );
    }

    painter->setPen( QPen(Qt::black) );
    painter->setBrush( Qt::NoBrush );

    painter->drawRect( rect );

    painter->setFont( QFont("Arial",8) );
    painter->drawText( rect, Qt::AlignHCenter | Qt::AlignVCenter | Qt::TextWordWrap,
        QString("GRAPH EXAMPLE: %1").arg(itemdata) );
}

```

38.17.3. Using the class

```

NCReport report;

TestItemRendering *irc = new TestItemRendering();

```

```
irc->setID("testitem0");
report.addGraphItemRenderer( irc );
```

38.18. Batch report mode

Batch report mode is a feature that enables running multiple reports into one output. Batch mode makes it possible to join two or more reports in a specified order and run them as one report. Page numbering is not continuous: each member report keeps its own page numbering (the `gpageno` system variable returns the global page number across the whole batch). In reports, the `reportno` and `reportcount` system variables can be used to determine the current report number and the total number of reports in the batch. For a more detailed description, see [Chapter 34](#).

38.18.1. Using batch mode

Batch mode is enabled as soon as a report is added to the batch with the `addReportToBatch(...)` function. The function accepts either a report definition XML string or a report file name (a string ending with `.xml` or `.ncr` is treated as a file name). This example shows how to prepare a batch report from existing report files by loading their content into strings:

```
#include "ncreportutils.h"
...
report->clearBatch();
QString report1;
NCReportUtils::fileToString( "/home/anywhere/report1.xml", report1 );
report->addReportToBatch( report1 );

QString report2;
NCReportUtils::fileToString( "/home/anywhere/report2.xml", report2 );
report->addReportToBatch( report2 );

QString report3;
NCReportUtils::fileToString( "/home/anywhere/report3.xml", report3 );
report->addReportToBatch( report3 );
```

The order of the reports in the batch matches the order of the `addReportToBatch(...)` calls.

38.19. Monitoring report progress

While a report runs, the `NCReport` object emits signals that report the state of the run. You can connect them to a progress dialog, a status bar or a log. The signals are emitted synchronously from the thread that calls `runReport()` (or one of the `runReportTo...` functions), so the connected slots run during the report run, before `runReport()` returns.

38.19.1. Progress signals

`void reportStarts()`

Emitted by `runReport()` after the interactive parameters (see `getParameters()` below) have been collected and before the report definition is loaded and processed. It is emitted only once per

`runReport()` call, also in batch mode. It is not emitted if no output is set or if the user cancels a parameter input dialog.

void reportEnds()

Emitted once at the end of `runReport()`, after all reports of the run (including all batch members and forced copies) have been processed and the output has been finished. It is also emitted when the run ended with an error or was cancelled. Output errors are copied into the error state of the report only after this signal, so check `hasError()` and `lastErrorMsg()` after `runReport()` returns rather than in a slot connected to `reportEnds()`.

void dataSourceOpen(const QString& dsID)

Emitted just before a data source is opened, for example before the SQL query of an SQL data source is executed. Despite the parameter name, the string is a translatable message in the form `Opening data source: <id>`, so it can be shown directly as a label text.

void dataSourceOpened(const QString& dsID, int totalRows)

Emitted after a data source has been opened successfully. The string is a message in the form `Data source opened: <id>`; `totalRows` is the size of the data source as reported by the data source itself, or `-1` if the size is not available (for example with SQL drivers that cannot report the query size).

void dataRowCount(int numRows)

Emitted when the processing of a detail section's data source begins: once after all data sources have been opened, and again each time the engine switches to the data source of the next detail. `numRows` is the row count of that data source. Use it to set the maximum of a progress bar. The value can be `-1` (or `0`) for data sources that cannot report their row count.

void dataRowProgress(const QString& dsID, int row), void dataRowProgress(int row)

Emitted each time the engine moves to the next record of the current detail's data source. The first overload also passes the ID of the data source. The row counter restarts at the beginning of every detail.

void sectionProgress(const QString& sectionID)

Emitted every time the engine starts to render a section. The parameter is the ID of the section. This signal is emitted very frequently; connect it only to lightweight slots.

void pageProgress(int page), void pageProgress(const QString& text)

Emitted when a new page is started. The integer overload passes the global page number (the page number across the whole batch). The string overload passes a ready-made message built from the page progress text, which is `Report is running (page %1)...` by default and can be changed with `setPageProgressText( const QString& );` the `%1` placeholder is replaced with the page number.

void batchReportRunning(int reportNumber)

Emitted in batch mode before each member report starts. `reportNumber` is 1-based. This signal is emitted only for batches built from report definition strings or file names (`addReportToBatch( const QString& ), setReportBatch()`); it is not emitted for batches built with `addReportToBatch( NCReport* )`.

In double-pass mode the engine processes the report twice. The progress signals are emitted in both passes, and the row values are scaled accordingly: `dataRowCount()` passes twice the row count, and during the second pass `dataRowProgress()` continues from the row count, so a progress bar fills up exactly once. Page numbers, however, are reported for both passes.

38.19.2. Using a progress dialog

The following example runs a report into a PDF file and shows its progress in a `QProgressDialog`. The **[Cancel]** button of the dialog stops the report with `cancel()`.

```
#include <QProgressDialog>
#include <QMessageBox>
#include <QFile>
#include "ncreport.h"

void MainWindow::runOrdersReport()
{
    NCReport report;
    report.setReportFile( "orders.ncr" );
    report.setProcessEvents( true ); ①

    QProgressDialog progress( tr("Running report..."), tr("Cancel"), 0, 0, this );
    progress.setWindowModality( Qt::WindowModal );
    progress.setMinimumDuration( 500 );
    progress.setAutoReset( false ); ②
    progress.setAutoClose( false );

    connect( &report, &NCReport::dataSourceOpen, &progress, &QProgressDialog::setLabelText );
    connect( &report, &NCReport::dataRowCount, &progress, [&progress](int rows) {
        progress.setMaximum( rows > 0 ? rows : 0 ); ③
        progress.setValue( 0 );
    });
    connect( &report, QOverload<int>::of(&NCReport::dataRowProgress),
        &progress, &QProgressDialog::setValue );
    connect( &report, QOverload<const QString>::of(&NCReport::pageProgress),
        &progress, &QProgressDialog::setLabelText );
    connect( &progress, &QProgressDialog::canceled, &report, &NCReport::cancel );

    report.runReportToPdfWriter( "orders.pdf" );
    progress.close();

    if ( progress.wasCanceled() ) {
        QFile::remove( "orders.pdf" ); ④
    } else if ( report.hasError() ) {
        QMessageBox::warning( this, tr("Report error"), report.lastErrorMsg() );
    }
}
```

- ① Keeps the GUI responsive while the report runs.
- ② The row counter restarts for every detail, so the dialog must not reset itself.
- ③ A maximum of 0 shows a busy indicator.
- ④ The file contains only the pages rendered before cancelling.

`QOverload` requires Qt 5.7 or later. With the old string-based syntax use `SIGNAL(dataRowProgress(int))` and `SIGNAL(pageProgress(QString))`; this is how the NCReport runner application (`src/runner`) connects

its progress dialog.

38.19.3. Cancelling a report

`cancel()` is a public slot. It has an effect only while a report is running; outside of a run it does nothing. When it is called, the engine:

- stops rendering at the next record (the remaining records are not processed and the report footer is not printed); text document reports (`TextDocumentPrint` type) don't check for cancellation,
- closes the current page without rendering the page footer,
- finishes the output normally: the output's `end()` and `cleanup()` functions still run, so a PDF or other file output is written with the pages rendered so far, a printer output sends those pages to the printer, and a preview output contains them,
- sets the error state of the report: `hasError()` returns `true` and `lastErrorMsg()` returns `Report process cancelled`.

The return value of `runReport()` and the result of `reportDone()` are not reliable indicators of a cancelled run: both can still report success. Detect cancellation from your own user interface (as `progress.wasCanceled()` does in the example above) or from `hasError()`.



Because the output is finished normally, cancelling a printer output still prints the pages rendered so far, and cancelling a file output leaves a partial file behind. Delete the file yourself if you don't need it.

38.19.4. Keeping the user interface responsive

The report runs in the calling (usually the GUI) thread. Without further measures the event loop is blocked while the report runs, so windows are not repainted and button clicks, such as the [**Cancel**] button of a progress dialog, are not processed.

`void setProcessEvents( bool enable ), bool isProcessEvents() const`

When enabled, the engine calls `QApplication::processEvents()` during the run: before each data source is opened, after the data sources have been opened, at the beginning of every page, and for every detail in text document mode. This lets the application repaint its windows and react to user input, which is required if you want to cancel the run from your user interface. It is disabled by default; the NCRReport Designer enables it.

Note that `processEvents()` is not called for every record, and the engine cannot process events while a long-running SQL query is being executed. A window-modal `QProgressDialog` additionally processes events itself whenever its value is set.

`void setShowWaitCursor( bool set ), bool showWaitCursor() const`

When enabled, the engine sets the `Qt::WaitCursor` override cursor while the report is rendered and restores it afterwards. The cursor is set after the output has been initialized, so it does not affect the print dialog of a printer output. It is disabled by default. Disable it when you show your own

progress dialog, since the wait cursor suggests that the application does not accept input.

void setUseWaitCursor(bool)

Obsolete. The value is stored, but the engine does not use it. Use `setShowWaitCursor()` instead.



While events are processed, the user can trigger other actions of the application, for example start another report or close the window that owns the `NCReport` object. Disable the relevant actions (or use a modal progress dialog) while the report runs, and don't delete the `NCReport` object from a slot that is called during the run.

38.20. Customization hooks

Besides data sources, item renderers and user functions, the API offers a few hooks to adjust the report definition, the printer and the way the report definition is loaded.

38.20.1. Customizing the printer

void customizePrinter(QPrinter* printer)

This signal is emitted when a printer output is initialized: after the engine has configured the `QPrinter` object from the page options of the report definition (paper size, orientation, full page mode, copies) and before the print dialog is shown. A slot connected to it can change any `QPrinter` setting, for example the duplex mode, the paper source or the color mode. Since the print dialog comes afterwards, the user can still override the settings in the dialog.

`NCReport` forwards this signal only for outputs created by `runReportToPrinter()` and `runReportToPrinterNoDialog()`. If you create an `NCReportPrinterOutput` (or the derived `NCReportPdfOutput`) yourself, connect to the signal of the output object, which has the same signature. When printing from the preview window, `NCReportPreviewWindow` emits its own `customizePrinter( QPrinter* )` signal.

```
connect( report, &NCReport::customizePrinter, this, [](QPrinter* printer) {
    printer->setDuplex( QPrinter::DuplexLongSide );
    printer->setColorMode( QPrinter::GrayScale );
    printer->setDocName( "Monthly invoices" );
});
report->runReportToPrinter( 1, true, this );
```



Don't change the printer resolution in this slot. The engine has already set its internal scaling from the printer resolution when the signal is emitted. Use `setPrintResolution()` (see [Section 38.21](#)) instead.

38.20.2. Modifying the report definition before running

After the report definition has been loaded (parsed into an `NCReportDef` object) and before rendering starts, the engine offers two hooks for modifying it. Both are called each time a definition is loaded, that is once per report run and once for every member report in batch mode.

void reportDefChange(NCReportDef* reportDef)

A signal emitted right after the definition has been loaded and the custom data (see `setCustomData()`) has been applied. It is emitted even if loading failed, so check `hasError()` in the slot before relying on the content of the definition.

virtual void customizeReportDef(NCReportDef* reportDef)

A protected virtual function called after `reportDefChange()`, only if the definition was loaded successfully. The default implementation does nothing. Subclass `NCReport` and reimplement it to modify the definition in code.

Changes made in these hooks affect the current run only, because the definition is reloaded from its source for every run. The page options are passed to the output after the hooks, so changes of the page settings (`reportDef->option()`) take effect.

```
#include "ncreport.h"
#include "ncreportdef.h"
#include "ncreportsection.h"

class MyReport : public NCReport
{
public:
    explicit MyReport( QObject* parent = nullptr ) : NCReport( parent ) {}
    bool landscape = false;
    bool hideReportHeader = false;

protected:
    void customizeReportDef( NCReportDef* rdef ) override
    {
        if ( landscape )
            rdef->option().setOrientation( NCReportPrinter::Landscape );

        if ( hideReportHeader && rdef->reportHeader() )
            rdef->reportHeader()->setShow( false );
    }
};

MyReport report;
report.setReportFile( "orders.ncr" );
report.landscape = true;
report.runReportToPdfWriter( "orders.pdf" );
```

38.20.3. Custom report sources

The report source loads the XML report definition (see `NCReportSource::ReportSourceType`). To load definitions from a place that the built-in types don't cover, for example from an application specific repository or an encrypted archive, subclass `NCReportSource` and reimplement its protected virtual `loadFromCustomSource( NCReportXMLDefReader& reader )` function (or any of the other `loadFrom...`() functions). Pass the XML text to `reader.read()`; on a parse error call `setParseError( reader )`, which sets the error message of the report.

The `NCReport` object creates its report source object internally. To make it create your class, set a report source factory: a subclass of `NCReportSourceFactory` that reimplements `virtual NCReportSource* createReportSource()`.

void setReportSourceFactory(NCReportSourceFactory* factory), NCReportSourceFactory* reportSourceFactory() const

Sets and returns the factory used to create the report source object. The `NCReport` object does not take ownership of the factory, but it deletes the report source object created by the factory when it is destroyed.

```
#include "ncreportsource.h"
#include "ncreportsourcefactory.h"
#include "ncreportxmldefreader.h"

class RepositoryReportSource : public NCReportSource
{
protected:
    bool loadFromCustomSource( NCReportXMLDefReader& reader ) override
    {
        const QString xml = Repository::instance()->reportXml( stringReportID() );
        bool ok = reader.read( nullptr, xml );
        if ( !ok )
            setParseError( reader );
        return ok;
    }
};

class RepositorySourceFactory : public NCReportSourceFactory
{
public:
    NCReportSource* createReportSource() override { return new RepositoryReportSource(); }
};

RepositorySourceFactory factory;    ①

NCReport report;
report.setReportSourceFactory( &factory );    ②
report.setReportSourceType( NCReportSource::Custom );
report.reportSource()->setStringReportID( "invoice" );
report.runReportToPreview();
```

① The factory must live as long as the report object uses it.

② Set the factory first, before any function that creates the report source (see the warning below).



The report source object is created only once, by the first function that needs it: `setReportFile()`, `setReportDefXml()`, `setReportSourceType()`, `setReportSource()` or `runReport()`. Set the factory before calling any of them; a factory set later is ignored.

38.20.4. Custom evaluator

void setEvaluator(NCReportEvaluator* evaluator), NCReportEvaluator* evaluator()

Sets a user defined evaluator. The evaluator is responsible for evaluating fields, variables, parameters, print-when conditions, dynamic items and script expressions. If an evaluator is set, the report engine uses it instead of creating its own `NCReportEvaluator` object. Subclass `NCReportEvaluator` and reimplement its virtual functions (for example `evaluate()`, `evaluateScript()`, `evaluatePrintWhen()` or `getDataSourceValue()`) to change how expressions are evaluated. The `NCReport` object does not take ownership: the evaluator must exist as long as the report uses it. Call

`setEvaluator( nullptr )` to return to the default evaluator.



For most needs, user functions are a simpler and safer way to add custom logic to a report than a custom evaluator. Reimplement evaluator functions only if you need to change the evaluation itself, and call the base class implementation for everything you don't handle.

38.21. Other runtime settings

The following functions of the `NCReport` class change how a report runs or return information about the last run.

void setCountry(QLocale::Country country)

Sets the country used together with the current report language. When a current language is set (with `setCurrentLanguage()` or in the report definition) and it is not the default language, the engine sets the default `QLocale` to that language and this country for the duration of the run and restores the previous default locale afterwards. The locale affects, for example, number and date formatting. Without a current language the country has no effect. The default is `QLocale::AnyCountry`. There is no getter and no separate `setLocale()` function.

void setCustomData(const QString& data), QString customData() const

Sets an arbitrary string that replaces the custom data (`customdata` page option) of the report definition each time the definition is loaded; an empty string leaves the value of the definition unchanged. The engine itself does not interpret the value. Your code can read it from the definition with `reportDef()->option().customData()`, for example in `customizeReportDef()`, in a slot connected to `reportDefChange()`, in a user function or in a custom data source.

void setZintPath(const QString& dir), QString zintPath() const

Sets and returns the directory of the Zint barcode program. On Windows the default value is detected from the registry (the installation path of `qtZint.exe`). The value is stored in the report definition object, but the current rendering engine does not use it: barcodes are rendered by the Zint library linked to NCReport. The function is kept for compatibility.

QPointF lastPagePositionMM() const, void setLastPagePositionMM(const QPointF& position)

`lastPagePositionMM()` returns the paint position, in millimeters from the top-left corner of the page, at the end of the last page of the last run. The vertical coordinate is useful for determining the printed height of endless (continuous paper) outputs. The engine sets the value at the end of every page with `setLastPagePositionMM()`; applications normally don't need to call the setter.

bool getParameters()

Asks the user for the values of the parameters whose value is the `{GET}` string, with an input dialog for each of them. If a parameter named `<id>_DEFAULT` exists, its value is used as the default value of the dialog. `runReport()` calls this function automatically before the report starts. If the user cancels a dialog, the function sets the error message `Parameter <id> input is cancelled by the user.` and returns `false`, and `runReport()` returns `false` without running the report. Set the parent widget of the dialogs with `setParentWidget()`. The function is available only if NCReport was built with the

preview module; otherwise it returns `true` without doing anything.

QVariant getParameter(const QString& id, const QVariant& value, QWidget* parent, const QString& title, const QString& label, bool* ok)

Shows a single input dialog for parameter `id` and, if the user accepts it, adds the entered value as a report parameter. The type of the default `value` determines the dialog: an integer input for `int`, a floating point input for `double`, and a text input otherwise. Returns the entered value and sets `ok` to `false` if the dialog was cancelled. Not available in builds without desktop support (`NCREPORT_NO_DESKTOP`).

```
report->addParameter( "customerId", "{GET}" );
report->addParameter( "customerId_DEFAULT", 1000 ); ①
report->setParentWidget( this );
report->runReportToPreview();
```

① The dialog asks for an integer, with 1000 as the default value.

void setForceCopies(int copies)

Runs the whole report `copies` times into the same output, one copy after the other, in a single job. Unlike the printer's copy count, the copies are rendered by the engine, so they also work with file outputs. In the report, the `forcecopies` system variable returns the number of copies and `currentcopy` returns the index of the current copy, starting from 0. The page number restarts for every copy. The default is 1.

void setPrintResolution(int dpi), int printResolution() const

Sets the print resolution of printer and file outputs. The value is stored in the page options of the report definition object (the same option as **Resolution** in the Designer's page settings), and the outputs except the preview outputs apply it when the report runs. When `runReport()` loads the definition, a resolution stored in the definition replaces the value set with this function; if the definition has no resolution (**Auto** in the Designer), the value set here is kept. `printResolution()` returns the current value of the option, so after a run it may return the resolution of the last loaded definition. A value of `-1` means that the default resolution of the output is used.

void setPrintQuality(NCReportPrinter::PrinterMode mode), NCReportPrinter::PrinterMode printQuality() const

Sets the `QPrinter` mode (`ScreenResolution`, `PrinterResolution` or `HighResolution`, the default) used when the one-step functions create a printer, PDF, PostScript or Qt preview output. It has no effect on output objects that you create yourself.

void setPageRanges(const QString& ranges), QString pageRanges() const

Restricts the output to the given pages. The expression is a comma-separated list of page numbers and ranges, for example `1-3,5,8-10`. The expression is passed to the output at every run, including outputs that you create and set with `setOutput()`.



`runReport()` always passes the `NCReport` page range expression to the output, and an empty expression clears the page list of the output. A page range set only on the output object with `NCReportOutput::setPageRanges()` is therefore lost when the report runs.

Use `NCReport::setPageRanges()` instead.

int runTimeElapsed() const

Returns the time, in milliseconds, spent on rendering in the last run. The measurement starts after the definition has been loaded and the output initialized (so a print dialog is not included) and includes finishing the output. In batch mode the value belongs to the last member report.

int pageCount() const, bool reportDone() const

`pageCount()` returns the number of pages of the last run. `reportDone()` returns `true` if the last run finished without a processing failure. Note that `reportDone()` does not reflect cancellation; see [Section 38.19](#).

ReportType reportType() const

Intended to return the type of the report (`NCReport::Report`, `NCReport::TextDocumentPrint` or `NCReport::Label`). In the current version the underlying value is never updated, so the function always returns `NCReport::Report`. To get the actual type, use `reportDef()->option().reportType()` after the definition has been loaded.

void setAlternateRowBackgroundColor(const QColor& color), QColor alternateRowBackgroundColor() const

Sets the default background color of alternating rows. It is used by detail sections that have alternating row background enabled (`altBackRowMode` set to **odd** or **even**) and use the default color (`useDefaultAltBackRowColor`, which is the default). Sections with their own alternating row color are not affected. The default color is light gray, RGB(220, 220, 220).

Part VI: Specification

This part is the specification of the NCReport report definition (XML template) file format. It describes every element and attribute of the report definition file.

Chapter 39. Specification

Report XML template structure specification.

A report template is a single UTF-8 XML file. Its root element is `<NCReport>`, which contains the page options, data sources, parameters, variables, scripts, and the report sections. This chapter describes the structure of that file element by element, together with the attributes and values that the report engine recognizes.

The overall skeleton of a report definition file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<NCReport version="2.0" name="myreport" type="report">
  <options>...</options>
  <datasources>...</datasources>
  <parameters>...</parameters>
  <variables>...</variables>
  <scripts>...</scripts>
  <pageheader>...</pageheader>
  <reportheader>...</reportheader>
  <details>...</details>
  <reportfooter>...</reportfooter>
  <pagefooter>...</pagefooter>
</NCReport>
```

39.1. Root element

The `<NCReport>` element is the root of every report definition. All other elements are its children.

39.1.1. XML syntax

```
<NCReport version="2.0" name="..." type="report">...</NCReport>
```

39.1.2. Tag properties

Attribute	Values	Default	Description
version	number		The template format version. The report engine requires version 2.0 or higher; older files are rejected as incompatible.
name	text		The report name.
type	report, textdoc, label	report	The report type: report (the default paged report), textdoc (text document print mode), label (label sheet report; see Chapter 24).

39.2. Page options

The `<options>` element holds the global page and report settings. Unlike most other elements, each option is stored as its own child element whose text content carries the value. All options are optional; when omitted, the engine uses its built-in default.

39.2.1. XML syntax

```
<options>
  <encoding>UTF-8</encoding>
  <orientation>portrait</orientation>
  <pagesize width="210" height="297">A4</pagesize>
  <topmargin>15</topmargin>
  <bottommargin>15</bottommargin>
  <leftmargin>20</leftmargin>
  <rightmargin>20</rightmargin>
  <fontname>Helvetica</fontname>
  <fontsize>10</fontsize>
  ...
</options>
```

39.2.2. Elements

Element	Values	Default	Description
encoding	encoding name	UTF-8	Character encoding of the report.
language	language codes	00	The list of language codes used by the report for multi-language templates.
langrole	0, 1	0	Behavior when a translation is missing. 0 uses the default language, otherwise the value is left empty.
orientation	portrait, landscape	portrait	Page orientation.
pagesize	paper size name; width, height attributes in mm	A4 (210 x 297)	Paper size. The element text is the paper size name (for example A4), and the <code>width</code> and <code>height</code> attributes hold the custom size in millimeters (used when the name denotes a custom size).
topmargin	number (mm)	15	Top page margin in millimeters.
bottommargin	number (mm)	15	Bottom page margin in millimeters.
leftmargin	number (mm)	20	Left page margin in millimeters.
rightmargin	number (mm)	20	Right page margin in millimeters.
backcolor	#RRGGBB color	#ffffff	Page background color in #RRGGBB format.
columns	integer	0	Number of page columns. The value is stored in the report definition but is currently not used by the report engine.

Element	Values	Default	Description
columnspace	number (mm)	0	Space between page columns in millimeters. Stored but currently not used by the report engine.
fontname	font family	Helvetica	Name of the default report font.
fontsize	integer (pt)	10	Size of the default report font in points.
fontstyle	text		Style of the default report font.
zoneSpacing	number (mm)	0	Vertical spacing between zones in millimeters. See the Zones feature.
zoneBreak	true, false	false	If true , zones are allowed to break across pages.
doublePass	true, false	false	If true , the report is rendered in two passes (double-pass mode), for example when the total page count is required.
subReportOnNewPage	true, false	false	If true , each sub-report starts on a new page.
subReportResetPageNo	true, false	false	If true , the page number is reset for each sub-report.
htmlBodyStyle	CSS text	{DEFAULT}	CSS style applied to the HTML body when producing HTML output. (<code>html10pt</code> is also accepted for backward compatibility.)
htmlHead	HTML text		Content inserted into the HTML <code><head></code> element of HTML output.
customdata	text		Arbitrary custom data stored with the report definition.
dataTrimming	true, false	true	Record trimming for correct totals. When enabled (the default), the engine steps back one record at page breaks and group footers, so that the variable values shown in page footers, page headers, and group footers are correct. Set this to false to disable it. See Chapter 12 .
updateOnlyCurrentDataSourceVariables	true, false	false	If true , only the variables that belong to the current data source are updated.
varErrorAlert	true, false	false	If true , the engine alerts on variable evaluation errors.
tocGen	true, false	false	If true , a table of contents is generated.
tocReset	true, false	false	If true , the table of contents is reset.
tocFile	file name		File name for the generated table of contents.

Element	Values	Default	Description
tocSectionTypes	comma-separated integers		Comma-separated list of section type codes included in the table of contents.
labelDirection	horizontal, vertical	horizontal	Fill direction for label reports.
labelColumnCount	integer	0	Number of label columns.
labelRowCount	integer	0	Number of label rows.
labelHGap	number (mm)	0	Horizontal gap between labels in millimeters.
labelVGap	number (mm)	0	Vertical gap between labels in millimeters.
labelWidth	number (mm)		Label width in millimeters.
labelHeight	number (mm)		Label height in millimeters.
endless	true, false	false	If true , the report uses an endless (continuous, non-paginated) page.
dpi	integer (dpi)		Print resolution in dots per inch.
emptyValue	text		The value displayed for empty (null) data.
fullPage	true, false	true	Full-page mode. On by default; set to false to respect the printer's non-printable margins.
outputBaseName	file name		Base file name used for the generated output file.

39.3. Data sources

Since the report generator builds a printable representation from data, the data source is one of the most important parts of the system. Data may come from an SQL query using Qt's SQL database connection drivers, or from other sources: **Text**, **XML**, **String list**, **Item model**, or a **custom** data source written in Qt/C++. One report can contain multiple data sources, and each detail can be connected to its own data source. Whenever a data source is not assigned to any detail, the initial (first) row of data in the data source is used.

All data source definitions are wrapped in a single `<datasources>` element, and each individual source is a `<datasource>` child element. The attributes listed below are common to every data source type; type-specific attributes follow in each subsection.

39.3.1. Common tag properties

Attribute	Values	Default	Description
id	text		Data source ID. Identification name of the data source. Details are assigned to the data source by this ID.

Attribute	Values	Default	Description
type	sql, txt, xml, list, model, custom		The data source type. Any other value is rejected as an invalid data source type.
source	static, file, http, ftp, parameter	static	The location of the data source definition. Depending on this value, the definition is read from the report file, an external file, a URL, or a parameter.
openRole	begin, child, group, sub	begin	When the data source is opened/updated: begin (opened once at the beginning of the report), child (a child data source updated by its parent through a data relation), group (updated on a group change), sub (a sub-report data source).
parentID	data source ID		If the data source is a child in a data relation (sub-query system), this is the ID of the parent data source.
enabled	true, false	true	Whether the data source is enabled.
allowEmpty	true, false	true	Whether the data source is allowed to return no rows. (The legacy <code>emptyIsForbidden</code> attribute is the inverse of this value.)
primaryKeyColumn	integer	0	The zero-based index of the primary-key column, used by the data relation system.
headers	comma-separated names		A comma-separated list of predefined column header names.
briefDesc	text		A short, human-readable description of the data source.

39.3.2. SQL data source

SQL queries (the SQL data source) are the most common and widely used data sources for NCReport. They require an SQL database connection using Qt's database driver plugins. The database connection can be either internal or external.

For an internal connection (the default), a valid database connection must be established by the application using NCReport before running the report. If an external connection is used, the connection parameters must be specified accordingly.

XML syntax

```
<datasource type="sql">[SQL query]</datasource>
<datasource type="sql" source="file">[query filename]</datasource>
```

Additional tag properties

Attribute	Values	Default	Description
connection	internal, external	internal	Specifies the SQL database connection handling method. With an internal connection (the default), a valid database connection must be established by the application using NCReport before running the report. If an external connection is specified, the report generator connects to the database when opening the data source. In this case, the host, database, user, password, port connection parameters are used. (The values <code>global</code> and <code>private</code> are also accepted as synonyms of <code>internal</code> and <code>external</code> .)
connID	connection name		The name of the database connection that is used when the QSqlDatabase::addDatabase(...) method is called in the report engine. This ID is required for running the SQL query assigned to the data source.
connOpt	text		Extra connection options passed to the database driver (equivalent to QSqlDatabase::setConnectOptions()).
driver	QDB2, QIBASE, QMYSQL, QOCI, QODBC, QPSQL, QSQLITE2, QSQLITE, QTDS		The name of the Qt SQL database driver.
host	host name		Host name for the SQL database connection. Used only when an external connection is defined.
database	text		Database name for the SQL database connection. Used only when an external connection is defined.
user	text		User name for the SQL database connection. Used only when an external connection is defined.
password	text		Password for the SQL database connection. Used only when an external connection is defined.
port	integer		Port number for the SQL database connection. Used only when an external connection is defined.
recordFilter	text		An optional filter applied to the fetched records.

Attribute	Values	Default	Description
forceUpdateOnStepBackRel	true, false	false	If true , the data source is re-queried when a data relation steps back to a previous parent row.
fwOnly	true, false	false	If true , the query is treated as forward-only, which can improve performance and reduce memory use for large result sets.

39.3.3. Text data source

Text and text files can be used as a data source for NCReport. The data columns of a text are usually delimited by a tab or another delimiter character. This kind of data source even makes it possible to avoid an SQL database connection. You must set the delimiter type; this delimiter separates the columns, and each row represents a data record. Text data sources can be static, stored in the XML definition, or a file.

XML syntax

```
<datasource type="txt">[static text]</datasource>
<datasource type="txt" source="file">[filename]</datasource>
```

39.3.4. XML data source

The Extensible Markup Language (XML) format can also be used as a data source for NCReport.

The following example shows what the data structure should look like:

Example

```
<textobject><textdata fileref="codes/xmltestdata.xml" /></textobject>
```

XML syntax

```
<datasource type="xml">...</datasource>
```

39.3.5. String list data source

It is also possible to use a QStringList as a data source for NCReport. Before running the report, a QStringList must be assigned to the specified data source, and it is also necessary to set a delimiter character to separate the columns in each list item, which represents a data record.

XML syntax

```
<datasource type="list"></datasource>
```

Additional tag properties

Attribute	Values	Default	Description
listID	text		ID of the list. This identification name specifies the ID of the <code>QStringList</code> added to NCReport.

39.3.6. Item model data source

Qt's item model classes provide a generic model for storing custom data. For example, `QStandardItemModel` can be used as a repository for standard Qt data types. It is one of the Model/View classes and is part of Qt's model/view framework. It is possible to use item models as a data source for NCReport. Before running the report, a `QAbstractItemModel`-based class must be created and assigned to the specified data source.

XML syntax

```
<datasource type="model"></datasource>
```

Additional tag properties

Attribute	Values	Default	Description
modelID	text		ID of the model. This identification name specifies the ID of the model added to NCReport.

39.3.7. Custom data source

Data is often stored in a special repository such as a list, array, etc. You can build your own data source class derived from the `NCReportDataSource` base class. It is an abstract class; you just have to implement the required methods.

XML syntax

```
<datasource type="custom"></datasource>
```

39.3.8. Data source relations

A child data source can be driven by a parent (master) data source, which is the basis of the master/detail (sub-query) system. The relation is established with two attributes: the child data source sets `openRole="child"` and points to its parent with `parentID`. The parent's primary-key column and the

child's foreign-key column are matched so that, as the parent steps from row to row, the child data source is re-queried for the matching records. This relation is supported for SQL and Item Model data sources. See the *Data Relation system* chapter for a full description.

39.4. Report Parameters

Parameters are data pulled from outside the report generator. The application that calls the NCReport object passes information as parameters to the NCReport class via the `addParameter(...)` method. Parameters are evaluated within SQL queries, fields, and script expressions. Field objects may have a parameter data source type, so they can be presented as data in the report. Parameters are mostly used in queries. If you want to embed a parameter into a query or an expression, use this syntax:

```
$P{parametername}
```

Example of using a parameter in an SQL query:

```
SELECT productId, productName FROM db.products WHERE primaryKey=$P{parametername}
```

Parameters may also be stored statically in the report definition inside the `<parameters>` element. Each parameter is a `<par>` element (the long form `<parameter>` is also accepted), whose text content is the value and whose `type` attribute declares the data type.

39.4.1. XML syntax

```
<parameters>
  <par id="date_from" type="date">2026-01-01</par>
  <par id="limit" type="int">100</par>
  <par id="title" type="string">Sales report</par>
</parameters>
```

39.4.2. Tag properties

Attribute	Values	Default	Description
id	text		The name/ID of the parameter.
type	string (or QString), int, bool, float, double (or numeric), long, longlong, date (or QDate), datetime (or QDateTime)	string	The data type of the parameter value. If the type is unknown, the value is treated as a string.

39.5. Variables

Variables are specific items of the report. They are special fields used for providing counts and totals. Each variable has a name, a function type, a data type, and an assigned data source column that the

variable is based on.

Variables are defined inside the `<variables>` element, one `<variable>` element each. The text content of the element is the expression: either a plain data source column name (`price` or `ds.price`) or a script expression enclosed in `<% %>`. Do not use the `$D{}` syntax here.

39.5.1. XML syntax

```
<variables>
  <variable id="sumPrice" type="num" funcType="sum" scope="group" initValue="0">price</variable>
</variables>
```

39.5.2. Tag properties

Attribute	Values	Default	Description
id	text		The name/ID of the variable.
type	num	num	The data type of the variable. The writer stores num ; the attribute is ignored when reading, since variables are always numeric.
funcType	count, sum, average, std, min, max	count	The aggregation function (see the function types below). Unknown values fall back to count .
scope	report, page, group	report	The scope at which the variable is reset. Unknown values fall back to report .
initValue	number	0	The initial (reset) value of the variable.

The function types have the following meaning:

count

Increments by 1 for every row returned by the data source.

sum

Sums the values of the specified data source column. The values are converted to numbers.

average

Computes the average (mean) of the specified column's values.

std

Computes the standard deviation of the specified column's values.

min

Returns the smallest value of the specified column.

max

Returns the largest value of the specified column.

To embed a variable into an expression, use this syntax:

```
$V{variablename}
```

39.6. System Variables

System variables are special variables that provide report system information such as the page number, the current date/time, etc. for fields. The most frequently used system variables are listed below. For the complete list, see [Section 12.5](#).

Variable	Description
pageno	Returns the current page number
pagecount	Returns the total number of pages in the report. Available in double pass mode and in Text document printout mode.
forcecopies	Returns the total number of forced copies
currentcopy	Returns the current forced copy number (zero-based)
currentrow	Returns the current detail row number
date	Returns the current date
time	Returns the current time
datetime	Returns the timestamp
appname	Returns the name of this application
applongname	Returns the long name of this application
appinfo	Returns the full info string of this application
appversion	Returns the version of this application
appcopyright	Returns the copyright info of this application
qtversion	Returns the Qt version
os	Returns the operating system

For variable fields, or to embed a system variable into an expression, use this syntax:

```
$V{systemvariablename}
```

39.7. Scripts

Named scripts are reusable script functions stored in the report definition. A script can be referenced from expressions (see *Data/Script Expressions*) by its ID using the `$S{scriptID}` syntax, which keeps complex or repeated logic in one place. Scripts are held in the `<scripts>` element, one `<script>` element each; the element text is the script code.

39.7.1. XML syntax

```
<scripts>
  <script id="ageGroup"><![CDATA[ if ($D{age}<18) "MINOR"; else "ADULT"; ]]></script>
</scripts>
```

39.7.2. Tag properties

Attribute	Values	Default	Description
id	text		The name/ID of the script, used to reference it as <code>\$S{id}</code> .

39.8. Expressions

Since version 2.0, NCReport handles script expressions using Qt Script, the powerful feature introduced in Qt 4.3. Qt Script is based on the ECMAScript scripting language, as defined in the ECMA-262 standard. Fields and group expressions may be script code instead of a data source column. The report engine evaluates the specified script code each time the fields are refreshed. Report items can have a **printWhen** property. These are also script expressions that return a boolean result. To use a script expression in a field, the `ftype="exp"` field property must be specified.

39.9. References in expressions

Expressions can contain and evaluate references such as:

- data source data
- parameter
- variable

References are always replaced with their current value before the expression is evaluated. The syntax of the references is as follows: `$D{[data source.]column[,n]}` data source column reference. Returns the current value of the data source column from the current row/record. If data source. is not specified, the current data source (assigned to the current detail) is used.

If *n* is specified, the data source is first positioned to the *n*th record. Works only if the `::seek( int )` method is defined in the appropriate data source class.

\$P{parametername}

Parameter reference. Returns the value of the parameter by name/ID.

\$V{variablename}

Variable reference. Returns the current value of the variable by name/ID.

\$S{scriptID}

Script reference. Evaluates the named script and returns its result.

39.10. Using a script expression in a field:

```
"$D{db.productName}"+ " "+"some string"+"$P{parametername}"
```

Using a script expression in the **printWhen** property:

```
$D{price}<1500
```

Quotation marks in expressions are required only when string data is used. Otherwise (i.e., for numbers) quotation marks are not necessary.

39.11. Report sections

Report sections are the representations of the function-specific areas inside the report. The whole report is built from sections. They are often recurring areas such as the detail, headers, and footers. The most important section is called Detail, since details can contain the fields that change row by row. Each section can contain all kinds of report items. An item's coordinates are always relative to its parent section.

One report can contain the following sections: report header, report footer, page headers, page footers, group headers and footers, and details.

39.11.1. Common section properties

The following properties may appear on any section element. Section-specific properties are listed with each section below.

Attribute	Values	Default	Description
id	text		Name/ID of the section, for identification purposes.
height	number (mm)		The height of the section in millimeters.
autoHeight	true, false	false	If true , the section height is calculated automatically from its content instead of using the fixed height .
printWhen	script expression		A logical script expression that controls whether the section is printed. If it is not empty, the engine evaluates it before rendering; the section is printed only when the expression returns true.
startsOnNewPage	true, false	false	If true , the section starts on a new page.
pageBreakExp	script expression		A script expression that, when it returns true, forces a page break at the section.

Attribute	Values	Default	Description
repeatExp	expression		A repeat expression; while it holds, the section is repeated (see repeated detail sections).
notAlone	true, false	false	If true , the section is not printed alone at the bottom of a page (keeps it together with related content).
orphanedTolerance	number (mm)	0	The tolerance, in millimeters, used to avoid orphaned sections near a page break.
hideIfBlank	true, false	false	If true , the section is hidden when it has no content.
anchorToPageFooter	true, false	false	If true , the section is anchored to the page footer.
altBackRowColor	#RRGGBB color	#dcdcdc	The alternating row background color in #RRGGBB format.
altBackRowMode	disabled, odd, even	disabled	The alternating background mode.
useDefaultAltBackRowColor	true, false	true	If true , the default alternating row color is used.
parentId	section ID		The ID of the parent section, used to build nested (child) detail sections.

39.11.2. Page header

Page headers are used to contain page headings. Page headers have the following characteristics:

- Always printed at the top of a page
- Always contain the first information printed on a page
- Display only one (current) row of data returned by a data source
- Only one allowed per page

XML syntax

```
<pageheader>...</pageheader>
```

39.11.3. Page footer

Page footers are commonly used to close the pages. Page footers have the following characteristics:

- Always printed at the bottom of a page
- Display only one (current) row of data returned by a data source
- Only one allowed per page

- Usually used to display information such as the page number, report titles, and so on.

XML syntax

```
<pagefooter>...</pagefooter>
```

39.11.4. Report header

The report header is used to contain report headings. The report header has the following characteristics:

- Always printed after the page header
- Printed only once, at the beginning of the report
- Displays only one (current) row of data returned by a data source

XML syntax

```
<reportheader>...</reportheader>
```

39.11.5. Report footer

The report footer is commonly used to close the report. The report footer has the following characteristics:

- Always printed before the page footer, at the end of the report
- Displays only one (current) row of data returned by a data source
- Only one allowed per report

XML syntax

```
<reportfooter>...</reportfooter>
```

39.11.6. Details

The core information in a report is displayed in its Detail section. This section is the most important part of the report, since it contains the row-by-row data from the data source. Detail sections have the following characteristics:

- Generally printed in the middle of a page
- Always contain the core information for a report
- Display multiple rows of data returned by a data source

- Generally contain fields or dynamic objects
- Multiple independent details are allowed in one report, each detail after the other
- Each detail is assigned to one specified data source

XML syntax

```
<detail>...</detail>
```

Structure:

```
<details>
  <detail datasource="ds1">
    <items>...</items>
    <groups>...</groups>
  </detail>
  <detail datasource="ds2">
    <items>...</items>
    <groups>...</groups>
  </detail>
  ...
</details>
```

Tag properties

Attribute	Values	Default	Description
datasource	data source ID		The name/ID of the data source the detail section is assigned to.
datasource2	data source ID		The name/ID of an optional secondary data source for the detail section.

(The common section properties above, such as `id` and `height`, also apply.)

39.11.7. Group sections

While most reports can be defined using a single Detail section with multiple columns and rows of data, others require summary data, such as subtotals. For reports requiring summary data, the report writer supports Group sections. Group sections have the following characteristics:

- Always associated with a Detail section
- Defined by Group Headers and Group Footers
- Group Headers always print above their Detail section
- Group Footers always print below their Detail section
- Reference the database column on which Group Headers and Group Footers will break
- Force a new Group Header each time the value of the referenced column changes

- Force a new Group Footer each time the value of the referenced column changes
- Unlimited number of group levels allowed

The groups added to the XML definition are shown in the order in which you added them. They are structured hierarchically. The first group is the primary group level, the second one is the second level, and so on. The added group sections appear in the designer after you apply the group settings.

XML syntax

```
<groups>
  <group id="g1" groupExp="$D{category}">
    <groupheader>...</groupheader>
    <groupfooter>...</groupfooter>
  </group>
</groups>
```

Tag properties

Attribute	Values	Default	Description
id	text		Identification label for naming the group.
groupExp	expression, column name, or <code>%CONST</code>		Group expression or data source column. Specifies the name of the data source column on which Group Headers and Group Footers will break. The expression can also be a constant value; in this case the detail rows do not break. The constant group expression is <code>%CONST</code> .
resetVariables	comma-separated variable IDs		A comma-separated list of variable IDs to reset when the current group ends. This is very useful when, for example, you want to reset a total or a count variable at each group change.
reprintHeader	<code>true</code> , <code>false</code>	<code>false</code>	If true , the group header is reprinted at the top of the next page when the group spans a page break.
startsOnNewPage	<code>true</code> , <code>false</code>	<code>false</code>	If true , each new group starts on a new page.
pageBreakExp	script expression		A script expression that forces a page break within the group when it returns true.
resetPageNo	<code>true</code> , <code>false</code>	<code>false</code>	If true , the page number is reset at each group change.
printLock	<code>true</code> , <code>false</code>	<code>true</code>	Controls whether print locking is allowed for the group. Set to false to disallow it.

39.11.8. Group header

Group headers are used to contain group heading items such as column head titles. They are always printed above their Detail section. A new Group Header is forced each time the value of the referenced column changes.

XML syntax

```
<groupheader>...items...</groupheader>
```

39.11.9. Group footer

Group footers are used to contain group footing items such as totals and subtotals. They are always printed below their Detail section. A new Group Footer is forced each time the value of the referenced column changes.

XML syntax

```
<groupfooter>...items...</groupfooter>
```

39.12. Report items

Report items are the visual building blocks placed inside sections. The following item types are available: **label**, **field**, **text** (HTML text), **line**, **rectangle**, **ellipse**, **image**, **barcode**, **graph** (custom item), **crosstab**, **table** (item-view table), **graphicsview**, and **chart**.

39.12.1. Common item properties

Most item types share a set of common properties. They are listed here once; the sections that follow describe only the properties that are specific to each item.

Attribute	Values	Default	Description
id	text		Item ID for internal identification.
posX	number (mm)		The item's X coordinate in millimeters, relative to the current section.
posY	number (mm)		The item's Y coordinate in millimeters, relative to the current section.
width	number (mm)		The item's width in millimeters.
height	number (mm)		The item's height in millimeters.
zValue	integer	0	An integer Z-order value that determines the stacking order of sibling (neighboring) items.
resource	item-specific		The source of the item's content. The accepted values depend on the item type (see each item below).
printWhen	script expression		A logical script expression that controls the item's visibility. If it is not empty, the engine evaluates it before each rendering; the item is shown when the expression returns true (or 1) and hidden otherwise.

Attribute	Values	Default	Description
template	true, false	false	If true , the item's content is treated as a template, so data references are embedded directly in the text without a script.
canBreak	true, false	false	If true , the item is allowed to break across a page boundary.
autoHeight	true, false	false	If true , the item's height is calculated automatically from its content.
pageBreakAfter	true, false	false	If true , a page break is forced after the item.
clickable	true, false	false	If true , the item is clickable in interactive output.
description	text		A free-text description of the item.
zoneID	integer	0	The zone the item belongs to. Items that share a zone ID form a virtual band inside the section (see the Zones feature).
pinToLeft	true, false	false	If true , the item is pinned to the left edge.
pinToRight	true, false	false	If true , the item is pinned to the right edge.

Dynamic position and size. Items support data-driven position and size. When one of these properties is set, its value is an expression (a data source column, parameter, variable, or script) that supplies the measurement in millimeters. When left empty, the corresponding static value is used.

Attribute	Values	Default	Description
dynPosX	expression		The source of the item's X position.
dynPosY	expression		The source of the item's Y position.
dynWidth	expression		The source of the item's width.
dynHeight	expression		The source of the item's height.
dynOffset	true, false	false	If true , the dynamic values are relative to the item's original static position (offset mode); otherwise they are explicit coordinates.

Font properties. Text-bearing items (label, field, HTML text) support the following font properties.

Attribute	Values	Default	Description
fontName	font family		Font family / face name. When omitted, the report's default font (<code>fontname</code> page option) is used.
fontSize	number (pt)		Font size in points.
fontWeight	light, normal, demibold, bold, black	normal	Font weight.
fontItalic	true, false	false	If true , the font is italic.
fontUnderline	true, false	false	If true , the text is underlined.
fontStrikeOut	true, false	false	If true , the text is struck out.

Attribute	Values	Default	Description
letterSpacing	number		Absolute letter spacing.
capital	integer		Capitalization mode (numeric code, following Qt's <code>QFont::Capitalization</code>).

Alignment. Alignment can be given either as a single numeric `alignment` value (a combination of Qt alignment flags) or with the separate horizontal and vertical attributes below (kept for backward compatibility).

Attribute	Values	Default	Description
alignment	integer (Qt flags)		Numeric combination of Qt alignment flags.
alignmentH	<code>left</code> , <code>right</code> , <code>center</code>		Horizontal alignment.
alignmentV	<code>top</code> , <code>center</code> , <code>bottom</code>		Vertical alignment.

Line (pen) and fill (brush) properties. Shape items (line, rectangle, ellipse) and bordered items use these.

Attribute	Values	Default	Description
lineColor	#RRGGBB color		Line/outline color in #RRGGBB format.
lineWidth	number (mm)		Line/outline width in millimeters.
lineStyle	<code>solid</code> , <code>dash</code> , <code>dot</code> , <code>dashdot</code> , <code>dashdotdot</code> , <code>nopen</code>	<code>solid</code>	Line drawing style. (no and none are also accepted for no line.)
fillColor	#RRGGBB color		Fill color in #RRGGBB format.
fillStyle	<code>no</code> , <code>solid</code> , <code>dense1</code> ... <code>dense7</code> , <code>hor</code> , <code>ver</code> , <code>cross</code> , <code>bdiag</code> , <code>fdiag</code> , <code>diagcross</code>	<code>no</code>	Fill (brush) pattern. The dense patterns are <code>dense1</code> , <code>dense2</code> , <code>dense3</code> , <code>dense4</code> , <code>dense5</code> , <code>dense6</code> , and <code>dense7</code> .

39.12.2. Text label

The Label represents simple text or label items. Label items are used to display descriptive information in a report definition, such as titles, headings, etc. Labels are static items; their values do not change when rendering the report.

XML syntax

```
<label>Text label...</label>
```

Tag properties

Attribute	Values	Default	Description
forecolor	#RRGGBB color		The foreground (text) color in #RRGGBB format.
backcolor	#RRGGBB color		The background color in #RRGGBB format.

Attribute	Values	Default	Description
bgMode			The background drawing mode.
wordbreak	true, false	false	If true , long text wraps at word boundaries.
wrapanywhere	true, false	false	If true , text may wrap anywhere, not only at word boundaries.
elide	none, left, mid, right	none	How text is elided (shortened with an ellipsis) when it does not fit.
rotation	integer (degrees)	0	Text rotation in degrees.
hideRepeated	true, false	false	If true , the value is hidden when it repeats the previous displayed value. This option takes effect for fields only; see Section 15.3 .
fillchar	character	(space)	The fill character used for padding.
lang	language code		The language code of the label text (for multi-language reports).
tr	true, false		If true , the label text is marked for translation.

(The common item properties, including the font and alignment properties, also apply.)

39.12.3. Field

The field is the core report item. It represents data field objects. By data type, fields may be text, numeric, or date. Field items are used to pull dynamically generated data into a report from the specified data source, such as the database the report generator uses. For example, a field item may be used to present SQL data, variables, and parameters. NCReport handles data formatting for the different types of fields, such as numbers and text.

XML syntax

```
<field>[data sourcename.]column</field>
<field>[expression]</field>
<field>[parametername]</field>
<field>[variablename]</field>
<field>[system variablename]</field>
```

Tag properties

Attribute	Values	Default	Description
type	txt, num, date, bool		The field's base data type: txt (text data), num (numeric data — the numeric formatting options apply only to this type), date (date data — the date formatting options apply only to this type), bool (boolean data, rendered as Yes/True or No/False).

Attribute	Values	Default	Description
<code>fType</code>	<code>ds</code> (or <code>sql</code>), <code>par</code> , <code>var</code> , <code>sys</code> , <code>exp</code> , <code>tpl</code>		The field's source type; it determines where the field's value comes from: ds (or sql — from the default or specified data source), par (from a parameter), var (from a variable), sys (from a system variable), exp (evaluates a script expression), tpl (a template that embeds data references in text). Unknown values are treated as ds .
<code>formatting</code>	<code>true</code> , <code>false</code>	<code>false</code>	If the field's data type is numeric, this option controls whether number formatting is applied.
<code>numwidth</code>	integer	0	The minimum field width, in digits, that the value is padded to using <code>fillchar</code> . A positive value produces right-aligned text; a negative value produces left-aligned text. Works only when number formatting is on.
<code>format</code>	<code>e</code> , <code>E</code> , <code>f</code> , <code>g</code> , <code>G</code>	<code>f</code>	A one-character numeric format code. With e , E , and f , <code>precision</code> is the number of digits after the decimal point; with g and G , <code>precision</code> is the maximum number of significant digits. Used by the <code>QString::arg(double a, int fieldWidth, char format, int precision, const QChar &fillChar)</code> function.
<code>precision</code>	integer	-1	The number of digits after the decimal point for numeric data.
<code>fillchar</code>	character	(space)	The padding character used together with <code>numwidth</code> .
<code>numChars</code>	integer	0	If greater than zero, limits the number of characters taken from the value.
<code>dateFormat</code>	date format string		Date formatting expression, using the same format as <code>QDate::toString()</code> . Works only when the field type is date.
<code>localized</code>	<code>true</code> , <code>false</code>	<code>false</code>	Whether localization is applied. Works for numeric data only.
<code>blankifzero</code>	<code>true</code> , <code>false</code>	<code>false</code>	If true , the field is left blank when its value is zero.
<code>ssifzero</code>	<code>true</code> , <code>false</code>	<code>false</code>	If true , a substitution string is displayed instead of the value when the value is zero.
<code>ssifzero_str</code>	text		The substitution string displayed when <code>ssifzero</code> is on and the value is zero.
<code>usedsapproles</code>	<code>true</code> , <code>false</code>	<code>false</code>	If true , the field takes its appearance (colors, font) from the data source's appearance roles (for example, Qt item-model roles).

Attribute	Values	Default	Description
arg	template string		A <code>QString::arg(...)</code> template string. The field's value replaces the first <code>%1</code> occurrence in this string. A <code>%</code> may be followed by an <code>L</code> to produce a localized representation, using the default locale set by <code>QLocale::setDefault()</code> (or the "C" locale if none was set).

(The common item properties, including the font and alignment properties, also apply.)

39.12.4. HTML Text

HTML Text represents rich text in HTML format.

XML syntax

```
<text>Static (encoded) HTML text</text>
<text>[data source].column</text>
<text>[filename]</text>
```

Tag properties

Attribute	Values	Default	Description
forecolor	#RRGGBB color		The foreground color in #RRGGBB format. Takes effect only if the font settings are enabled for the item.
useFont	true, false		If true , the item's own font settings are applied to the text.
markdown	true, false		If true , the content is interpreted as Markdown rather than HTML.
plain	true, false		If true , the content is treated as plain text.
editorMode	0, 1		The editor mode used for the text content: 0 selects the rich-text editor, any other value the source-code editor.

(The common item properties, including the font properties, also apply.)

39.12.5. Line

The Line option enables you to create Line items. In general, Line items are used to draw vertical and horizontal lines for headings, to underline titles, and so on. Lines are defined by their start and end point coordinates.

XML syntax

```
<line></line>
```

Tag properties

Attribute	Values	Default	Description
fromX	number (mm)		X coordinate of the start point of the line in millimeters, relative to the current section.
fromY	number (mm)		Y coordinate of the start point of the line in millimeters, relative to the current section.
toX	number (mm)		X coordinate of the end point of the line in millimeters, relative to the current section.
toY	number (mm)		Y coordinate of the end point of the line in millimeters, relative to the current section.

(The common line/pen properties — `lineColor`, `lineWidth`, `lineStyle` — and the common item properties also apply.)

39.12.6. Rectangle

The Rectangle option enables you to create Rectangle items. Rectangles are usually used to draw boxes or borders around a specified area. A rectangle makes box drawing easier than drawing four separate lines.

XML syntax

```
<rectangle></rectangle>
```

Tag properties

Attribute	Values	Default	Description
roundX	integer	0	Horizontal corner rounding radius.
roundY	integer	0	Vertical corner rounding radius.

(The common line/pen and fill/brush properties, together with the common item properties, also apply.)

39.12.7. Ellipse

The Ellipse item draws an ellipse or a circle. Like the rectangle, it uses the common line (pen) and fill (brush) properties.

XML syntax

```
<ellipse></ellipse>
```

Tag properties

(The common line/pen and fill/brush properties, together with the common item properties, apply.)

39.12.8. Image

The Image option enables you to create Image items. Image items are used to insert either static or dynamic images into a report definition. Static images, such as a company logo often displayed in the Report Header, can be loaded from a static file or from the report definition. Dynamic images can be loaded from the specified SQL data source.

XML syntax

```
<image>[image in Base64-encoded format]</image>
<image>[image file name]</image>
<image>[data source.]column</image>
```

Tag properties

Attribute	Values	Default	Description
resource	static, datasource, file		The source of the image: static (loaded from the report definition; the image is stored Base64-encoded in the XML), datasource (loaded from the data source, e.g. an SQL database), file (loaded from the specified file, given with a full path or relative to the program's directory).
format	b64, bin, svg		The encoding of a static image: b64 (Base64-encoded raster image), bin (binary), svg (SVG vector image). (The numeric codes 1 , 0 , and 2 are also accepted.)
scaling	true, false	true	Whether the image is scaled. (The current report engine ignores this attribute when reading and always scales the image.)
aspectRatio	ignore, keep, expand	keep	When scaling is on, this controls how the aspect ratio is handled: ignore (scaled freely, aspect ratio not preserved), keep (scaled as large as possible inside the rectangle while preserving the aspect ratio), expand (scaled as small as possible while covering the rectangle and preserving the aspect ratio).
transformMode	fast, smooth	fast	The scaling quality: fast (fast transformation) and smooth (smooth transformation).

(The common item properties also apply.)

39.12.9. Barcode

The Barcode option enables you to create barcodes. Barcodes are rendered through the Zint barcode library, so a wide range of symbologies is supported. Barcodes may be either static or dynamic items, similar to images: static barcodes read their value from the report definition, while dynamic barcodes are loaded from the specified data source.

XML syntax

```
<barcode>[code]</barcode>
<barcode>[data source.]column</barcode>
```

Tag properties

Attribute	Values	Default	Description
resource	static, datasource		The source of the barcode: static (the code is stored in the report definition) and datasource (the code is loaded from the data source).
barcodeType	integer (Zint symbol ID)	13	The barcode symbology, given as the numeric Zint symbol ID. (The legacy values EAN13 , EAN , and EANX are accepted for backward compatibility and map to the default symbology.)
code	text		The static barcode value.
showText	true, false	true	Whether the human-readable text is shown beneath the barcode.
foreColor	#RRGGBB color		The barcode foreground color in #RRGGBB format.
backColor	#RRGGBB color		The barcode background color in #RRGGBB format.
rotation	integer (degrees)		The barcode rotation in degrees.
scale	number		The rendering scale factor. This is useful when printing barcodes to a high-resolution device such as a printer.
whiteSpace	integer		The quiet-zone (white space) width around the barcode.
borderType	integer		The border type of the barcode.
borderWidth	integer		The border width of the barcode.
fontSize	number (pt)		The font size, in points, of the barcode's human-readable text.

(The common item properties also apply. Additional advanced Zint options — such as `mode`, `secure`, `cols`, `vers`, `primary`, `opt1`, `opt2`, and `opt3` — are also stored when set, for symbologies that need them.)

39.12.10. Cross tab

The Cross tab item renders a cross-tabulation (pivot) table from a data source: it summarizes data across row and column dimensions with aggregated values in the cells.

XML syntax

```
<crosstab></crosstab>
```

Tag properties

Attribute	Values	Default	Description
datasource	data source ID		The name/ID of the data source the cross tab is built from.
isPivot	true, false	false	If true , the item works in pivot-table mode.
ptRow	column name		The data column used as the pivot row dimension.
ptColumn	column name		The data column used as the pivot column dimension.
ptValue	column name		The data column used as the pivot value.
ptAgg	integer	1	The aggregation function applied to the pivot value, given as a numeric code.
ptSort	true, false	false	The sort order applied to the pivot: if true , the pivot is sorted automatically.
showColHeader	true, false	true	Whether the column header is shown.
showRowHeader	true, false	true	Whether the row header is shown.
showBottomSummary	true, false	true	Whether the bottom summary row is shown.
showSideSummary	true, false	true	Whether the side summary column is shown.
cellPadding	number (mm)	0	The cell padding.
cellSpacing	number (mm)	0	The spacing between cells.
leftMargin	number (mm)	0	The left margin of the table.
rightMargin	number (mm)	0	The right margin of the table.
topMargin	number (mm)	0	The top margin of the table.
bottomMargin	number (mm)	0	The bottom margin of the table.
autoWidth	true, false		If true , the column widths are calculated automatically. (Currently not read by the report engine.)
autoHeight	true, false		If true , the row heights are calculated automatically. (Currently not read by the report engine.)
breakTable	true, false	true	If true , the table may break across pages.

(The common item properties also apply.)

39.12.11. Table view

The Table item renders the contents of a Qt item-view / item-model as a table. It is useful for embedding a `QAbstractItemModel`-based model or a `QTableView` directly in the report.

XML syntax

```
<table></table>
```

Tag properties

Attribute	Values	Default	Description
<code>datasource</code>	data source ID		The name/ID of the data source the table is built from.
<code>model</code>	model ID		The ID of the item model to render.
<code>tableView</code>	view ID		The ID of the table view to render.
<code>showHeaderH</code>	<code>true</code> , <code>false</code>	<code>true</code>	Whether the horizontal (column) header is shown.
<code>showHeaderV</code>	<code>true</code> , <code>false</code>	<code>true</code>	Whether the vertical (row) header is shown.
<code>showBorder</code>	<code>true</code> , <code>false</code>	<code>true</code>	Whether the table border is shown.
<code>alternateRowBack</code>	<code>true</code> , <code>false</code>	<code>false</code>	If true , alternating row background colors are used.
<code>autoRowHeight</code>	<code>true</code> , <code>false</code>	<code>false</code>	If true , row heights are calculated automatically.
<code>cellPadding</code>	number (mm)	0	The cell padding.
<code>cellSpacing</code>	number (mm)	0	The spacing between cells.
<code>leftMargin</code>	number (mm)	0	The left margin of the table.
<code>rightMargin</code>	number (mm)	0	The right margin of the table.
<code>topMargin</code>	number (mm)	0	The top margin of the table.
<code>bottomMargin</code>	number (mm)	0	The bottom margin of the table.
<code>hiddenCols</code>	comma-separated integers		A comma-separated list of hidden column indexes.
<code>hiddenRows</code>	comma-separated integers		A comma-separated list of hidden row indexes.

(The common item properties also apply.)

39.12.12. Graph or custom item

The Graph/Custom item is a special member of the NCReport items. This option enables you to render special, custom-defined content in reports. The typical use case is using this feature to render graphs

or similar content. To use this feature, you need to do the following:

- Add a Graph (Custom) item to your report in the designer and specify the size and location of this object.
- Set the class ID of the specified item.
- If needed, add a static item definition for the object. If you set its resource to the data source and fill in the data source column, this information will come from the specified data source column.
- Derive the `NCReportAbstractItemRendering` class, implementing its `paintItem` method. You may 'attach' this class to your graph or any kind of rendering class by multiple inheritance. The `paintItem` method receives the following parameters:

QPainter* painter

The painter pointer.

NCReportOutput* output

The output object pointer.

const QRectF& rect

The rectangle of the object in the specified output. The geometry of the rectangle depends on the output's resolution.

const QString& itemdata

Item definition information that comes from the data source or report definition for custom purposes.

- Set the string ID of your class for identification, using the `setID(...)` method.
- Create your custom rendering object (it must be derived from the `NCReportAbstractItemRendering` class) and add it to the NCReport object using the `addItemRenderingClass(...)` method.

XML syntax

```
<graph></graph>
```

Tag properties

Attribute	Values	Default	Description
<code>classID</code>	text		Class ID text used to identify the custom item class.
<code>resource</code>	<code>static</code> , <code>datasource</code>		The source of the graph definition: static (the definition text is stored in the report definition) and datasource (the definition text is loaded from the data source).

(The common item properties also apply.)

39.12.13. Graphics view item

The Graphics view item renders a custom Qt Graphics View / Graphics Scene into the report. Like the graph item, it is identified by a class ID and backed by a custom rendering class supplied from code.

XML syntax

```
<graphicsview></graphicsview>
```

Tag properties

Attribute	Values	Default	Description
classID	text		Class ID text used to identify the custom graphics-view class.

(The common item properties also apply.)

39.12.14. Chart

The Chart item renders a chart in the report. It is available only in builds compiled with the Highcharts integration enabled; when that integration is not present, the element is ignored.

XML syntax

```
<chart></chart>
```

Appendix A: Glossary

This glossary explains the most important terms used throughout this book.

Band

A horizontal area of the page that holds report items. In NCReport, every report section is a band. See also *Section* and *Zone*.

Batch mode

A running mode in which several report definitions are rendered one after another into a single output document.

Cross-tab table

A table item (also called a pivot table) that presents data source records as columns and data source columns as rows, often with horizontal and vertical totals.

Data source

An object that provides the data of a report, row by row. NCReport supports SQL queries, text (CSV) data, XML data, Qt item models, string lists, and custom data source classes.

Data source relation

A parent–child link between two data sources. The child data source is filtered or re-run for each record of the parent, which makes master–detail (sub-query) reports possible.

Designer

NCReport Designer, the standalone GUI application used to create and modify report definitions.

Detail section

The section that is printed once for each record of its assigned data source. A report can contain several detail sections.

Double pass mode

A running mode in which the report is processed twice. The first (test) pass calculates values that are only known at the end, such as the total number of pages; the second pass produces the real output.

Expression

A formula that is evaluated at run time. Expressions can reference data source columns (`$D{}`), parameters (`$P{}`), variables (`$V{}`), and other values, and can be evaluated by the script engine.

Field

A report item that displays a dynamic value: data source column, parameter, variable, system variable, or the result of an expression.

Group

A set of consecutive detail records that share the same value of the group expression. Each group can have its own header and footer section and can reset variables.

Item

An object placed in a section, such as a label, field, line, rectangle, ellipse, image, barcode, rich text, table, chart, or custom item.

Label

A report item that displays static text which does not change while the report is rendered.

Output

The target of the rendered report, for example the print preview, a printer, a PDF, SVG, or image file, HTML, or plain text.

Page footer

A section printed at the bottom of every page.

Page header

A section printed at the top of every page.

Parameter

A named value passed to the report by the host application, the Designer, or the command line tool. Parameters are referenced as `$P{name}`.

Print when expression

A logical expression attached to an item or section. The item or section is printed only when the expression evaluates to true.

Report definition

The XML template file (or database record) that describes the layout, data sources, and logic of a report. Also called a report template.

Report director

The part of the report engine that controls the rendering process: it iterates the data sources and manages the printing of sections and items.

Report engine

The NCReport library (and the command line tool built on it) that reads a report definition, processes the data, and renders the result to an output.

Report footer

A section printed once, at the end of the report.

Report header

A section printed once, at the beginning of the report.

Report source

The origin of the report definition: a file, a string, or an SQL database table.

Section

A band of the report, such as a page header, report header, detail, group header, group footer, report footer, or page footer.

Sub-report

A report part that is iterated for each record of a master data source, using its own data source and detail sections.

Sub-section

An additional detail section linked to a parent section and always rendered directly below it.

System variable

A built-in variable provided by the report engine, such as the current page number, the total page count, or the current date and time.

Template mode

A field or text mode in which references such as `$D{}`, `$P{}`, and `$V{}` are substituted inside a fixed text, for example `Page $V{pagenum} of $V{pagecount}`.

Variable

A named value calculated during report processing, typically used for counts, sums, averages, and other totals. Variables are referenced as `$V{name}`.

Zone

A virtual band inside a section, formed by items that share the same zone ID. Empty zones are skipped, so the section shrinks accordingly.

Index

@

`$D{}`, [87](#)

`$P{}`, [33](#)

`$S{}`, [113](#)

`$V{}`, [95](#)

A

API, [207](#)

B

Barcode item, [134](#)

Barcodes

 symbolologies, [135](#)

Base64 image, [129](#)

Batch mode, [192](#)

C

Cancelling a report, [221](#)

Chart item, [141](#)

Charts, [140](#)

Command line tool, [203](#)

Compiling NCReport, [9](#)

Conditional formatting, [180](#)

Cross-tab table, [174](#)

CSV data, [89](#)

Custom data source, [92](#)

Custom item, [143](#)

 C++, [219](#)

Customization hooks, [225](#)

`customizeReportDef()`, [225](#)

D

Data relations, [159](#)

Data source functions, [99](#)

Data source tree, [39](#)

Data sources, [84](#)

 adding in Designer, [53](#)

 column references, [87](#)

 custom, [92](#)

 item model, [91](#)

 location types, [86](#)

 opening roles, [86](#)

 SQL, [87](#)

 string list, [90](#)

 text, [89](#)

 XML, [89](#)

Database

 storing reports, [74](#)

Database connection

 Designer, [42](#)

Designer

 command line arguments, [82](#)

 data source manager, [39](#)

 editing tools, [70](#)

 geometry editor, [39](#)

 launching, [36](#)

 preferences, [77](#)

 translations, [81](#)

 user interface, [36](#)

Detail section, [45](#)

Double pass mode, [164](#)

Dynamic position, [148](#)

Dynamic style, [150](#)

E

Ellipse item, [31](#)

Excel output, [188](#)

Expression builder, [40](#)

Expressions, [95](#)

F

Field item, [31](#)

Formatting

 data, [115](#)

 dates, [117](#)

 numbers, [116](#)

G

Geometry editor, [39](#)

Graph item, [143](#)

Graphics view item, [145](#)

Grid, [73](#)

Group footer, [65](#)

Group header, [65](#)

Groups, [65](#)

 totals, [108](#)

H

Hide repeated values, [123](#)

Highcharts, [141](#)

I

Image item, [128](#)

Images

sources, [129](#)

Installation, [8](#)

Linux, [8](#)

source package, [9](#)

Windows, [9](#)

Internationalization, [165](#)

Item model data source, [91](#)

L

Label item, [30](#)

Label reports, [154](#)

Line item, [31](#)

Lock items, [70](#)

M

Margins, [47](#)

Markdown, [122](#)

Master-detail reports, [159](#)

Multi-language reports, [165](#)

N

NCReport

architecture, [6](#)

features, [5](#)

NCReport class, [207](#)

NCReportUserFunction, [198](#)

O

Odd and even pages, [196](#)

Outputs, [182](#)

Excel, [188](#)

HTML, [187](#)

image, [186](#)

PDF, [185](#)

preview, [184](#)

printer, [185](#)

SVG, [186](#)

text, [190](#)

P

Page breaks, [151](#)

Page footer, [45](#)

Page header, [45](#)

Page settings, [47](#)

Page size, [47](#)

Page totals, [107](#)

pagecount, [111](#)

pageno, [111](#)

Parameters, [33](#)

PDF output, [185](#)

Pivot table, [174](#)

Preferences, [77](#)

Print preview, [184](#)

Print-when expression, [97](#)

Progress signals, [221](#)

Project file (.pro), [26](#)

Q

QAbstractItemModel, [91](#)

QGraphicsView, [145](#)

QR code, [135](#)

Qt application

integrating NCReport, [26](#)

QTableView, [169](#)

R

Rectangle item, [31](#)

References, [95](#)

Repeated detail, [197](#)

Report definition, [7](#)

Report director, [6](#)

Report engine, [6](#)

Report footer, [45](#)

Report header, [45](#)

Report items, [30](#)

barcode, [134](#)

ellipse, [31](#)

field, [31](#)

image, [128](#)

label, [30](#)

line, [31](#)

rectangle, [31](#)

text, [30](#)

Report options, [47](#)

Report source, [209](#)

database, [74](#)
Report types, [154](#)
Requirements, [8](#)
Rich text, [30](#)
 page breaking, [124](#)
Right-to-left layout, [76](#)
Rotation, [125](#)
Running a report
 from Designer, [68](#)
Running balance, [107](#)

S

Script editor, [113](#)
Script expressions, [95](#)
Sections, [45](#)
Snap to grid, [73](#)
SQL connection, [42](#)
SQL data source, [87](#)
String list data source, [90](#)
Sub-reports, [167](#)
Sub-sections, [195](#)
System variables, [111](#)

T

Table view item, [169](#)
Template mode, [98](#)
Test data editor, [80](#)
Text data source, [89](#)
Text document mode, [153](#)
Text item, [30](#)
Text output, [190](#)
Totals, [103](#)
 group, [108](#)
 page, [107](#)
Translations
 reports, [165](#)
Tutorial
 Designer, [42](#)
 first report, [11](#)

U

Undo and redo, [70](#)
User functions, [198](#)

V

Variable modifier, [109](#)
Variables, [103](#)

X

XML data source, [89](#)
XML template, [7](#)

Z

Zint, [134](#)
Zones, [146](#)
Zoom, [74](#)