The KDevelop Programming Handbook

The User Guide to C++ Application Design for the K Desktop Environment (KDE) with the KDevelop IDE, Version 1.0

Ralf Nolden <*Ralf.Nolden@post.rwth-aachen.de*>

The KDevelop Team

Version 2.1 , July 7, 1999

Contents

Chapter 1

Introduction

As Unix Systems are becoming more and more popular to even beginners working with computer machines due to its advantages in regards of stability and functionality, most are somehow disappointed, because those applications don't have a consistent look and each one behaves different from another. With KDE, developers have an almost perfect way to create first-class applications for Unix desktop systems to get a wider user community by the mere quality their applications have to offer. Therefore, KDE becomes more and more popular as a base for programming design, and developers want to take advantage of the possibilities that the system has to offer.

1.1 What you should know already

For making the best use of this programming handbook, we assume that you already know about the C++ programming language; if not, you should make yourself familiar with that first. Information about C++ is available through various sources either in printed form at your local bookstore or by tutorials found on the Internet. Knowledge about the design of Graphical User Interfaces is not required, as this handbook tries to cover the application design for KDE programs, which also includes an introduction into the Qt toolkit as well as the KDE libraries and the design of User Interfaces. Also, you should have made yourself comfortable with KDevelop by reading *The User Manual to KDevelop*, which contains a descriptive review of the functionality provided by the IDE.

1.2 About this Handbook

This handbook has been written to give developers an introduction into KDE application development by using the KDevelop Integrated Development Environment.

The following chapters therefore give an introduction on how to create projects, explains the source-codes already generated and shows how to extend the given sources on various topics such as toolbars, menu bars and view areas.

Then the dialogeditor is discussed in detail, explaining how widgets are created and covers widget properties settings in detail for all provided widgets.

Finally, you will learn about several topics that will complete your knowledge in regards of project design and helps you work out additional issues besides coding such as adding API documentation and extending online-manuals.

In the next chapter

we'll take a look at the Qt and KDE libraries, showing basic concepts and why things are the way they are. Also, we will discuss how to create the tutorial applications provided with the Qt toolkit by using KDevelop, so beginners can already see first results with a few steps, and thereby will learn how to make use of some of KDevelop's best features.

In the following chapters you will learn:

- how to create an application with the KAppWizard,
- what the project skeleton already provides,
- what the code already created means,
- how to create your own views,
- how to extend your application's functionality by dialog, menu bars and toolbars
- how to make your application user friendly by providing help functions and
- how to write SGML online documentation.

1.3 Additional Information

Additional information about Qt/KDE programming is available by various sources:

- *Programming with Qt* by Matthias Kalle Dalheimer, published by O'Reilly (see `<http://www.oreilly.com>`, covering almost all aspects of the Qt GUI toolkit and contains examples as well.
- *The User Manual to KDevelop*, provided with the KDevelop IDE,
- *Online-Reference* to the Qt-library, provided with your copy of the Qt toolkit in HTML and available as Postscript on `<http://www.troll.no>`
- On the Internet, see
  - the Troll Tech web site at `<http://www.troll.no>`,
  - the KDE web site at `<http://www.kde.org>`,
  - the KDE developer web site at `<http://developer.kde.org>`
  - the KDevelop home page at `<http://www.kdevelop.org>`

Additionally, you should look for help by subscribing to the various mailing lists, whose addresses are available on the mentioned web sites, and on the Usenet newsgroups dedicated to users of KDE and Unix Systems as well as about the C and C++ programming language.

For obtaining help about the KDevelop IDE, you should send requests to our mailinglist at *kdevelop@fara3.cs.uni-potsdam.de*. Mind that the KDevelop team is dedicated to provide the means to enable you to program applications and therefore is not intended as a technical support team in cases where the applications you're developing don't work due to implementation errors or misconfigurations of your operating system. By this, we ask all users to take advantage of the mailinglist in any case you're running into problems with the use of the IDE itself, as well as for bug reports and suggestions for improving the functionality of the development environment.

Chapter 2

The KDE and Qt Libraries

The Norwegian company Troll Tech (<http://www.troll.no>) provides a so-called GUI toolkit, named Qt. Thereby, GUI means "**G**raphical **U**ser **I**nterface", and therefore, Qt-based applications represent themselves with buttons, windows etc, allowing user input by visualizing the functions an application provides. Such a toolkit is needed for developing graphical applications that run on the X-Window interface on Unix Systems, because X does not contain a pre-defined user interface itself. Although other toolkits are also available to create User Interfaces, Qt offers some technical advantages that make application design very easy. Additionally, the Qt toolkit is also available for Microsoft Windows systems, which allows developers to provide their applications for both platforms.

The KDE Team (<http://www.kde.org>) joined together with the goal to make using Unix Systems more friendly, and decided to use the Qt toolkit for the development of a window manager on X-Window, plus a variety of tools included with the KDE packages. The K Desktop Environment therefore contains the window manager *kwm*, the file manager *kfm* and the launch panel *kpanel* as the main components plus a variety of first-class utilities and applications. After KDE was out, a lot of developers turned their eyes towards the new environment and what it has to offer them. The KDE libraries are providing essential methods and classes that make all applications designed with them look similar and consistent, so the user has the great advantage that he only has to get accustomed with an application's specific usage, not with handling dialogs or buttons. Also, KDE programs integrate themselves into the desktop and are able to interact with the file manager via drag'n drop, offer session management and many more, if all features offered by the KDE libraries are used.

Both, the Qt toolkit and the KDE libraries, are implemented in the C++ programming language; therefore applications that make use of these libraries are also mostly written in C++. In the following chapter, we'll make a short trip through the libraries to see what already is provided and how Qt and KDE applications are created in general.

2.1 The Qt GUI Toolkit

As said, the Qt library is a toolkit that offers graphical elements that are used for creating GUI applications and are needed for X-Window programming. Additionally, the toolkit offers:

- A complete set of classes and methods ready to use even for non-graphical programming issues,
- A good solution towards user interaction by virtual methods and the signal/slot mechanism,

- A set of predefined GUI-elements, called "widgets", that can be used easily for creating the visible elements
- Additional completely pre-defined dialogs that are often used in applications such as progress and file dialogs.

Therefore knowing the Qt classes is very essential, even if you only want to program KDE-applications. To have an impression on the basic concept how GUI-applications are constructed and compiled, we'll first have a look at a sample Qt-only program; then we'll extend it to a KDE program.

2.1.1 The first Qt Application

As usual, programs in C++ have to contain a `main()` function, which is the starting point for application execution. As we want them to be graphically visible in windows and offering user interaction, we first have to know, how they can show themselves to the user. For an example, we'll have a look at the first tutorial included with the Qt Online Reference Documentation and explain the basic execution steps; also why and how the application window appears:

```
#include <qapplication.h>
#include <qpushbutton.h>

int main( int argc, char **argv )
{
QApplication a( argc, argv );

QPushButton hello( "Hello world!" );
hello.resize( 100, 30 );

a.setMainWidget( &hello );
hello.show();
return a.exec();
}
```

This application merely paints a window containing a button with "Hello world" as its text. As for all Qt-based applications, you first have to create an instance of the class `QApplication`, represented by `a`.

Next, the program creates an instance of the class `QPushButton` called `hello`, this will be the button. The constructor of `hello` gets a string as a parameter, which is the contents of the widget visible as the buttons text.

Then the `resize()` method is called on the `hello` button. This changes the default size a widget (which is in this case the QPushButton) has when created to the length of 100 pixels and the height of 30 pixels. Finally, the `setMainWidget()` method is called for `a` and the `show()` method for `hello`. The `QApplication` is finally executed by `a.exec()`, enters the main event loop and waits until it has to return an integer value to the overlaying Operating System signaling that the application is exited.

2.1.2 The Reference Documentation for Qt

Now, let's have a quick look at the reference documentation of the Qt library. To do this, start KDevelop and select "Qt-library" from the "Help"-menu in the menubar. The documentation browser

opens and shows you the start page of the Qt reference. This will be your first place to get information about Qt, it's classes and the available functions they provide. Also, the above program is the first that is included in the tutorials section. To get to the classes we want to have a look at, `QApplication` and `QPushButton`, select "Alphabetical Class List" and search for the according names. Follow either of them to have a look at the class documentation.

For `QApplication`, you will see the constructor and all other methods that this class provides. If you follow a link, you will get more information about the usage and meaning of the methods, which is very useful when you sometimes can't detect the correct use or want to have an example. This also counts for the KDE library documentation, which uses a similar documentation type; therefore this is almost all you have to know about using the class-references with the documentation browser.

2.1.3 Interpretation of the Sample

Starting with `QApplication`, you will find all the methods used in our first example:

- the constructor `QApplication()`,
- the `setMainWidget()` method and
- the `exec()` method.

The interpretation why we use these methods is very simple:

1. first create an instance of the class `QApplication` with the constructor, so we can make use of the GUI elements provided by Qt,
2. create a widget which will be the contents of our program window,
3. set the widget as the main widget for `a`,
4. execute the `a` instance of `QApplication`.

The second object of our program is the pushbutton, an instance of the class `QPushButton`. From the two constructors given to create an instance, we used the second: this accepts a text, which is the label contents of the button; here, it is the string "Hello world!". Then we called the `resize()` method to change the size of the button according to it's contents- the button has to be larger to make the string completely visible.

But what about the `show()` method ? Now, you see that like most other widgets, `QPushButton` is based on a single-inheritance- here, the documentation says, *Inherits* `QButton`. Follow the link to the `QButton` class. This shows you a lot of other methodss that are inherited by QPushButton, which we'll use later to explain the signal/slot mechanism. Anyway, the `show()` method is not listed, therefore, it must be a method that is provided by inheritance as well. The class that `QButton` inherits, is `QWidget`. Just follow the link again, and you will see a whole bunch of methods that the `QWidget` class provides; including the `show()` method. Now we understand what was done in the sample with the button:

1. create an instance of `QPushButton`, use the second constructor to set the buttons text,
2. resize the widget to it's contents,
3. set the widget as the main widget of the `QApplication` instance `a`,

4. tell the widget to display itself on the screen by calling `show()`, an inherited method from `QWidget`.

After calling the `exec()` method, the application is visible to the user, showing a window with the button showing "Hello world!". Now, GUI programs behave somewhat differently than procedural applications. The main thing here is that the application enters a so-called "main event loop". This means that the program has to wait for user actions and then react to it, also that for a Qt application, the program has to be in the main event loop to start the event handling. The next section tells you in short what this means to the programmer and what Qt offers to process user events.

(For already advanced users: The button has no parent declared in the constructor, therefore it is a top-level widget alone and runs in a local event loop which doesn't need to wait for the main event loop, see the `QWidget` class documentation and *The KDE Library Reference Guide*)

Summary:

A Qt application always has to have one instance of the class `QApplication`. This provides that we can create windows that are the graphical representation of programs to the user and allow interaction. The window contents itself is called a "Main Widget", meaning that all graphical elements are based on the class `QWidget` and can be any type of widget that fits the needs of the application to communicate with the user. Therefore, all user elements somehow have to inherit `QWidget` to be visible.

2.1.4 User Interaction

After reading the last sections, you should already know:

- What the Qt-library provides in terms of GUI applications,
- how a program using Qt is created and
- where and how to find information about classes that you want to use with the documentation browser

Now we'll turn to give the application "life" by processing user events. Generally, the user has two ways to interact with a program: the mouse and the keyboard. For both ways, a graphical user interface has to provide methods that detect actions and methods that do something as a reaction to these actions.

The Window system therefore sends all interaction events to the according application. The QApplication then sends them to the active window as a `QEvent` and the widgets themselves have to decide what to do with them. A widget receives the event and processes `QWidget`::event(QEvent*)/, which then decides which event has been executed and how to react; event() is therefore the main event handler. Then, the `event()` function passes the event to so-called event filters, that determine what happened and what to do with the event. If no filter signs responsible for the event, the specialized event handlers are called. Thereby we can decide between:

a) Keyboard events –TAB and Shift-TAB keys:

changes the keyboard input focus from the current widget to the next widget in the focus order. The focus can be set to widgets by calling `setFocusPolicy()` and process the following event handlers:

- `virtual void focusInEvent ( QFocusEvent * )`

- `virtual void focusOutEvent ( QFocusEvent * )`

b) all other keyboard input:

- `virtual void keyPressEvent ( QKeyEvent * )`
- `virtual void keyReleaseEvent ( QKeyEvent * )`

c) mouse movements:

- `virtual void mouseMoveEvent ( QMouseEvent * )`
- `virtual void enterEvent ( QEvent * )`
- `virtual void leaveEvent ( QEvent * )`

d) mouse button actions:

- `virtual void mousePressEvent ( QMouseEvent * )`
- `virtual void mouseReleaseEvent ( QMouseEvent * )`
- `virtual void mouseDoubleClickEvent ( QMouseEvent * )`

e) window events containing the widget:

- `virtual void moveEvent ( QMoveEvent * )`
- `virtual void resizeEvent ( QResizeEvent * )`
- `virtual void closeEvent ( QCloseEvent * )`

Note that all event functions are virtual and protected; therefore you can re-implement the events that you need in your own widgets and specify how your widget has to react. `QWidget` also contains some other virtual methods that can be useful in your programs; anyway, it is sufficient to know about `QWidget` very well generally.

2.1.5 Object Interaction by Signals and Slots

Now we're coming to the most obvious advantages of the Qt toolkit: the signal/slot mechanism. This offers a very handy and useful solution to object interaction, which usually is solved by `callback` functions for X-Window toolkits. As this communication requires a strict programming and sometimes makes user interface creation very difficult (as referred by the Qt documentation and explained in *Programming with Qt* by K.Dalheimer), Troll Tech invented a new system where objects can emit signals that can be connected to methods declared as slots. For the C++ part of the programmer, he only has to know some things about this mechanism:

1. the class declaration of a class using signals/slots has to contain the `Q_OBJECT` macro at the beginning (without the semicolon); and have to be derived from the `QObject` class,
2. a signal can be emitted by the keyword `emit`, e.g. `emit signal(parameters);` from within any member function of a class that allows signals/slots,
3. all signals used by the classes that are not inherited have to be added to the class declaration by a `signals:` section,

4. all methods that can be connected with a signal are declared in sections with the additional keyword `slot`, e.g. `public slots:` within the class declaration,

5. the meta-object compiler `moc` has to run over the header file to expand the macros and to produce the implementation (which is not needed to know.). The output files of `moc` are compiled as well by the C++ compiler.

Another way to use signals without deriving from `QObject` is to use the `QSignal` class- see the reference documentation for more information and example usage. In the following, we assume you're deriving from `QObject`.

This way, your class is able to send signals anywhere and to provide slots that signals can connect to. By using the signals, you don't have to care about who's receiving it- you just have to emit the signal and whatever slot you want to connect to it can react to the emission. Also the slots can be used as normal methods during implementation.

Now, to connect a signal to a slot, you have to use the `connect()` methods that are provided by `QObject` or, where available, special methods that objects provide to set the connection for a certain signal.

Sample Usage

To explain the way how to set up object-interaction, we'll take our first example again and extend it by a simple connection:

```
#include <qapplication.h>
#include <qpushbutton.h>

int main( int argc, char **argv )
{
QApplication a( argc, argv );

QPushButton hello( "Hello world!" );
hello.resize( 100, 30 );

a.setMainWidget( &hello );

connect(&hello, SIGNAL( clicked() ), &a, SLOT( quit() );

hello.show();
return a.exec();
}
```

You see, the only addition to give the button more interaction is to use a `connect()` method: `connect(&hello, SIGNAL( clicked() ), &a, SLOT( quit() );` is all you have to add. What is the meaning now ? The class declaration of `QObject` says about the `connect()` method:

```
bool connect ( const QObject * sender, const char * signal, const QObject *
receiver, const char * member )
```

This means, you have to specify a QObject instance pointer that is the sender of the signal, meaning that it can emit this signal as first parameter; then you have to specify the signal that you want to connect to. The last two parameters are the receiver object that provides a slot, followed by the member function which actually *is* the slot that will be executed on signal emission.

By using signals and slots, your program's objects can interact with each other easily without explicitely depending on the type of the receiver object. You will learn more about using this mechanism for productive usage later in this handbook. More information about the Signals/Slot mechanism can also be found in *The KDE Library Reference Guide* and the Qt online reference.

2.2 What KDE provides

2.2.1 The KDE 1.1.x libraries

For the time of this writing and due to the fact that KDevelop uses KDE 1.1, I'm referring to the state of the KDE libraries at that release. The main KDE libraries you'll be using for creating your own KDE applications are:

- the KDE-Core library, containing all classes that are non-visible elements and provide functionality your application may use.
- the KDE-UI library, containing user interface elements like menu bars, toolbars and the like,
- the KFile library, containing the file selection dialogs,

Additionally, for specific solutions KDE offers the following libraries:

- the KHTMLW library, offering a complete HTML-interpreting widget that is used by various programs like KDEHelp, KFM, KDevelop,
- the KFM library, allowing to use the KDE file manager from within your application.
- the KAb library, the KAddressBook. Provides address-book access for e.g. email applications
- the KSpell library, offering widgets and functionality to integrate the use of Ispell, the common spell-checker, in applications like editors; used for the KEdit application.

Next, we'll have a look at what is needed to turn our first Qt application into a KDE one.

2.2.2 Example KDE Application

In the following, you will see that writing a KDE application is not much more difficult than a Qt application. For the use of KDE's features, you just have to use some other classes, and you're almost done. As an example, we'll discuss the changed version of the Qt example from above:

```
#include <kapp.h>
#include <qpushbutton.h>

int main( int argc, char **argv )
{
KApplication a( argc, argv );

QPushButton hello( "Hello world!" );
hello.resize( 100, 30 );

a.setTopWidget( &hello );
```

```
    connect(&hello, SIGNAL( clicked() ), &a, SLOT( quit() );

    hello.show();
    return a.exec();
    }
```

You see that first we have changed from `QApplication` to `KApplication`. Further, we had to change the previously used `setMainWidget()` method to `setTopWidget`, which `KApplication` uses to set the main widget. That's it ! Your first KDE application is ready- you only have to tell the compiler the KDE include path and the linker to link in the KDE-Core library with -lkdecore.

As you now know what at least the main() function provides generally and how an application gets visible and allows user and object interaction, we'll go on with the next chapter, where our first application is made with KDevelop- there you can also test everything which was mentioned before and see the effects.

What you should have looked into additionally until now is the reference documentation for Qt, especially the `QApplication`, `QWidget` and `QObject` class and the KDE-Core library documentation for the `KApplication` class. The *KDE Library Reference* handbook also covers a complete description about the invocation of the `QApplication` and `KApplication` constructors including command-line argument processing.

Chapter 3

Creating new Applications

The KAppWizard, or also called the KDE Application Wizard, is intended to let you start working on new projects with KDevelop. Therefore, all your projects are first created by the wizard; then you can start building them and extend the already provided source skeleton. KAppWizard also allows to choose between several project types according to your project's goals:

- Normal KDE Application: includes source code for a complete frame structure of a standard KDE application with support for the Document-View-Controller model, a menubar, toolbar and statusbar as well as a set of standard documentation based on SGML, KDE-links and application icons. This is the application type usually needed for a new KDE project.
- Mini KDE Application: contains the same general structure as the Normal KDE Application type but with the difference that the application's code only provides a widget as a window.
- Normal Qt Application: works like the normal KDE application by it's code with the difference that the project is based on the Qt library only and doesn't contain KDE support. This is intended for projects that have to be portable to Windows platforms or that don't want to require KDE libraries by the end-user.
- C Application: Is based on the C compiler only and runs in a console.
- C++ Application: Is based on the C++ compiler only and runs in a console like the C application, therefore doesn't require X-Window.
- Custom project: creates an empty project without any sourcecode. This is intended for already existing projects to port to KDevelop or for projects where you want to start from scratch. Mind that you have to take care for makefiles and configure scripts all by yourself.

In this chapter we'll see how the KAppWizard can be invoked and what has to be done to generate a KDE application project. This will also be the initial step of our coverage, where we will create the initial version of a sample project. For all other project types the steps are usually the same, just you may not have certain options available.

3.1 Invoking KAppWizard and Project Generation

3.1.1 Starting KAppWizard and the First Page

To start with your first KDE application, open KDevelop. Then select "New..." from the "Project"-menu. The KAppWizard starts, and you see a tree on the first page, containing the project types.

When a type is selected, you see a preview how it will look like after the initial build process. Choose the KDE subtree, Normal type. Then press the "Next" button on the bottom of the first wizard page. This will switch to the next page, where you have to set the general project options.

3.1.2 The Generate Settings Page

For our sample application, we choose the project name `KScribble`; therefore insert this in the field "Projectname". Then select the directory you want to have your project build in; the default is your home directory. You can enter the path manually or you can as well press the button on the right to select the directory by a dialog.

Next, you have to enter the Version number. For the first version, set this to `0.1`. It is usual to number new applications that are in development for the first release lower than 1, and as the initial version will only contain the standard framework, we'll name this the 0.1 version.

Finally, add your name to the "Author" field and your email address. You can leave all other options to their default settings.

To give you some information about all other options, you can press the right mouse button over the options, and you will get a quick-help window that describes the option's purpose.

These are:

- **generate sources and headers:** generates the application source code
- **GNU-Standard-Files:** adds a copy of the GNU-General Public License to the project as well as some standard files for user information when distributing the package.
- **User-Documentation:** the user handbook in SGML, already prepared for your project.
- **API-Documentation:** creates an initial HTML documentation set for the **A**pplication **P**rogramming **I**nterface.
- **lsm-File:** the Linux Software Map, used by distribution sites and contains short information about the project's purpose and requirements.
- **.kdelnk-File:** a KDE link that will install your application in the "Applications" tree of the KDE-Panel.
- **Program-Icon:** an Icon that represents your project and can be used to create a link on the desktop.
- **Mini-Icon:** a Mini-version of the program icon that represents your application besides its name in the KDE-Panel and is shown in your application's dialogs and main-window in the left upper corner.

Now we'll switch to the next page by pressing the "Next" button again to set the template for the header files of your project.

3.1.3 The Header and Source Templates

The header template page allows you to automatically include a preface for your header files, containing the filename, the construction date, the year of the copyright, also your name and your email address. You don't have to change those uppercase parts yourself, as KAppWizard does this automatically and stores the template for this project, so it can be used later again for creating new files.

The second part of the default header template contains a license information. By default, your project is set under the GNU General Public License, which is also included in the package. This license is used to protect your source code against any person that just copies your sources for his own purpose. The General Public License offers you this license for free and thereby protects your rights as the author, and is common for distributing free software. To get more information about the license, you should read the COPYING file in the base directory of your new project later which is a copy of the GPL and ships with your application already.

Anyway, you may want to choose another license or another header template you're already using for your projects. Therefore you can either edit the given default template directly. To do this, you're given the template in an editing window. To clear the default page, select "New", to use another template, select "Load...", which lets you choose the template file.

When you're done, go to the next page by entering "Next". This is the template page for your source files and is generally the same as the header template page. The only difference is that this template is used for your implementation files.

3.1.4 Creating the Project

Now that you've set all options for *KScribble*, select "Next" and press the "Generate" button on the bottom of the wizard window. If the button is not available, you haven't set all options correctly. To correct any errors, step back in the Wizard with "Back".

Then you'll see what KAppWizard does- he copies all templates to your project directory and creates the new project. After KAppWizard is finished, the "Cancel" button changes to an "Exit" button to leave the wizard.

After this last step, you're finished with creating a new project. KDevelop then loads it and the tree-views let you browse through the project's files and classes.

In the next section, we'll discuss how to build and run your first version of *KScribble* and how the source code is organized.

3.2 The First Build

After our project is generated, we'll first make a trip through the source code to get a general understanding how the application frame works. This won't only help to get started but we'll know where to change what in later steps.

When opening the LFV (Logical File Viewer) page on the tree-view, you see some folders that already sort the project files relevant to the developer. The first two folders are "Header" and "Sources". The Header-folder therefore logically contains all header files of the project, the Sources-folder all sourcecodes. All other folders are of no interest right now, so we'll turn back here later to see what they contain.

The two folders then contain the following files:

Headers:

- **kscribble.h :** contains the class declaration for the class **KScribbleApp**.
- **kscribbledoc.h :** contains the class declaration for the class **KScribbleDoc**.
- **kscribbleview.h :** contains the class declaration for the class **KScribbleView**.
- **resource.h :** contains a macro collection for the menu-ID's

Sources:

- **kscribble.cpp :** contains the implementation of the class **KScribbleApp**.
- **kscribbledoc.cpp :** contains the implementation of the class **KScribbleDoc**.
- **kscribbleview.cpp :** contains the implementation of the class **KScribbleView**.
- **main.cpp :** contains the **main()** function implementation.

Before diving into the sources, we'll let KDevelop build and run our new application. To do this, select "Make" from the "Build"-menu or hit the according button on the toolbar. The output window opens from the bottom of KDevelop and lets you see what `make` does by the messages it gives us:

```
1  Making all in docs
2  make[1]: Entering directory `/home/rnolden/Tutorial/kscribble1/kscribble/docs'
3  Making all in en
4  make[2]: Entering directory `/home/rnolden/Tutorial/kscribble1/kscribble/docs/en'
5  make[2]: Nothing to be done for `all'.
6  make[2]: Leaving directory `/home/rnolden/Tutorial/kscribble1/kscribble/docs/en'
7  make[2]: Entering directory `/home/rnolden/Tutorial/kscribble1/kscribble/docs'
8  make[2]: Nothing to be done for `all-am'.
9  make[2]: Leaving directory `/home/rnolden/Tutorial/kscribble1/kscribble/docs'
10 make[1]: Leaving directory `/home/rnolden/Tutorial/kscribble1/kscribble/docs'
11 make[1]: Entering directory `/home/rnolden/Tutorial/kscribble1/kscribble'
12 g++ -DHAVE_CONFIG_H -I. -I. -I.. -I/opt/kde/include -I/usr/lib/qt/include  -I/usr/X11R6/include    -OO
   kscribbleview.cpp
13 g++ -DHAVE_CONFIG_H -I. -I. -I.. -I/opt/kde/include -I/usr/lib/qt/include  -I/usr/X11R6/include    -OO
   kscribbledoc.cpp
14 g++ -DHAVE_CONFIG_H -I. -I. -I.. -I/opt/kde/include -I/usr/lib/qt/include  -I/usr/X11R6/include    -OO
   kscribble.cpp
15 g++ -DHAVE_CONFIG_H -I. -I. -I.. -I/opt/kde/include -I/usr/lib/qt/include  -I/usr/X11R6/include    -OO
   main.cpp
16 /usr/bin/moc ./kscribble.h -o kscribble.moc.cpp
17 g++ -DHAVE_CONFIG_H -I. -I. -I.. -I/opt/kde/include -I/usr/lib/qt/include  -I/usr/X11R6/include    -OO
   kscribble.moc.cpp
18 /usr/bin/moc ./kscribbledoc.h -o kscribbledoc.moc.cpp
19 g++ -DHAVE_CONFIG_H -I. -I. -I.. -I/opt/kde/include -I/usr/lib/qt/include  -I/usr/X11R6/include    -OO
   kscribbledoc.moc.cpp
20 /usr/bin/moc ./kscribbleview.h -o kscribbleview.moc.cpp
21 g++ -DHAVE_CONFIG_H -I. -I. -I.. -I/opt/kde/include -I/usr/lib/qt/include  -I/usr/X11R6/include    -OO
   kscribbleview.moc.cpp

22 /bin/sh ../libtool --silent --mode=link g++  -OO -g -Wall   -o kscribble -L/opt/kde/lib  -L/usr/X11R6/
   -rpath /usr/X11R6/lib kscribbleview.o kscribbledoc.o kscribble.o main.o kscribble.moc.o kscribbledoc.m
   -lkfile -lkfm -lkdeui -lkdecore -lqt -lXext -lX11

23 make[1]: Leaving directory `/home/rnolden/Tutorial/kscribble1/kscribble'
```

As you see, we've put line-numbers in front of each line, which won't appear in your output; it just makes it easier to describe what happened during the build now. First of all, `make` works recursively. That means, it starts from the directory it is invoked in and then goes into the subdirectories first, returns and processes the next directory. Finally, the directory it was started is processed and `make` finishes. Therefore, `make` started in the main project directory containing the sources first. In line

1 and 2, you see how the `make` process goes into the `docs` directory, then into the `en` subdirectory. As there isn't anything to do, it leaves these directories until it returns to the source-directory `kscribble` in line 11. Then, the real work starts: `make` invokes the compiler, here `g++` to compile the source-file `kscribbleview.cpp`. The macro `-DHAVE_CONFIG_H` says that the file `config.h` should be used. This is a file containing macros for the specific platform and application and is located in the main project directory. The following `-I` commands add the include path where `g++` can find the includes it needs. These are the current directory, the main project directory (by `-I..`) and the include path for the KDE, Qt and X11 library header files. The directories for these include files were determined by the `configure` script and set in the Makefiles, therefore, the compiler knows where these are located. Finally, `-O0` sets the optimization to zero (no optimization), `-g` enables debugging, `-Wall` sets the compiler warnings to `all` and `-c` tells the compiler to produce an object file, so only compile the file.

This is done for the other source-files of our project as well in lines 13-15. Obviously, our sources are compiled, but instead of linking the object files of the sources to the final binary, we see some other commands. In line 16, you see that the program "moc" is called to process the header-file `kscribble.h`, with it's output in `kscribble.moc.cpp`. Then, in line 17, this source file is compiled as well. The same happens with the other project header files until line 21. Now, as the Qt toolkit contains the signal/slot mechanism, but still stays a C++ implementation, you're using certain keywords that are not originally C++ language, such as the `signals:` and `slots:` declaration in your classes. This gives you the ability to easily allow object communication for all class objects that inherit the class `QObject`, so you can avoid the usual callback pointer functions. Therefore, the application needs the sources that implement this functionality, and that is why `moc` is called. `Moc` is the Meta Object Compiler of the Qt toolkit and builds the implementation for signals and slots mechanisms by parsing the header file and producing a source output that has to be compiled in the binary. As KDevelop projects use `automoc` to determine, which header file needs to be processed, you don't have to take care for any call on `moc` and the C++ compiler on the moc output files. Just remember the rules that make a class use the signals and slots- inheritance from `QObject` or any class that inherits `QObject` itself, inclusion of the `Q_OBJECT` macro (without semicolon !) at the beginning of the class declaration and the declarations for signals and slots.

Finally, your binary is built by the compiler. The output binary is called `kscribble`, the linker includes the path for the KDE and X11 libraries and links the sources against the libraries `kfile, kfm, kdeui, kdecore, qt, Xext` and `X11`. Then you're done and make exits.

3.3 The Source Skeleton

To gain a concept of how a KDE application works, we'll first have a very close look at the source skeleton already provided by the Application Wizard. As we already saw, we're having a set of source and header files that build the initial code for the application and make it ready-to-run. Therefore, the easiest way to explain the code is to follow the implementation line by line as it is processed during executing the program until it enters the main event loop and is ready to accept user input. Then, we'll have a look at the functionality that enables user interaction and how certain things work. This is probably the best way to explain the framework and, as it is similar to almost all KDE applications, will enable you to read source codes from other projects as well; additionally, you will know where to change what part of the code to make your applications behave the way they are designed for.

3.3.1 The `main()` Function

As the application begins its execution with entering the `main()` function, this will be the start for our code examination. The `main()` function of *KScribble* is implemented in the file `main.cpp` and can also be found using the Class Browser by selecting the "Globals" folder, sub-folder "Functions":

```
1  #include "kscribble.h"
2
3  int main(int argc, char* argv[]) {
4    KApplication app(argc,argv,"KScribble");
5
6    if (app.isRestored())
7    {
8       RESTORE(KScribbleApp);
9    }
10   else
11   {
12      KScribbleApp* kscribble = new KScribbleApp;
13      kscribble->show();
14      if(argc > 1){
15        kscribble->openFile(argv[1]);
16      }
17    }
18    return app.exec();
19 }
```

Now, what happens first is the usual creation of a `KApplication` object, which gets our application name *KScribble* as a third parameter. When creating a new `KApplication`, a new `KConfig` instance is created as well which is connected to a configuration file in $HOME/.kde/share/config/appname + rc which stores all information we want to use when starting application windows. The name we passed the constructor of `app` will be used as the window title later.

Despite of the example code for turning the first Qt application into a KDE one, the following code is somewhat different. After the `KApplication` object is present, we're testing if the application is started by the session management of `kwm` or manually by the user. This can be found out when calling `isRestored()` on the `app` object, which returns `true` for session management and `false` for a normal start.

As session management is a main feature of KDE applications and widely used by the framework but a lot more to explain, we'll follow the `else{}` section first; then we'll come back and explain the session functionality in a later step.

3.3.2 User Application Start

The `else{}` section now creates an instance of the class `KScribbleApp` in line 12. This object is called to show itself in line 13 as usual; line 14 determines if a command-line argument has been passed and, as this is usually the name of a file, calls the `kscribble` object to open it with `openFile()`.

Note that we didn't call the method `setTopWidget(kscribble)` for our application- this is already done by the class that `KScribbleApp` inherits. Now we'll have a look at our `KScribbleApp` object- what is it and what does it provide already ? The only thing we know until now is that it has to be a Widget to represent the user interface in the main window. Let's turn to the class implementation of `KScribbleApp`, which can be found in the file `kscribble.cpp` or by a click on the class icon in the Class Browser. As the instance is created by the constructor. First of all, we see that it inherits

the class `KTMainWindow`, which is a part of the `kdeui` library. This class itself inherits `QWidget`, so, as usual, we have a normal widget as the top-level window. `KTMainWindow` contains a lot of functionality that the class `KScribbleApp` makes use of. It provides a menubar, toolbar, statusbar and session management support. The only thing we have to do when sub-classing `KTMainWindow` is to create all the objects we need and create another widget that is managed by our `KTMainWindow` instance as the main view in the center of the window; usually this is the place where the user works like a text-editing view.

The Constructor

Let's have a look at the code for the constructor and see how the instance is created:

```
1   KScribbleApp::KScribbleApp()
2   {
3     config=kapp->getConfig();
4
5
6     ///////////////////////////////////////////////////////////////////
7     // call inits to invoke all other construction parts
8     initMenuBar();
9     initToolBar();
10    initStatusBar();
11    initKeyAccel();
12    initDocument();
13    initView();
14
15    readOptions();
16
17    ///////////////////////////////////////////////////////////////////
18    // disable menu and toolbar items at startup
19    disableCommand(ID_FILE_SAVE);
20    disableCommand(ID_FILE_SAVE_AS);
21    disableCommand(ID_FILE_PRINT);
22
23    disableCommand(ID_EDIT_CUT);
24    disableCommand(ID_EDIT_COPY);
25    disableCommand(ID_EDIT_PASTE);
26  }
```

We see that our config instance of `KConfig` now points to the applications configuration, so we can operate with the configuration file entries later.

Then, all parts of the application that are needed are created by their according member functions that are specific to our main window:

- **initMenuBar():** constructs the menubar,
- **initToolBar():** constructs the toolbar,
- **initStatusBar():** creates the statusbar,
- **initKeyAccel():** sets all keyboard accelerators for our application by the global and application specific keyboard configuration
- **initDocument():** creates the document object for the application window

- **initView():** creates the main widget for our view within the main window
- **readOptions():** reads all application specific settings from the configuration file and initializes the rest of the application such as the recent file list, the bar positions and the window size.

Finally, we disable some commands that the user can do, because they should not be available in the current application state. As we now have a general overview how the application window is created, we will look into the details of how the user elements are constructed by following the above methods.

The Menubar

As shown above, the menubar of *KScribble* is created by the method `initMenuBar()`. There, we create a set of `QPopupMenus` that pop up if the user selected a menuentry. Then, we insert them into the menubar and connect to the entries.

First, we create our `recent_file_menu`, which will contain the names of the last 5 opened files. We have to do this first, because this menu is inserted into the `file_menu`. Then we add the connection directly- we just retrieve the signal that is emitted by the menuentry with its entry number and call the `slotFileOpenRecent( int )`, which then calls the right file from the recent file list to be opened.

Then we create our "File"-menu. This will be the menu that will be visible in the menubar. The standard actions are then inserted into the popup-menu one by one- first the commands for creating a new file, open a file, close a file etc., finally "E&xit" to close the application. All menu entries have to be created in the order as they appear later, so we have to keep an eye on which we want to have at what place. As an example, we look at the following entries:

```
file_menu->insertItem(Icon("fileopen.xpm"), i18n("&Open..."), ID_FILE_OPEN );
file_menu->insertItem(i18n("Open &recent"), recent_files_menu, ID_FILE_OPEN_RECENT );
```

The first one inserts the "Open..." entry. As we want to have it with an icon, we use the `insertItem()` method with the icon's name. To understand the icon loading process, we need to know what or where `Icon()` is declared- in fact, it is a macro provided by the class `KApplication`:

```
#define Icon(x) kapp->getIconLoader()->loadIcon(x)
```

Additionally, it uses the following macro internally to get access to the application object:

```
#define kapp KApplication::getKApplication()
```

This means that the `KApplication` object already contains an Icon loader instance- we only have to get access to it; then it will load the according icon. As our icons are all from the KDE libraries, we don't have to take care for anything else- they are installed on the system automatically, therefore we also don't have to include them into our application package to use them.

After the icon parameter (which is optional), we insert the menuentry name by `i18n("&Open...")`. There, we have to watch two things: first, the entry is inserted with the `i18n()` method. Like the `Icon()` entry, it is a macro defined in `kapp.h` as well and calls the `KLocale` object of `KApplication` to translate the entry to the currently used language:

```
#define i18n(X) KApplication::getKApplication()->getLocale()->translate(X)
```

Hereby, it should be mentioned that one could think "I don't want to use macros"- you can do that in most cases. But here it is immanent to use `i18n()` because for internationalization the according language files have to be build. As this build process depends on the `i18n` string, you have to use the macro.

As you might have already guessed, the ampersand within menu entries is later interpreted as a line under the following letter in the menuentry. This allows fast access to the menu command via the keyboard when the user presses the `Alt`-key in conjuction with the underlined letter.

Finally, we're giving the menuentry an ID, which is an integer value by which we can find the entry later. To keep an overview over the used values, these are defined by macros and are collected in the file `resource.h` within your project. For consistency, these macros are all uppercase and begin with ID_, then the menu name followed by the entry. This makes it very easy to remember the sense of each entry anywhere within the code, so you don't have to turn to the menubar implementation again to look up the entries.

The second example entry shows another variant of the `insertItem()` method. Here, we add the recent_files_menu popup menu as a menuitem. This means, that the entry shows itself with the given string "Open recent", followed by a right arrow. On selection, the recent file popup menu appears and the user can choose the last file.

Last but not least there are a lot of other ways to insert menu items- the framework keeps this as simple as possible. More information can be obtained in the Qt documentation about the `QMenuData` class.

Now, after we created the popup menus `file_menu, edit_menu` and `view_menu`, we have to include a "Help"-menu as well. We could do this like the others as well, but the `KApplication` class offers a nice and quick method to cover this:

```
help_menu = kapp->getHelpMenu(true, i18n("KScribble\n" VERSION ));
```

This is all we have to do to get a help menu that contains an entry for the help contents with the F1 keyboard shortcut, an about-box for the application and an about-box for the KDE (which can be disabled by calling `getHelpMenu(false,...);`). The contents for our applications about-box is set with the `i18n()` string again- VERSION takes the macro that is defined for the project version number in the file `config.h`, so we don't have to change this every time manually when we want to give out a new release. Feel free to add any information about your application here, e.g. your name, email address, copyright and the like.

Now we only have to insert the pop-ups into the menubar. As `KTMainWindow` already constructs a menubar for us, we just insert them by calling `menuBar()->insertItem();`.

What is left to do is to connect the menu-entries with the methods they will execute. Therefore, we connect each popup menu by its signal `activated( int )` to a method `commandCallback( int )`, which contains a `switch` statement that calls the according methods for the menu entries. Additionally, we connect the pop-ups by their signal `highlighted( int )` to provide statusbar help on each entry. Whenever the user moves his mouse or keyboard focus to an entry, the statusbar then shows the according help message.

After we finished with the menubar, we can continue with the toolbar in the following section. Mind that an instance of a `KTMainWindow` can only have one menubar visible at a time; therefore if you want to construct several menu bars, you have to create them separately with instances of `KMenuBar` and set one of them by the according methods of `KTMainWindow` as the current menubar. See the class documentation of `KMenuBar` for more detailed information about how to extend the features, also see 5 (Configuring Menubars and Toolbars).

The Toolbar

The creation of toolbars now is even simpler than that of menubars. As `KTMainWindow` already provides toolbars, which are created by the first insertion, you are free to create several ones. Just add the buttons for the functions you want to provide:

```
toolBar()->insertButton(Icon("filenew.xpm"), ID_FILE_NEW, true, i18n("New File") );
```

This adds a left-aligned button with the icon "filenew.xpm" with the according ID to the toolbar. The third parameter decides if the button should be enabled or not; by default we set this to `true`, because our `disableCommand()` methods at the end of the constructor do this for us automatically for both menu and toolbar entries. Finally, the last parameter is uses as a so-called "Quick-Tip"- when the user moves the mouse pointer over the button so that it gets highlighted, a small window appears that contains a short help message, whose contents can be set here.

Finally, all toolbar buttons are connected to our `commandCallback()` method again by their signal `clicked()`. On the signal `pressed()`, we let the user receive the according help message in the statusbar.

Additional Information:

As toolbars are created using the class `KToolBar`, you should have a look at the according documentation. With `KToolBar`, a lot of things needed in a toolbar can be realized such as delayed pop-ups if your button wants to pop up a menu when the button keeps being pressed or even widgets like combos. Also, by default, the toolbar fills the complete width of the window, which makes it look nice for using a single bar. When using more than one, you should also think about setting the bar size to end at the most right button, so other bars can be displayed in the same row below the menubar. We will discuss certain techniques about designing and extending toolbars in section 5 (Configuring Menubars and Toolbars).

The Statusbar

The statusbar is, as well as the other bars, already provided by the `KTMainWindow` instance, so we just have to insert our items as we want to. By default, the framework contains only one entry that displays statusbar help. For a lot of applications this may not last; then you would add the entries you need for displaying e.g. coordinates and the like.

Also, an application can only have one statusbar at a time like a menubar. If you want to construct several ones, you should create them separately and set the current bar by the according method of `KTMainWindow`. The statusbar also offers to insert widgets, which can be used to produce nice habits for displaying progress-bars like KDevelop does. Refer to the class documentation of `KStatusBar`.

Keyboard Accelerators

With reaching the method `initKeyAccel()`, we already constructed the standard items of an application main window- the menubar, toolbar and statusbar. Indeed, we didn't set any keyboard accelerators by which advanced users that only want to work with the keyboard have a quick access to certain commands that are used most often during work with our program. To do this, we could have inserted the accelerator keys by the insertion of the menu-items for example, but KDE offers a good solution to construct and maintain keyboard accelerators. A lot of users want to have them configurable on one hand and on the other standard accelerators should be the same over all applications. Therefore, the KDE control center offers configuring standard keyboard accelerators globally by using the `KAccel` class. Additionally, the KDE libraries contain a widget that lets users configure

application specific keyboard shortcuts easily. As the application framework only uses menu-items that have standard actions such as "New" or "Exit", these are set by the method `initKeyAccel()`. Standard actions just have to be connected, for your application specific keyboard values, you have to insert them first by specifying the keyboard accelerator name and then connect them. As our accelerators are all present in the menubar, we have to change the accelerators for the popup entries. Finally we call `readSettings()`, which reads the current settings from the root window of KDE containing the configuration of standard accelerators, then the settings for accelerators specified in the application's config file. When we're going further into our example project, we will also talk about how to configure our application specific accelerators by a configuration dialog, see 5 (Configuring Menubars and Toolbars) for that part of the development process.

The Document-View Model

The next two member function calls, `initDocument()` and `initView()`, are finally building the part that the application windows are supposed to provide to the user: an interface to work with data that the application is supposed to manipulate; and that is also the reason why the application framework contains three classes, an `*App`, `*View` and `*Doc` class. To understand, why this structure is helpful, we'll look a bit aside the actual code and introduce some theory, then we'll switch to the program again to see how the KDevelop frameworks support such a model.

Basically, all what has been explained about the framework is that we need an application instance that contains a main window. This window is responsible to provide the basic interface for the user- it contains the menubar, toolbars and statusbar and the event controlling for user interaction. Also, it contains an area, that is described as a "view". Now, the purpose of a view is generally, to display the data that the user can manipulate, e.g. a part of a text file. Although the text file is probably larger than the view is able to display on the screen, it offers the user to go to the part that he wants to see (therefore it is a view), and there the user can as well change the data of the file contents. To give the programmer a better way to separate parts of the application by code, the Document-View Model has been invented. Although not a standard, it provides a structure how an application should work:

- The application contains a controller object,
- a View object that displays the data the user works with
- and a Document object that actually contains the data to manipulate.

Back to the example of working with a text file- there, this model would work the way that the Document would read the file contents and provides methods to change the data as well as to save the file again. The view then processes the events that the user produces by the keyboard and the mouse and uses the document object's methods to manipulate the document data.

Finally, the controller object is responsible for user interaction by providing the document and the view objects as well as the interfaces to send commands like opening and saving. Additionally, certain methods of the view object can be provided by commands that can be accessed via keyboard accelerators or the mouse on menubars and toolbars.

This Document-View model has some advantages- it separates the program's code more object-oriented and by this offers more flexibility in general, e.g. the same document object could be displayed by two views at the same time; either by a new view in a new window or by tiling the current one that then contains two view object that build the actual window view region.

Now, if you're coming from MS-Windows systems you may have some experience with that- the MFC already provide a document model that is ready to use. For KDE and Qt applications, things

are a bit different. Qt is a powerful toolkit as it provides the most needed classes, widgets etc. But there wasn't any intention to take care of the document-view model, and as KDE is inheriting Qt, there weren't any tendencies to introduce such a model either. This somehow has its reason in the fact that usually X-applications don't work with an MDI (Multiple Document Interface). Each main window is responsible for its data and that reduces the need of a document model to the fact that methods to work on documents are always implied into widgets. The only exception from this currently is the *KOffice* project that is intended to provide a complete office suite of applications like a word processor, a spreadsheet etc. Technically, this is realized by two changes two the normal usage of Qt and KDE:

- KOffice uses KOM and the free MICO implementation of CORBA for object communication,
- the KOffice applications use a document-view model to allow all applications to work with any KOffice data objects

But as KDevelop currently targets on using the current libraries of KDE 1.1.x and Qt 1.4x, we can't use this model by default- this will come in further releases of a KDE 2, which will (hopefully) contain two new major changes in relation to the current situation:

1. an MDI interface for KTMainWindow
2. the KOM libraries that provide a document model

Therefore, the current way for application developers can be to either implement all needed document methods within their view or to try to reproduce a document model by themselves. KDevelop therefore contains such a reproduction by providing the needed classes and the basic methods that are generally used for a Document-View model with the application frameworks for Qt and KDE.

Back to the code, you now can imagine the purpose of the two methods we mentioned at the beginning of this section: the `initDocument()` and `initView()` functions. The `initDocument()` constructs the document object that represents the application window data and initializes the basic attributes like setting the modification bit that indicates if the data currently used has been changed by the user. Then, the `initView()` method constructs the `*View` widget, connects it to the document and calls the `setView()` method of `KTMainWindow` to tell the `*App` window to use the `*View` widget as it's center view.

For the developer, it is important to know that during the development process he has to:

- re-implement the virtual methods for mouse and keyboard events provided by `QWidget` in the `*View` object to provide the means to manipulate data,
- re implement the `paintEvent()` of `QWidget` in the `*View` object to repaint() the view after changes,
- complete the implementation for printing the document via the printing method of the `*View` object,
- add the serialization for the `*Doc` object to provide file loading and saving,
- add the document data structure implementation to the `*Doc` object that is representing the document data logically in the memory.
- add any methods that have to be accessible by the user via accelerator keys and menus/toolbars.

Application Configuration

Now, after we created all instances of the `KTMainWindow` instance of our application to create the first window, we have to initialize certain values that influence the look of the program. For this, we call `readOptions()`, which gets all values and calls the methods needed to set the according attributes. The KDE-Core library contains the class `KConfig` that provides a good possibility to store values in configuration files as well as to read them in again. Also, as each `KApplication` instance creates it's resource file already, we only have to access this file and create our values. As `KConfig` provides us the file object, we have to use the class `KConfigBase` to read and write all entries. As writing is very easy to do with `writeEntry()` methods, reading depends on the attribute type which we want to initialize. Generally, an entry in the configuration file contains a value name and a value. Values that belong together in some context can be collected in groups, therefore we have to set the group name before we access the value afterwards; the group has to be set only once for reading a set of attributes that are in the same group.

Let's have a look at what we want to read in:

```
1   void KScribbleApp::readOptions()
2   {
3
4      config->setGroup("General Options");
5
6      // bar status settings
7      bool bViewToolbar = config->readBoolEntry("Show Toolbar", true);
8      view_menu->setItemChecked(ID_VIEW_TOOLBAR, bViewToolbar);
9      if(!bViewToolbar)
10       enableToolBar(KToolBar::Hide);
11
12     bool bViewStatusbar = config->readBoolEntry("Show Statusbar", true);
13     view_menu->setItemChecked(ID_VIEW_STATUSBAR, bViewStatusbar);
14     if(!bViewStatusbar)
15       enableStatusBar(KStatusBar::Hide);
16
17     // bar position settings
18     KMenuBar::menuPosition menu_bar_pos;
19     menu_bar_pos=(KMenuBar::menuPosition)config->readNumEntry("MenuBar Position", KMenuBar::Top);
20
21     KToolBar::BarPosition tool_bar_pos;
22     tool_bar_pos=(KToolBar::BarPosition)config->readNumEntry("ToolBar Position", KToolBar::Top);
23
24     menuBar()->setMenuBarPos(menu_bar_pos);
25     toolBar()->setBarPos(tool_bar_pos);
26
27     // initialize the recent file list
28     recent_files.setAutoDelete(TRUE);
29     config->readListEntry("Recent Files",recent_files);
30
31     uint i;
32     for ( i =0 ; i < recent_files.count(); i++){
33       recent_files_menu->insertItem(recent_files.at(i));
34     }
35
36     QSize size=config->readSizeEntry("Geometry");
37     if(!size.isEmpty())
38       resize(size);
```

```
39   }
```

As we have seen in one of the above code parts, the first action our constructor does was:

```
config=kapp->getConfig();
```

which sets the `KConfig` pointer `config` to the application configuration. Therefore, we don't have to care for the location of the configuration file. Indeed, the file is, according to the KDE File System Standard (KDE FSS), located in `$HOME/.kde/share/config/`; we will have a closer look about the KDE FSS in a later step when we're setting installation locations for project files. As the config file is placed in the user's home directory, each user has it's own appearance of his application except for values that are located in a system wide configuration file that can optionally be created and installed by the programmer in the KDE directory. But, although this could help in some cases, we should avoid any dependency of our application towards the existing of file entries. Therefore, all read methods provided by `KConfigBase` allow to add a default value to be used when the entry doesn't exist. Another thing important to a programmer is that the configuration file is stored in plain text, and this is for some reasons as well as you have to watch some criteria:

- the user is able change the configuration file by a plain text editor
- if the user wants to change values by hand, the entries should be very transparent to determine their purpose
- for entries that have to be saved, but are critical in terms of security like passwords, you have to look for a proper solution to ensure the security.

Now that we know the basics, we're going to analyze the code. As said, we only have to use our config pointer to access the values. First, in line 4, we set the current group to "General Options". This indicates that the values used are somewhat general attributes for the application. Then we read the values for the toolbar and statusbar- these have to be saved when the application closes to restore their status again when the user restarts the program. As the bars can only be on or off, we use a boolean value, therefore, our method is `readBoolEntry()`. The process is identical for both bars, so we only have a look at the lines 7-10 to watch what's happening for the toolbar. First, we read the value into the temporary variable `bViewToolbar` at line 7. The value name in the file is "Show Toolbar" and, if the value is not present (which would be the case the first time the application starts), the default value is set to `true`. Next, we set the checkmark for the menuentry for en-/disabling the toolbar by this value: we call `setItemChecked()` on the view menu, entry `ID_VIEW_TOOLBAR` with our attribute. Finally, we set the toolbar to use the value. By default, the toolbar is visible, therefore, we only have to do something if `bViewToolbar` is `false`. With `enableToolBar()` (line 10) we're setting the bar to hide itself if it is disabled.

Next, we have to read the bar positions. As the user might have changed the bar position by dragging a bar with the mouse to another view area, these have to be saved as well and their status restored. Looking at the classes `KToolBar` and `KMenuBar`, we see that the bar positions can be:

```
enum BarPosition {Top, Left, Bottom, Right, Floating, Flat}
```

As this value has been written in a numeric value, we have to read it with `readNumEntry()` and convert it to a position value. With `setMenuBarPos()` and `setBarPos()` we tell the bars where to show up.

Now you probably have noticed that our "File" menu contains a menu for recently used files. The filenames are stored in a list of strings, which has to be saved on application closing and now has

to be read in to restore the menu. First, we initialize the list with the entries stored by using the `readListEntry()`. Then, in a `for`-loop, we create a menu entry for each list item.

Finally, we only have to take care for the geometry of our window. We read in the appearance by a `QSize` variable containing an x and y value for width and height of the window. As the window is initialized by `KTMainWindow`, we don't have to take care for a default value and only will use `resize()` if the entry is not empty.

What is left to explain on application construction is that we initially have to disable available user commands that shouldn't be available if some instances don't match the needed criteria. These are file saving and operations that are using the clipboard. During the application's lifetime, we have to take care of these several times, but which is quite easy. The framework only gives us two methods to enable/disable menubar and toolbar items with one method call at the same time.

Executing

During the past section, we have only monitored what happens during the constructor call of our `KScribbleApp` instance providing us the main window. After returning to the `main()` function, we have to call `show()` to display the window. What is different from any `KApplication` or `QApplication` here is that when we're using `KTMainWindow` as the instance for our main widget, we don't have to set it with `setMainWidget()`. This is done by `KTMainWindow` itself and we don't have to take care of that. The only thing left then is to interpret the command-line. We get the command-line option and ask, if `int argc` is > 1, which indicates that the user called our application with `kscribble filename_to_open`. Our window is then asked to open the file by it's name and calls `openDocumentFile()` with the filename.

The last line of the `main()` function does the known job: it executes the application instance and the program enters the event loop.

Now, in section 3.3.1 (The main() Function), we started to separate the execution process by `if( app.isRestored() )` and described the usual invocation process. The following now gives an introduction to session management and how our application makes use of this.

3.3.3 Invocation by Session Management

As we said, the `main()` function tests, if the application is invoked by the session manager. The session manager is responsible to save the current status of all open application windows on the user's desktop and has to restore them when the user logs in the next time, which means that the application is not started by the user but automatically invoked. The part of the code which is executed was:

```
6    if (app.isRestored())
7    {
8       RESTORE(KScribbleApp);
9    }
```

In 3.3.1 (The main() Function), we stated that we test the invocation by asking `app.isRestored()`. Then line 8 gets executed. It looks like a simple statement, but in fact this will result in a complex execution process which we want to follow in this section.

RESTORE() itself is a macro provided by `KTMainWindow`. It expands to the following code:

```
if (app.isRestored()){
  int n = 1;
```

```
  while (KTMainWindow::canBeRestored(n)){
    (new KScribbleApp)->restore(n);
    n++;
  }
}
```

This will restore all application windows of the class `KScribbleApp` by creating the instances and calling `restore()` to the new window. It is important to realize that if your application uses several different widgets that inherit `KTMainWindow`, you have to expand the macro and determine the type of the top widgets by using `KTMainWindow::classNameOfToplevel(n)` instead of the class `KScribbleApp`. The `restore()` method then reads the part of the session file that contains the information about the window. As `KTMainWindow` stores all of this for us, we don't have to care for anything else. Only information that belong to our specific instance of `KScribbleApp` has to be found then. Usually this would be a temporary file that we created to store the document or other initialization that we might need. To get to this restoration information, we only have to overwrite two virtual methods of `KTMainWindow`, `saveProperties()` and `readProperties()`. The information we have to save on session end is if the currently opened document is modified or not and the filename. If the file is modified, we will get a temporary filename to save it to. On session beginning, this information now is used to restore the document contents:

```
void KScribbleApp::readProperties(KConfig*)
{
  QString filename = config->readEntry("filename","");
  bool modified = config->readBoolEntry("modified",false);
  if( modified ){
    bool b_canRecover;
    QString tempname = kapp->checkRecoverFile(filename,b_canRecover);

    if(b_canRecover){
      doc->openDocument(tempname);
      doc->setModified();
      QFileInfo info(filename);
      doc->pathName(info.absFilePath());
      doc->title(info.fileName());
      QFile::remove(tempname);
    }
  }
  else if(!filename.isEmpty()){
  doc->openDocument(filename);
  }
  setCaption(kapp->appName()+": "+doc->getTitle());
}
```

Here, the line `kapp->checkRecoverFile()` seems a bit strange, as `b_canRecover` is not initialized. This is done by the method which sets it to `true`, if there is a recover file. As we only saved a document in a recover file if it was modified, we set the modified bit directly to indicate that the information hasn't been saved to the belonging file. Also we have to take care that the recover file has another filename than the original file which was opened. Therefore, we have to reset the filename and path to the old filename. Finally, we have the information we wanted to recover and we can delete the temporary file by the session manager.

Summary:

During this chapter, you got to know how the application starts either by normal user invocation or by the session manager. We went through the code to learn how the parts of the visual interface

of the application are constructed as well as how to initialize attributes by configuration file entries. Now you can execute the framework application to test these functions and see how the program window reacts.

3.4 Additional Contents of KDevelop Projects

Besides the source code provided, KDevelop projects contain a lot of other additional parts that are of interest to the developer. These are:

- a program icon
- a program mini-icon
- a .kdelnk file
- a sample SGML-documentation file
- a set of API-documentation generated from the framework source

Except the API-documentation, these elements of the project will be installed together with the application binary. As the project framework has to be as open as possible, you have to adapt these parts towards your project goals. These are first to edit the icons provided. This will give your application a unique identifier by which the user can determine your application visually in window manager menus. The .kdelnk file then is a file that installs your application into `kpanel` in the `Applications` menu. This has to be edited by setting the installation path which will be discussed later in this handbook. Finally, the documentation that you will provide to the user is written in SGML. This makes it very easy to create several different output from the same source. By default, KDevelop offers to create a set of HTML files from this source, for KDE-projects this will automatically use the `ksgml2html` program to add a consistent KDE look and feel to the documentation. In a later section, we will see how the SGML source is edited and what we have to watch for installation on the end-user.

Finally, the API (Application Programming Interface) documentation allows you and other developers to quickly get into the code and use the classes without having to guess what purpose each class is for. We will learn how to extend the API documentation in a later step, for now it lasts to know that the documentation is generated by the `KDoc` program, which processes the header files and creates the HTML output, therefore all documentation is placed in the headers.

Chapter 4

Application View Design

When developing an application with a graphical user interface, the main work takes place in providing a so-called "view" for the application. A view generally is a widget that displays the data of a document and provides methods to manipulate the document contents. This can be done by the user via the events he emits by the keyboard or the mouse; more complex operations are often processed by toolbars and menubars which interact with the view and the document. The statusbar then provides information about the document, view or application status. As an example, we look at how an editor is constructed and where we can find which part.

An editor generally is supposed to provide an interface to view and/or change the contents of a text document for the user. If you start *KEdit*, you see the visual interface as the following:

- The menubar: providing complex operations as well as opening, saving and closing files and exiting the application.
- The toolbar: offers icons which allow quicker access for most needed functions,
- The statusbar: displays the status of the cursor position by the current row and column,
- The view in the center of the window, displaying a document and offering a cursor connected to the keyboard and the mouse to operate on the data.

Now it's easy to understand, that a view is the most unique part of the application and the design of the view decides about the usability and acceptability of an application. This means that one of the first steps in development is to determine the purpose of the application and what kind of view design would match best to allow any user to work with the application with a minimum of work learning how to handle the user interface.

For some purposes like text editing and displaying HTML files, views are provided by the Qt and KDE libraries; we will discuss certain aspects of these high-level widgets in the next section. But for most applications new widgets have to be designed and implemented. It is that what makes a programmer also a designer and where his abilities on creativity are asked. Nevertheless, you should watch for intuitivity first. Remember that a lot of users won't accept an application that isn't

- graphically nice,
- offering a lot of features,
- easy to handle,
- fast to learn how to use it.

Needless to say that stability is a major design goal. Nobody can prevent bugs, but a minimum can be reached at least by clever design goals and wide use of object-oriented design. C++ makes programming a joy if you know how to exploit it's capabilities- inheritance, information hiding and reusablitity of already existing code.

When creating a KDE or Qt project, you always have to have a view that inherits `QWidget`, either by direct inheritance or because the library widget you want to use inherits `QWidget`. Therefore, the Application Wizard already constructed a view that is an instance of a class `<yourapp>View`, which inherits `QWidget` already. The application creates your view in the method `initView()`, where an instance is created and connected to the main widget as it's view with `KTMainWidget::setView()`.

This chapter therefore describes how to use library widgets for creating views of KDE or Qt applications that are generated with KDevelop, then we're looking at the libraries and what kind of views are already offered.

4.1 Using Library Views

When your application design has been set up, you first should look for already existing code that will make your life a lot easier. A part of this search is to look for a widget that can be used as a view or at least as a part of it; either directly or by inheritance. The KDE and Qt libraries already contain a set of widgets that can be used for this purpose. To use them, you have two options:

1. remove the new view class and create an instance of a library widget; then set this as the view,
2. change the inheritance of the provided view class to the class of the library widget to use.

In either way, it is important to know that if the application framework is currently not linked against the library that contains the widget, the linker will fail. After you decided to use a certain widget, look for the library to link to; then open "Project"->"Options" from the KDevelop menubar. Switch to the "Linker Options" page and look for the checkmarks indicating the libraries that are currently used. If the library of your view widget is already checked, you can leave the project options untouched and start doing the necessary changes due to your choice. If not, and the linker options offer to add the library by a check box, check it and press "OK" to leave the project options dialog again. In any other case, add the library in the edit line below with the `-l` option. For libraries that your application has to search for before preparing the Makefiles by the `configure` script on the end-user machine, add the according search macro to the `configure.in` file located at the root directory of your project and add the macro to the edit line. Mind that you have to run "Build"->"Autoconf and automake" and "Build"->"Configure" before the Makefiles contain the correct expansion for the library macro.

Also, if the include files for the library to add are not in the current include path (which can be seen by the `-I` options in the output window on "Make"), you have to add the path to the Project Options dialog -"Compiler Options" page with the `-I` option or the according automake macro at the edit line for "Additional Options".

4.1.1 Qt Views

Looking at the first page of the Qt online documentation, you will find a link to "Widget Screenshots" where you can have a look at how the widgets Qt contains look like. These are ready to use and can be combined together to form complex widgets to create application views or dialogs. In the following, we'll discuss some of these which are very usable for creating application views, but keep

in mind that the KDE libraries sometimes contain other widgets for the same purpose; those will be reviewed in the next section.

Here are a set of hints for what purpose you could use which Qt component:

1. if your view area isn't big enough to display all your data, the user must be enabled to scroll over the document with bars on the left and bottom of the view. For this, Qt provides the class `QScrollView`, which offers a scrollable child area. As explained, you could inherit your own widget from `QScrollView` or use an instance to manage your document's view widget.

2. to create a ScrollView yourself, inherit the View widget from `QWidget` and add vertical and horizontal `QScrollBar`s. (this is done by KDE's KHTMLView widget).

3. for text processing, use `QMultiLineEdit`. This class provides a complete text editor widget that is already capable to cut, copy and paste text and is managed by a scrollview.

4. use `QTableView` to display data that is arranged in a table. As `QTableView` is managed by scrollbars as well, it offers a good solution for table calculation applications.

5. to display two different widgets or two widget instances at the same time, use `QSplitter`. This allows to tile views by horizontal or vertical dividers. Netscape's Mail window is a good example how this would look like- the main view is separated by a splitter vertically, the right window then is divided again horizontally.

6. `QListView` displays information in a list and tree. This is useful for creating file trees or any other hierarchical information you want to interact with.

You see that Qt alone offers a whole set of widgets which are ready to use so you don't have to invent new solutions if these match your needs. The sideffect when using standard widgets is that users already know how to handle them and only have to concentrate on the displayed data.

4.1.2 KDE Views

The KDE libraries were invented to make designing applications for the K Desktop Environment easier and capable of more functionality than what Qt alone is offering. To see what's available, we have a look at the documentation tree in KDevelop. You see that the KDE libraries start with `kdecore`, which is a base for all KDE applications. Then,`kdeui` offers user interface elements. This is where we will find some useful things first. For creating new applications, the `kdeui` library offers:

1. `KTabListBox`: offers a multi-column list box where the user can change the rows with drag'n drop.

2. `KTreeList`: inherited from `QTableView`, offering a collapsible tree. This could be used instead of `QListView`.

3. `KEdit`: the base classes for the *KEdit* application offered with KDE. This could be used instead of `QMultiLineEdit`.

4. `KNewPanner`: manage two child widgets like `QSplitter`.

The `khtmlw` library on the other hand offers a complete HTML-interpreting widget that is ready to use. It is scrollable already, so you don't even have to take care for that. A possible use could be to integrate it as a preview widget for an HTML editor; used by applications such as KFM, KDEHelp and KDevelop to display HTML files.

4.2 Creating your own Views

Now that you have a general overview of what is already provided, you may notice that for a lot of purposes already existing widgets can be used or combined together. KMail is an example as well as KDevelop itself makes use of library view components to display data.

For applications that use a special file format or have to deal with graphics, you are probably forced to create your own view widget to allow data manipulation. This is realized in our sample by the class `KScribbleView`, already providing a base for a view area.

The inheritance from `QWidget` is necessary to overwrite the virtual methods to process user events, this is probably the most work besides providing popup menus for easier access of certain functions. Also it is likely that you have to implement a set of slots which can be accessed by toolbar buttons or menu bar commands to connect to as well as methods to manipulate variables such as e.g. a painter color.

For completeness, we will repeat the necessary methods:

a) Keyboard events –TAB and Shift-TAB keys:

changes the keyboard input focus from the current widget to the next widget in the focus order. The focus can be set to widgets by calling `setFocusPolicy()` and process the following event handlers:

- `virtual void focusInEvent ( QFocusEvent * )`
- `virtual void focusOutEvent ( QFocusEvent * )`

b) all other keyboard input:

- `virtual void keyPressEvent ( QKeyEvent * )`
- `virtual void keyReleaseEvent ( QKeyEvent * )`

c) mouse movements:

- `virtual void mouseMoveEvent ( QMouseEvent * )`
- `virtual void enterEvent ( QEvent * )`
- `virtual void leaveEvent ( QEvent * )`

d) mouse button actions:

- `virtual void mousePressEvent ( QMouseEvent * )`
- `virtual void mouseReleaseEvent ( QMouseEvent * )`
- `virtual void mouseDoubleClickEvent ( QMouseEvent * )`

e) window events containing the widget:

- `virtual void moveEvent ( QMoveEvent * )`
- `virtual void resizeEvent ( QResizeEvent * )`
- `virtual void closeEvent ( QCloseEvent * )`

When re-implementing these functions, you should watch certain issues to avoid implementation mistakes that will make it almost impossible to change the widget's behavior afterwards:

1. declare your virtual methods as **virtual** as well and keep the access to protected. This allows code-reuse by inheritance and is consistent.
2. don't hard-code any event-processing which should be made configurable. This counts most for keyboard events which should be realized with keyboard accelerators if any function is called. This even counts for text processing ! (Imagine that a lot of users are familiar with their favorite editor's behavior. If this is configurable, they can use the behavior they like and are used to)
3. forward popup menu highlighting signals to the main widget to enable statusbar help

Chapter 5

Configuring Menubars and Toolbars

Menubars and toolbars are one of the most important parts of an application to provide methods to work with a document structure. As a general rule, you should make all functions available by the menubar. Those methods that should not be available at a current stage of the application process should be disabled.

Further, an application can only contain one menubar, but several toolbars. Toolbars on the other hand should contain only the most frequently used commands by pixmap icons or provide quick access methods like combos to select values.

5.1 How does it work ?

Each entry, may it be a menuentry or a toolbar item, has a resource ID which is an integer value. As these values can't be used twice, those are set by macros, where the numeric values are replaced by a descriptive ID name that can be used in your sources then.

All resource ID's are collected in the file `resource.h`, where you can keep an overview over the used values. Anyway, the compiler will inform you if you've used a value twice for constructing entries. Also, the resource file should contain all menu accelerators by IDK macro replacements. An example:

```
(resource.h)

#define ID_VIEW_TOOLBAR             12010


(kscribble.cpp)

// menu entry Toolbar in the "view" menubar menu
view_menu->insertItem(i18n("&Toolbar"), ID_VIEW_TOOLBAR);
```

This inserts the entry Toolbar to the View popup menu of the menubar in the kscribble application. The resource ID's name is held to contain the menu name and the action's name visible. The ampersand is set in front of the letter that functions as a keyboard accelerator and the entry itself is embraced by the i18n() internationalization macro.

On activating the menu item, the commandCallback() switch is called with the ID number. There, you have to add an according comparator value with the method you want to execute on activating the menuentry:

```
case ID_VIEW_TOOLBAR:
    slotViewToolBar();
    break;
```

Note: you don't have to use the ID system. If no ID is given, the menu gets numbered automatically. The KDevelop framework uses this as it allows accessing menu and toolbar ID's to create switch statements that select the slot to call on activated() for menus, clicked() for toolbar buttons. The connection can also be made directly using the provided methods of the classes providing menus and toolbars.

5.2 Adding a new menu

A new menubar is added to an application by the following:

1. add a pointer to the new menu in the App-class header
2. call the constructor of `QPopupMenu` to the pointer in `initMenuBar()` at the location where your menubar should appear later.
3. insert the according menu-items into the popup menu and set their resource ID's in the resource file
4. add connects for `commandCallback()` and `statusCallback()` to the menu at the end of `initMenuBar()`
5. add the methods you want to call by the menu-entries in the header and implementation file.
6. add the switch statements for the entries to the `commandCallback()` and `statusCallback()` methods

5.3 Integrating Toolbar buttons

Toolbar buttons can be added like menu-entries with the difference that the used method is `insertButton()` and takes a button pixmap and tool-tip text instead of a menu text.

The icons you want to use can be loaded by `KIconLoader`, where `KApplication` offers the macros `ICON()` and `Icon()` to access the icon loader and load the icon. These macros take the filename of the pixmap as their parameter to load the icon from the KDE file system in a certain order (see `KIconLoader` for the search order).

The KDE libraries also offer a set of toolbar buttons that can be used for standard actions. In cases where they don't meet your needs, you will have to paint your own pixmaps. KDevelop supports this by selecting "New" from the "File" menu, then select Pixmap as the file type. Usually you will place your toolbar pixmaps in a project subdirectory "toolbar" and install them into your application specific toolbar directory.

5.4 Configuring Statusbars

The KDevelop projects already make use of the statusbar by providing statusbar messages for menuentries and toolbar buttons. When adding a menuentry, also add your status message in the method `statusCallback()`.

`statusCallback()` uses the method `slotStatusHelpMsg()` to display a statusbar message for two seconds. When executing a command, you should use the method `slotStatusMsg()` at the beginning with the string describing what your application does; before a return or method end, you should reset the statusbar message with a "Ready." string calling the same method.

5.5 Keyboard Accelerator Configuration

A very professional thing you should always add to your application are keyboard accelerators. Those are mainly used by experienced users that want to work fast with their applications and are willing to learn shortcuts. For this, the KDE libraries provide the class `KAccel`, which provides the keyboard accelerator keys and access to global configured standard keyboard accelerators.

By default, frame applications generated by KDevelop only use standard keyboard accelerators such as F1 for accessing online-help, Ctrl+N for New File etc. You should look for the keyboard accelerators already set in `KAccel` first before adding a new accelerator.

If your application contains a lot of accelerators, you should make them configurable by an Options-menu; either it could be combined with other application configuration in a `QWidget` or stand alone. The KDE library already provides a `KKeyChooser` for use in tab dialogs, whereas `KKeyDialog` provides a ready-to use key-configuration dialog.

See the following classes for more information:

`KAccel`(kdecore), `KKeyChooser`, `KKeyDialog` (kdeui)

Chapter 6

The Dialogeditor: Where your Dialogs are Build

6.1 What the Dialogeditor provides

The built-In dialogeditor of KDevelop is designed to help you construct widgets and dialogs that fit your application's purpose and reduces the time rapidly to extend the GUI of your application. The only limitation for now is that the dialogeditor does not support geometry management that is provided by Qt; therefore the dialogs are static in their size and this may lead to certain circumstances where e.g. the label width is not long enough to support the full length of a translation.

On the other hand, the current state of the editor in conjunction with KDevelop's project management offers the fastest way to create full-featured applications for the K Desktop Environment.

6.2 Qt and KDE Widgets

Currently provided widgets are:

QT-Widgets:

- **QWidget** - a widget that can be specified by yourself and can contain other widgets as well. This allows creating a widget hierarchy within your dialog.
- **QLabel** - a label that represents text information on the widget. Use QLabel e.g. in front of lineedits to signal what the purpose of the line-edit is or which variable e.g. a combo box allows to set.
- **QPushButton** - a button that allows to e.g. call another dialog like QFileDialog for selecting a filename.
- **QCheckBox** - a check box for e.g. enabling/disabling options. QCheckBox is widely used for configuration dialogs.
- **QLCDNumber** - displays numbers in LCD style. Often used for clocks.
- **QRadioButton** - like QCheckBox often used to let the user set any options. QRadioButton specializes the options setting when more of them depend on each other, e.g. you have three radio-buttons, but you want the user to choose one of three offered options. See `QButtonGroup` for additional information.

- **QComboBox** - a combo box lets the user set a value by selecting it from a drop-down menu or by inserting the value, if the box is write enabled.
- **QListBox** - provides a single-column list of items that can be scrolled.
- **QListView** - creates a multi-column list view that can be used to display e.g. file trees etc. in tree and table view.
- **QMultiLineEdit** - offers a multi-line editor.
- **QProgressBar** - displays the progress of an action that takes a longer time to be finished.
- **QSpinBox** - allows choosing numeric values by up- and down buttons or insertion if write enabled.
- **QSlider** - sets a value in a program-defined range by a slider.
- **QScrollBar** - indicates the range of a value and sets the current value by a slider as well as up- and down buttons; often used for widgets whose contents is larger than the actually visible view area. By using the scrollbar, the visible area can be changed to another part of the widgets' contents.
- **QGroupBox** - provides a group box frame with title to indicate that child widgets within the box belong together.

KDE-Widgets:

- **KColorButton** - a pushbutton displaying a selected color. On a button press, the KDE Color dialog is shown where the user can select another color. Often used for drawing applications or in any case where color values can be set.
- **KCombo** - similar to `QComboBox`. Lets the user choose a value by a drop-down list box.
- **KDatePicker** - a complete widget to get a date value by the user.
- **KDateTable** - a calendar table to select a date of a month. Used by `KDatePicker` to build the date picker dialog.
- **KKeyButton** - a button to select a key value. If the button is selected, it gets activated. Pressing a keyboard button will change the key value for the button which can be used to configure key-bindings.
- **KLed** - and LED (Light Emitting Diode) widget to display a certain state.
- **KLedLamp** - and LED lamp that also supports click actions
- **KProgress** - similar to `QProgressBar`, `KProgress` supports certain other values.
- **KRestrictedLine** - a `QLineEdit` that only accepts certain user input. This can be used to restrict access to certain data by password dialogs.
- **KSeparator** - a separator widget to be used in all cases where KDE applications require a separator to provide a unique look. Often used in dialogs to separate logical parts where `QGroupBox` doesn't fit.
- **KTreeList** - a collapsible list view to display trees similar to `QListView`.

6.3 Properties of Qt supported Widgets

The following chapter gives a complete overview over the currently supported widgets of the Qt toolkit. To achieve a better understanding of the properties, these are separated to their inheritance. As all of them inherit `QWidget`, this class is described first. All `QWidget` properties are available for all other widgets as well, so these are not listed for them again. For widget groups that inherit an abstract subclass of `QWidget` as their base-class, the base-classes' properties are listed first (though this class does not provide a widget in the dialogeditor itself). Then the widget properties for the available widget of the group contains the properties that are specific to it. For a better understanding the inheritance tree of the available widgets is listed below:

- 6.3.1 (<cdx/QWidget/)
  - 6.3.2 (<cdx/QButton/) (abstract)
    - 6.3.2 (QCheckBox)
    - 6.3.2 (QPushButton)
    - 6.3.2 (QRadioButton)
  - 6.3.3 (<cdx/QComboBox/)
  - 6.3.4 (QFrame) (abstract for now)
    - 6.3.4 (QGroupBox)
    - 6.3.4 (QLCDNumber)
    - 6.3.4 (<idx/QLabel/)
    - 6.3.4 (QProgressBar)
    - 6.3.4 (QScrollView) (abstract for now)
      - 6.3.4 (QListView)
    - 6.3.4 (QSpinBox)
    - 6.3.4 (QTableView) (abstract)
      - 6.3.4 (QListBox)
      - 6.3.4 (QMultiLineEdit)
  - 6.3.5 (QLineEdit)
  - 6.3.6 (QScrollBar)
  - 6.3.7 (QSlider)

6.3.1 QWidget Properties

`QWidget` is the base class for almost all widgets in Qt and KDE. Therefore widgets that inherit `QWidget` will allow to use the same settings in most cases.

- **Appearance:**
  - **BgColor:** Background color of the widget
  - **BgMode:** Background mode of the widget
  - **BgPalColor:** Color pallette for the background
  - **BgPixmap:** filename for a background pixmap
  - **Cursor:** Cursor over the widget
  - **Font:** Font for the widget

  - **MaskBitmap:** filename for a masking bitmap.

- **C++ Code:**
  - **AcceptsDrops:** if set to true, the widget item will accept drops by drag'n drop mechanisms (Qt drag'n drop protocol, not KDE 1.x !)
  - **Connections:** connects the item's signals to slots
  - **FocusProxy:** the item that gives its focus to this widget.
  - **HasFocus:** sets if the item has the focus by default. Mind that only one item per dialog can have this value as true
  - **ResizeToParent:** resizes the widget to its parent's size (not visible in editing mode)
  - **VarName:** Variable name of the item. Change this to names that describe the item's purpose.

- **General:**
  - **IsEnabled:** sets if the widget will accept user events
  - **IsHidden:** sets the item to be visible(false) or hidden(true)
  - **Name:** sets the name of the widget. Mind that the name is different from the VarName in C++ Code.

- **Geometry:**
  - **Height:** height of the item
  - **IsFixedSize:**
  - **MaxHeight:** maximum value for Height
  - **MaxWidth:** maximum value for Width
  - **MinHeight:** minimum value for Height
  - **MinWidth:** minimum value for Width
  - **SizeIncX:** pixel steps for resize actions to X direction
  - **SizeIncY:** pixel steps for resize actions to Y direction
  - **Width:** width of the item
  - **X:** position horizontal, counted from the left corner
  - **Y:** position vertical, counted up to down

6.3.2 QButton inherited widgets

`QButton` is an abstract widget class that provides properties common to buttons.

Inherits 6.3.1 (<cdx/QWidget/)

Inherited by 6.3.2 (QCheckBox), 6.3.2 (QPushButton) and 6.3.2 (QRadioButton) inherit `QButton`.

- **Appearance:**
  - **setPixmap:** sets the pixmap filename to use

- **General:**
  - **setText:** the text on labels, buttons and boxes, also pre-set text for lineedits.
  - **setAutoRepeat:** if enabled, the clicked() signal is emitted at regular intervals while the button is down. No effect on toggle buttons.
  - **setAutoResize:** Enables auto-resizing if TRUE. When auto-resize is enabled, the button will resizes itself whenever the contents changes.

QCheckBox Properties

Inherits 6.3.1 (<cdx/QWidget/) and 6.3.2 (<cdx/QButton/)

- **General:**
  - **isChecked: (setChecked)** defines is the checkbox is set checked on construction

QPushButton Properties

Inherits 6.3.1 (<cdx/QWidget/) and 6.3.2 (<cdx/QButton/)

- **General:**
  - **isAutoDefault: (setAutoDefault)** the auto-default button becomes the default push button if it receives the keyboard input focus.
  - **isDefault: (setDefault)** there can be only one default button and it is only allowed to use in a dialog (see `QDialog`). The default button emits `clicked()` if the user presses the Enter key.
  - **isMenuButton: (setIsMenuButton)** tells the button to draw a menu indication triangle if enabled. The menu has to be inserted separately.
  - **isToggleButton::(setToggleButton)** makes a push button a toggle button, so the button has a similar state as check boxes.
  - **isToggledOn: (setOn)** (public slot) switches a toggle button on.

QRadioButton Properties

Inherits 6.3.1 (<cdx/QWidget/) and 6.3.2 (<cdx/QButton/)

- **General:**
  - **isChecked: (setChecked)** defines is the radio button is set checked on construction

6.3.3 QComboBox Properties

Inherits 6.3.1 (<cdx/QWidget/)

(no additional properties for now)

6.3.4 QFrame inherited widgets

Inherits 6.3.1 (<cdx/QWidget/)

For now only used as an abstract class.

- **Appearance:**
  - Margin (setMargin): sets the margin, which is the distance from the innermost pixel of the frame and the outermost pixel of the contents.

QGroupBox Properties

Inherits 6.3.1 (<cdx/QWidget/) and 6.3.4 (QFrame)

- **General:**
  - **Title: (setTitle)** sets the group box title that is displayed in the box frame.

QLCDNumber Properties

Inherits 6.3.1 (<cdx/QWidget/) and 6.3.4 (QFrame)

- **General:**
  - **NumDigits:(setNumDigits)** sets the number of digits displayed in QLCDNumber
  - **Value: (display)** (public slot) sets the initial value for QLCDNumber

QLabel Properties

Inherits 6.3.1 (<cdx/QWidget/) and 6.3.4 (QFrame)

- **Appearance:**
  - Margin (setMargin): sets the margin, which is for QLabel the distance from the frame to the first letter of the label text, depending on the alignment of the label.
- **C++ Code:**
  - **Buddy: (setBuddy)** sets the buddy widget of the label.
- **General:**
  - **Text: (setText)** sets the label text.
  - **isAutoResize: (setAutoResize)** if TRUE, the label will resize itself if the contents changes. The top left corner is not moved.

QProgressBar Properties

Inherits 6.3.1 (<cdx/QWidget/) and 6.3.4 (QFrame)

- **General:**
  - **TotalSteps: (setTotalSteps)** (public slot) sets the total steps of the progress bar. During the iteration of your action to display the progress, you have to call setProgress(int) to advance the progress step displayed to (int).

QScrollView

Inherits 6.3.1 (<cdx/QWidget/) and 6.3.4 (QFrame)

Inherited by 6.3.4 (QListBox) (abstract for now)

Provides a scrollable widget that manages the display of a child widget by a vertical and horizontal scrollbar.

QListView Inherits 6.3.1 (<cdx/QWidget/), 6.3.4 (QFrame) and 6.3.4 (QListView)

Provides a list view to display hierarchical data either in a table or a tree. Manages itself by scrollbars through 6.3.4 (QScrollView).

- **Appearance:**
  - **ListViewFont:** (setFont()) Sets the font of the ListView items
  - **ListViewPalette:** (setPalette()) Sets the palette of the list view items
  - **TreeStepSize:** (setTreeStepSize(int)) Offset of pixels of a child item to its parent item
  - **hScrollBarMode:** Scrollbar mode provided by 6.3.4 (QScrollView) for the horizontal scrollbar
  - **isAllColumnsShowFocus:** (setAllColumnsShowFocus(bool)) displays focus on all columns of an item.
  - **isMultiSelection:** enables multi-selection of list items
  - **isRootDecorated:** enables the + and - decoration to open and close trees
  - **vScrollBarMode:**Scrollbar mode provided by 6.3.4 (QScrollView) for the vertical scrollbar
- **General:**
  - **Entries:** lets you insert a list of entries that are pre-set as QListViewItems.
  - **isAutoUpdate:**

QSpinBox Properties

Inherits 6.3.1 (<cdx/QWidget/) and 6.3.4 (QFrame)

- **General:**
  - **MaxValue:** the maximum value the user can choose
  - **MinValue:** the minimum value the user can choose
  - **Prefix:**
  - **Suffix:**
  - **Value:** the pre-set value when the widget is shown
  - **isWrapping:**

QTableView inherited widgets

Inherits 6.3.1 (<cdx/QWidget/), 6.3.4 (QFrame) and 6.3.4 (QTableView)

Inherited by 6.3.4 (QListBox) and 6.3.4 (QMultiLineEdit)

QListBox Properties Inherits 6.3.1 (<cdx/QWidget/), 6.3.4 (QFrame) and 6.3.4 (QTableView)

- **General:**
  - **isAutoBottomScrollBar: (setAutoBottomScrollBar)**
  - **isAutoScroll: (setAutoScroll)**

  - **isAutoScrollBar: (setAutoScrollBar)**
  - **isAutoUpdate: (setAutoUpdate)**
  - **isBottomScrollBar: (setBottomScrollBar)**
  - **isDragSelect: (setDragSelect)**
  - **isSmoothScrolling: (setSmoothScrolling)**

- **Geometry:**
  - **setFixedVisibleLines:** sets a fixed height for the widget so that the given number of text lines are displayed using the current font.

QMultiLineEdit Properties Inherits 6.3.1 (<cdx/QWidget/), 6.3.4 (QFrame) and 6.3.4 (QTableView)

- **General:**
  - **Text: (setText)** (public slot) sets the text of the widget.
  - **isAutoUpdate: (setAutoUpdate)** used to avoid flicker during large changes; the view is not updated if disabled.
  - **isOverWriteMode: (setOverwriteMode)** (public slot) sets overwrite enabled or disabled.
  - **isReadOnly: (setReadOnly)** (public slot) sets the widget text to read only; disables text input.
  - **isTextSelected: (selectAll)**(public slot) marks the whole text selected

- **Geometry:**
  - **setFixedVisibleLines:** sets a fixed height for the widget so that the given number of text lines are displayed using the current font.

6.3.5 QLineEdit Properties

Inherits 6.3.1 (<cdx/QWidget/)

- **General:**
  - **CursorPosition: (setCursorPosition)** sets the default cursor position.
  - **MaxLength: (setMaxLength)** sets the maximum string length
  - **Text: (setText)** (public slot) sets the contents displayed on construction
  - **hasFrame: (setFrame)** draws the line edit within a two-pixel frame if enabled.
  - **isTextSelected: (selectAll)** (public slot) sets the text to be selected.

6.3.6 QScrollBar Properties

Inherits 6.3.1 (<cdx/QWidget/) and `QRangeControl`.

- **General:**
  - **MaxValue:** sets the maximum slider value; used in constructor (optional)

  - **MinValue:** sets the minimum slider value; used in constructor (optional)
  - **Orientation: (setOrientation)** sets the orientation of the scrollbar to horizontal or vertical.
  - **Value:** sets the initial value of the scrollbar in the constructor (optional)
  - **isTracking: (setTracking)** if enabled, the scrollbar emits `valueChanged()` whenever the bar is dragged; otherwise only on mouse release.

6.3.7 QSlider Properties

Inherits 6.3.1 (<cdx/QWidget/) and `QRangeControl`.

- **General:**
  - **MaxValue:** sets the maximum slider value; used in constructor (optional)
  - **MinValue:** sets the minimum slider value; used in constructor (optional)
  - **Orientation: (setOrientation)** sets the orientation of the slider to horizontal or vertical.
  - **Value: (setValue)** (public slot) uses `QRangeControl::setValue()` to set the value.
  - **isTracking:(setTracking)** if enabled, the slider emits `valueChanged()` whenever the slider is dragged; otherwise only on mouse release.

6.4 Properties of KDE supported Widgets

- 6.3.1 (<cdx/QWidget/)
  - 6.3.2 (<cdx/QButton/) (abstract)
    - 6.3.2 (QPushButton)
      - 6.4.1 (KColorButton)
      - 6.4.2 (KKeyButton)
  - 6.3.3 (<cdx/QComboBox/)
    - 6.4.3 (KCombo)
  - 6.3.4 (QFrame) (abstract for now)
    - 6.4.4 (KDatePicker)
    - 6.4.5 (KLedLamp)
    - 6.4.6 (KProgress)
    - 6.4.7 (KSeparator)
    - 6.3.4 (QTableView) (abstract)
      - 6.4.8 (KDateTable)
      - 6.4.9 (KTreeList)
  - 6.3.5 (QLineEdit)
    - 6.4.10 (KRestrictedLine)
- 6.4.11 (KLed)

6.4.1 KColorButton

Inherits 6.3.2 (QPushButton)

- General
  - **DisplayedColor (setColor())** the displayed color on the button

6.4.2 KKeyButton

6.4.3 KCombo

Inherits: 6.3.3 (<cdx/QComboBox/)

- General
  - **Entries** the string list of entries displayed in the combo box
  - **Text** the text displayed in the combo box currently
  - **isAutoResize** resizes the combo box to the current item

6.4.4 KDatePicker

- Appearance
  - **FontSize** the font size for the date picker

6.4.5 KLedLamp

6.4.6 KProgress

6.4.7 KSeparator

- General
  - **Orientation** sets the orientation of the separator to horizontal or vertical; default is horizontal

6.4.8 KDateTable

6.4.9 KTreeList

- Appearance
  - **TreeListBgColor**
  - **TreeListPalette**
  - **isBottomScrollbar**
  - **isScrollBar**
  - **isShowItemText**
  - **isSmoothScrolling**

  - **isTreeDrawing**

- General
  - **Entries**
  - **isAutoUpdate**

6.4.10 KRestrictedLine

6.4.11 KLed

Inherits 6.3.1 (<cdx/QWidget/)

- Appearance
  - **LedColor: (setColor())** sets the displayed LED color

6.5 Constructing a new Dialog

Constructing a new dialog is very easy if you already have experience with graphical construction applications. KDevelop offers to create a widget visually and displays the look as it will be shown to the user directly. Further, you can have a preview of your widget by selecting "Preview" from the "View" menu.

To begin constructing a dialog or any other widget, switch to the Dialogeditor and select "New Dialog" from the "File" menu. Then enter all needed information to the "New Dialog" dialog. Those are:

1. The Dialog inheritance. This is necessary because any widget is at least derived from `QWidget`. Besides the widget types provided by Qt, you can inherit e.g. form an abstract base class you constructed yourself within your project. In this case, select "custom" and enter the header file path to the line edit below.
2. The Dialog name. This sets the class name of the generated dialog. Select a classname that is descriptive for what the dialog does; in cases of inheritance from `QDialog`, you may enter a name that ends with `Dlg` to remember yourself it's a dialog. Naming convention should match that of KDE and Qt: Use uppercase letters for your classname. For e.g. a grid-size selection dialog, you would enter `GridSizeDlg`.
3. The generated filenames. Those are preset when entering the dialog name, but can be changed afterwards. If you want to use other filenames, the naming convention should also match that of KDE and Qt: the filenames are all lowercase and contain the classname to remember what class is kept where. The data file that has to be set will later contain the generated code that will build up your dialog. You should not edit this file manually afterwards; use the implementation file for any additions towards dialog construction code and method implementations.

The dialog will then show itself as a widget with a grid. As the dialogeditor uses the grid to snap any child widgets to the geometry, you can change the grid size with the "Grid Size" entry in the "View" menu, if the preset values don't match your needs.

Then select the "Widgets" tabulator on the left pane and press the button for the widget item you want to add to the main widget. It directly appears on the main widget's left upper corner and

gets selected by a resizable frame. Then move or resize the widget with the mouse. The cursor will change to indicate which action can be done at the current position.

After having finished the construction, select "Generate Files" from the "Build" menu or hit the according toolbar button. The files will then be generated at the preset location and included into your project sources. A rebuild or make will compile all generated files within your project and you can add the according constructor call to the application to invoke the dialog or widget. For KDE projects, all widget properties that will be visible later, e.g. label texts, are set with the `i18n()` macro of `kapp.h` to support internationalization. Therefore you should do a "Make Messages and merge" when finished with construction and implementation.

When creating a dialog or widget, you should watch the following guidelines:

- Always try to be consistent! This is probably the most important rule when constructing GUI elements. Mind that the user will only accept an application that is easy to understand no matter how complex it's purpose may be.

- Add help wherever you can by tool-tips, What's this..? help or Quick-help. This allows getting direct information about the purpose of the GUI elements.

- watch the **keyboard focus** ! The generator does not take care of that- this has to be watched when constructing any widget; otherwise you have to reorder your initialization code by hand which is a very unthankful job. The keyboard focus on any widget means the oder on which items get the keyboard input focus when the user presses the tab and shift+tab button. It would be very annoying if the focus changes everywhere but not to the next widget visible below or to the right of the current widget. Therefore start constructing your widget top down from left to right to ensure the consistency of the focus.

6.6 Setting Widget Properties

Widget properties can be set easily with the properties window entries. When a widget gets selected, the properties window automatically updates to the properties of the current widget. As all widgets are derived from `QWidget`, you can set the `QWidget` properties plus an amount of properties that are specific to the selected widget. Properties can be:

- Integer values, such as the geometry of a widget or the font size

- Boolean values to enable/disable certain parameters of the widget. Set with combos containing true and false

- enumerable values of a widget, e.g. the palette. Set with combos containing all possible values

- Color values for e.g. the displayed color. Set with the KDE Color Dialog

- Font values for e.g. labels. Be careful to set Font values other than the default because this may prevent KDE from updating the font. Set with the KDE Font Dialog

- File names for e.g. background pixmaps. Do not use gif images here as these may get unsupported by further Qt versions > 1.42

6.7 Integrating the Dialog

Whenever you created a widget, you probably want to add it to the project to execute the action it is designed for. As a widget can be constructed for several purposes, we will watch for two cases: a `QWidget` inherited widget and a `QDialog` one.

6.7.1 `QWidget` inherited

Let's say you created a widget that will be part of the main view. If it fills the whole view area, you have to add an instance pointer to the header declaration of your `KTMainWindow` instance replacing the currently set view widget. Then change the code in the `initView` method to set this widget the main view. Additionally, you could remove the View class of the generated project, but mind that the document instance and the App instance depends on the view class. In this case, it is technically a much better way to create a mini-KDE application and construct your `KTMainWindow` instance yourself.

More often the widget is intended to be a part of the view area, which means it is combined with other widgets. This can be done by using one of the following classes that provide a divider to separate two widgets:

1. `QSplitter`
2. `KPanner`
3. `KNewPanner`

If the main view shall contain more than two widgets, you have to use another instance of the divider as one of the two managed widgets by the first one. Then add the according widgets to each panner and set the first panner the view area.

6.7.2 `QDialog` inherited

If your widget inherits `QDialog`, it is probably intended to change one or more values; often this is used to set the application preferences. To invoke the dialog, you have to add a slot to the `App` class by adding the method declaration and the implementation body. Then add the constructor call to the method as well as a call to `show()` or `exec()` the dialog. Finally, you should take care for processing the results of the dialog; this can either be done by the dialog who changes values of the parent widget itself or by retrieving the values from the dialog (which would make your dialog a lot more reusable in other projects). Mind that you should call `delete` if you called the dialog instance with `new` to avoid memory leaks.

Finally, you have to connect a menuentry (with according statusbar help) to the new slot invoking the dialog; optionally a keyboard accelerator and a toolbar icon. For this, add a resource ID to the file `resource.h` with a define. Then add an according menuentry to one of the popup menus already present in the menubar or create a new popup to add your menuentry. The menuentry consists of:

- an optional icon pixmap. Call this with the `Icon("iconname.xpm")` macro of `KApplication` to use the provided `KIconLoader` instance.
- the menuentry name. Add this with the `i18n("&entryname")` macro of `KApplication` to allow internationalization. The ampersand should be in front of the letter that will be displayed underlined to access the entry directly by keyboard acceleration.

- the member instance to call. Normally this would be the `this` pointer.
- the member slot to call. Use `SLOT(yourmethod()` to call the slot on the signal `activated()`.
- the accelerator key. This should be set to zero as this is done by an entry in `initKeyAccel()` where you have to introduce an accelerator key together with the slot to call. Then call `changeMenuAccel()` to change the menu item's accelerator. This will make it configurable by a key-chooser dialog later. For standard actions, use the enumerable values given by `KAccel`.
- the menu ID as set in `resource.h`

Chapter 7

Printing Support

Printing is usually provided by your application to let the user create a printed version of the document he created with the application; therefore only needed for those programs that are used to produce something the user may want to print out, e.g. text or pictures. In any case, this requires an interface that is provided by the Qt library by two classes: the `QPrintDialog` class, offering the printing dialog, and the `QPainter` class that is also used to draw the widget's contents usually. As the view-class of an application is responsible for displaying a document, it also is responsible for printing.

7.1 The Qt Print Dialog

The Qt Printer dialog can be used including `qprintdialog.h`. When using the KDE framework application, this is already used by the view class, so you only have to complete the implementation of the method `print()` by using `QPainter`.

7.2 The QPainter Class

Independent of the printer's capabilities, you can use `QPainter` to draw your document onto the printer provided by `QPrinter` like you would when drawing onto a widget. The only difficulty would be where you have to implement the way things have to be printed.

Chapter 8

Help Functions

A very important part of the development process is to provide help functionality to the user wherever possible. Most developers tend to delay this, but you should remember that a normal user isn't necessarily a Unix-expert. He may come from the the dark side of computer software usage offering all sweets that a user may need to work himself into using an application even without ever touching the manuals. Therefore, the KDE and Qt library provide all means usually considered making an application professional in the eyes of the normal user by help functions that are ready to use. Within the application, those are:

- Tool-Tips
- Quick-Help
- Statusbar help
- What's this...? buttons

Additionally, the application should provide means to access a HTML-based online manual directly using the standard help key F1.

As KDevelop also offers all types of help as well as the KDE framework generated by the application wizard already contains support for this, this chapter will help you find out where and how to add your help functionality.

During the development of your application you should try to be consistent whatever you're doing; therefore you should do the necessary steps directly while extending the code. This will prevent you from diving into the code again and figuring out what your application does or what you intended by certain parts of the code.

8.1 Tool-Tips

A very easy mean of providing help are tool-tips. Those are small help messages popping up while the user moves the mouse over a widget that provides a tool-tip and disappears when the mouse moves away. The most popular usage of tool-tips is made in toolbars where your tool-tips should be kept as small as possible because toolbars can be configured to display their contents in various ways: either displaying the button, button with text on the right, button with text below, text only. This possibility should be made configurable by the user, but isn't a must-be. The text is shown as a tool-tip anyway and a toolbar usually consists of buttons and other widgets like lineedits and

combo boxes. For a complete reference, see the `KToolBar` class reference located in the KDE-UI library.

As an example, we have a look at the the "New File" button in a generic application:

```
toolBar()->insertButton(Icon("filenew.xpm"), ID_FILE_NEW, true, i18n("New File") );
```

There, the part `i18n("New File")` provides a tool-tip message. It is enclosed by the i18n() macro provided by `kapp.h` to translate the tool-tip towards the currently selected language.

Tool-tips can also be added to any custom widget by using the classes `QToolTip` and `QToolTipGroup` provided by Qt. An example of that would be:

```
QToolTip::add( yourwidget, i18n("your Tip") );
```

For more information, see the Qt-Online Reference, class `QToolTip`.

8.2 Adding Quick-help

Quick-Help windows are another good example of providing help. The user can access the quick-help over a widget that it is connected to by pressing the right mousebutton and selecting "Quick-Help" in the context menu. Therefore, Quick-Help can be placed somewhere in between a detailed handbook reference help and tool-tips- the documentation would be too extensive and a tool-tip would not provide enough information. To see how Quick-Help works, open any dialog within KDevelop and press the right mouse button over a dialog item. Then select the Quick-Help menuentry and you're offered the help message. Additionally, those messages can be formatted by color, font and even can be used for containing URL's to refer a certain webpage (and therefore can refer to the documentation handbook as well).

To make use of Quick-Help, add the include file `kquickhelp.h` to your sourcefile containing quick-help. As the `KQuickHelp` class is part of the KDE-UI library, it should already be used by your application; if not, set the linker flags of your project to use kdeui.

An example would be:

```
KQuickHelp::add( yourwidget, i18n("your Tip") );
```

which is almost the same as with QToolTip. When constructing a dialog with the KDevelop dialogeditor, add your tool-tips and Quickhelp in the implementation file- NOT within the data sourcefile as this is rebuild by the dialogeditor every time you edit the widget.

The `KQuickHelp` class provides also formatting text by using tags. It allows hyperlinks including Internet protocols, colors, font types and sizes. See the *KDE Library Reference Guide* and the class documentation for `KQuickTip` for more information.

8.3 Extending the Statusbar Help

As the frame applications provided by KDevelop contain a statusbar as well, it also offers a set of statusbar messages already for all menu and toolbar items. A statusbar help message is a short message that extends the meaning of a tool-tip or can be seen as a replacement for a tool-tip over menubar items and is (as the name suggests) displayed in the statusbar when the user enters a menu

and highlights the menu entry; therefore all menu items connect their signal `highlighted(int)` to the method `statusCallback(int)` which selects the according message in a switch statement. Whenever you add a menuitem to already existing menus or a toolbar item, add an according entry in this method with a short description of the action the user will cause when activating the button or menuentry.

Example:

```
case ID_FILE_NEW:
            slotStatusHelpMsg(i18n("Creates a new document"));
            break;
```

This will display a statusbar message by calling the method `slotStatusHelpMsg()` with the according translated help string whenever the user highlights a menu or toolbar item with the id ID_FILE_NEW that is connected to the `statusCallback()` method. Toolbars connect to this method by their signal `pressed(int)`, which allows the user to press the toolbar button and move away the mouse when he doesn't want to invoke the command. `KToolBar` also offers the signal `highlighted(int, bool)` which can be used to display the message whenever the user highlights the button instead of the preset signal used.

8.4 The "What's This...?" Button

The "What's This...?" button provides help windows like Quickhelp, but with the intention that the user wants to get help about a certain widget within the working view or a toolbar item. It is placed in the toolbar and gets activated once the user hits the button. The cursor changes to an arrow cursor with a question mark like the button itself looks like. The the user can press on a visible widget item and gets a help window. As an exercise, you could try this behavior with the What's this...? button within KDevelop. To add the What's This...? button, do the following:

1. include `qwhatsthis.h` into your sourcecode
2. add a private member `QWhatsThis` whats_this/ or with another member name to your `KTMainWindow` derived class declaration
3. define a resource id for your what's this button into the `resource.h` file,e.g. `#define ID_HELP_WHATS_THIS 10100`
4. in your method to create the toolbar (usually `initToolBar()`), add at the location you want to have the button displayed:

```
whats_this = new QWhatsThis;
QToolButton *btnwhat = whats_this->whatsThisButton(toolBar());
QToolTip::add(btnwhat, i18n("What's this...?"));
toolBar()->insertWidget(ID_HELP_WHATS_THIS, btnwhat->sizeHint().width(), btnwhat);
btnwhat->setFocusPolicy(QWidget::NoFocus);
```

5. finally, add the messages you want to have on a click over a certain widget like this:

```
whats_this->add(class_tree, i18n("Class Viewer\n\n"
                                 "The class viewer shows all classes, methods and variables "
                                 "of the current project files and allows switching to declarations
                                 "and implementations. The right button popup-menu allows more speci
                                 "functionality."));
```

Chapter 9

Extending the Documentation with SGML

Due to the fact that projects often lack a complete set of user documentation, all KDevelop projects contain a pre-build handbook that can be easily adapted; therefore fulfilling another goal of KDE: providing enough online-help to support users that are not familiar with an application. This chapter therefore introduces you on how to extend the provided documentation template and what you have to do to make it available to the user.

9.1 Why SGML ?

SGML (Standard Generalized Markup Language) is a text formatting system that allows to create output in various formats; therefore technical documentation has to be written only once and can then be transfered to the desired output. The most used output is probably HTML to provide online help through web-browsers in a time where Internet standards are available even on single-desktop systems. KDE makes use of HTML documentation by it's KDEHelp application where all KDE applications are listed and give access to their user manuals as well as by a helpmenu where the user can access the online-help directly from within the application.

To provide a unique look to KDE documentation, the KDE-Software Development Kit (SDK) contains a tool called `ksgml2html` which itself uses the sgml-tools' ksgml2html program to extend the HTML output with the KDE logo.

But besides this, sgml-tools allow output in plain text, GNU info, LyX, DVI, Postscript and RTF; therefore the SGML documentation can be easily used to provide a printed version of the online handbook. The KDevelop IDE itself contains four handbooks that are all written in SGML - you may read this in a printed version or online right now; this alone shows how useful it is to learn the few things about writing in SGML.

9.2 What the Documentation already contains

When creating a KDevelop project, the subdirectory docs/en already contains the index.sgml documentation file and the already produced output HTML files. Those are already included into the project as well as their installation destination is preset to the KDE HTML directory. The documentation is already adapted to your project name, version number and the programmer's information.

Further, the output covers the index.html file containing the table of contents (which is opened by KDE Help when the user requests help); an installation introduction and a copyright information.

Therefore, when extending the documentation, you only have to concentrate on the information you want to add. Mind that for KDE projects you have to run "Make Doc-Handbook" from the "Project" menu again after the project is created. The index.sgml file is again processed by `ksgml2html` and the logo is added. Then open the RFV and add the `logo.gif` file to the project and set the file properties correctly to install the logo file into the same location the HTML files will go.

9.3 Adding new Pages

Adding another page to the documentation is very easy by adding another <sect>, followed by the name of the chapter to the index.sgml file at the location where your chapter will appear. For that, search the <sect> that is the chapter before and add your section after that into the sgml file. Whenever you want to rebuild your documentation output to control for errors, select "Make Doc-Handbook" from the "Project" menu which will invoke either ksgml2html or ksgml2html depending on the project type. If errors occur, you are able to locate the error by clicking on the error message or selecting "next error" from the "View" menu.

For each <sect> you added, another HTML output file will be generated; therefore all chapters after the inserted will change their chapters. This is important if you call your HTML documentation pages from within your application- mind that you have to watch the page you call for help.

The sgmltools-package contains a reference for almost all valid tags and contains sample code for various formatting purposes. As the standard SGML file uses the <article> format, you should think about changing to <book> formatting if your documentation extends more than 10-20 pages in printed form. The KDevelop handbooks have been created this way; the only thing you have to watch out for is using <chapt> instead of <sect> for chapters. The following subsections then have to be declared with <sect>, <sect1> etc. as usual.

9.4 How to call Help in Dialogs

Calling help in dialogs is often done by adding a Help-button; then you add a slot that is called when the button gets pressed. Within the slot implementation, call

```
kapp->invokeHTMLHelp( QString aFilename, QString aTopic );
```

where `aFilename` is the the filename to be called within your HTML documentation directory of the application; e.g index-3.html. `aTopic` then is the topic that is to be called. The hash prefix is automatically added; just enter the chapter you want to have on this page, actually this would be a subsection's name.

Chapter 10

Class Documentation with KDoc

Another important part of the documentation is including a descriptive help for your class interfaces. This will allow you and other programmers to use your classes by reading the HTML class documentation that can be created with `KDoc`. KDevelop supports the use of `KDoc` completely by creating the KDE-library documentation, also your application frameworks are already documented. To work yourself into the provided code, it would be a good start to read the included documentation online. The following describes what to do to get the API documentation, where KDevelop helps you add it and what kind of special tags `KDoc` provides additionally.

10.1 How to use KDevelop's Documentation features

To create the API documentation after you generated a project, select "Make API-Doc" from the "Project" menu. This will process all header files and create the HTML output. Then you can access the documentation by selecting "API-Documentation" from the Help-menu or the according book symbol in the Documentation tree, folder "Current Project".

The documentation is already cross-referenced to the KDE and Qt online-class documentation, so you can follow the inheritance easily with the inheritance overview. This may help you getting started with the KDE and Qt documentation as well.

10.2 Adding Class and Member Documentation

As KDevelop provides all means to add code automatically, it also offers direct documentation. Whenever you're using the Class Generator by choosing "Project"->"New Class", add a descriptive help message to the documentation field. This will add the documentation to the class header.

When adding class member functions and attributes with the classtools, add the member documentation to the according documentation fields as well.

You may think that documentation is a part of the development process that isn't very necessary. But remember that the more your project grows and the more people take part on the development process, class documentation is the best help to save time. If developers have to guess by method names what exactly the method does, it is even more likely that the meaning is misunderstood and the method apparently doesn't do the job a developer guessed it would do. Therefore keep track of your documentation and rebuild it as often as possible.

Besides this, the documentation files are NOT included into the project, nor do they have any

internationalization support. Therefore all API documentation should be held in English to allow international development groups to work with your sources.

Whenever you may want to add documentation by hand into the header file, just add the documentation **above** the method or class in a C-comment style with the difference that the first line has to begin with a slash and a double asterisk.

Example:

```
/** enables menuentries/toolbar items
      */
void enableCommand(int id_);
```

10.3 Special Tags

NOTE: The following documentation of this chapter is taken from the KDoc documentation provided with KDoc by Sirtaj S. Kang *taj@.kde.org*), author of KDoc; Copyright (c) 1997

The documentation is a mixture of:

- Normal text. Paragraphs must be separated by at least one blank line.
- text of the form

```
<pre>
.....code fragments....
</pre>
```

- Various tags of the form:

```
@tagname [tag parameters]
```

The valid tags for each type of source code entity are:

- Classes

```
@short [one sentence of text]
  A short description of the class
@author [one sentence of text]
  Class author
@version [once sentence of text]
  Class version (I normally set this to the RCS/CVS tag "Id")
@see [one or more references to classes or methods]
  References to other related documentation.
```

- Methods

```
@see
  as above
@return [one sentence]
  A sentence describing the return value
@param [param name identifier] [param description]
  Describe a parameter. The param description can span multiple
  lines and will be terminated by a blank line, the end of the
  comment, or another param entry. For this reason, param entries
  should normally be the last part of the doc comment.
```

- Constants, Enums, Properties

```
@see
  as above
```

- ALSO `@ref` As a departure from the javadoc format, the metatag "@ref" has the same format as @see, but can appear anywhere in the documentation (all other tags must appear on a line by themselves).

Chapter 11

Internationalization

11.1 What is i18n ?

i18n is an internationalization system that is used to offer internationalized versions of an application or project. The difficulty with writing applications is that they only support the language they originally are composed with; visually this can be seen on labels, menu entries and the like. Goal of the internationalization is to provide applications and library functions in the language of the user; therefore enabling users that are not capable of the original language to make use of the provided functionality and feel more comfortable.

11.2 How KDE supports Internationalization

KDE, as one of the most modern desktop environments, has set one of its numerous goals to provide applications for users in their native languages, and simplifies the work for developers to provide their application in any of the supported language.

Technically, this is realized by the KDE File System Standard which contains localization support for languages in terms of documentation and by providing application internationalization through the use of the KDE-core library class `KLocale`. This class does all the translation, dependent on the preferred language set in the KDE Control Center.

The developer on the other hand only has to know two things to make his application able to use this feature:

1. include `kapp.h` into your sourcecode wherever a visible text appears in your application, e.g. in source files that contain QLabels.

2. wherever you set the visual string, embrace it with the `i18n()` macro provided by `kapp.h` to enable translation.

3. whenever you have to access a locale object, use the `klocale` macro provided by `kapp.h`

That is almost all you have to watch for while coding. Mind that you should not internationalize any configuration strings that are used by `KConfig`, because this is not necessary on one hand and doesn't work for reading in values on the other.

11.3 Adding a Language to your Project

KDevelop also takes part on making life easier for developers to include native language support to their applications. Whenever you create a new KDE project, a `po` directory is added to the main project directory. There, your <*application*>.pot file will be placed after the generation is complete. The .pot file already contains all strings that are set up with the i18n() macro, therefore you only have to write your code using the macro again. From time to time, you should do a "Project"-"Make messages and merge", which will automatically extract all macros again and rebuilds the potfile.

To add a language to your application, choose "Project"-"Add translation file", which opens the language selection dialog. Select the desired language and press OK. Then, the according <*lang*>.po file will be build in the `po` directory. Then start translating the po file by selecting it from the po directory in the Real File Viewer or from the LFV, folder "Translations". If you have KTranslator installed, it will be opened in the "Tools" window with KTranslator, otherwise as a text file in the header/resource window. KTranslator makes it very easy to translate strings by scanning the existing translations of your local KDE installation, so they can be used already.

For editing by hand, we'll have a look at an example:

```
#: kscribble.cpp:619
msgid "Opens an existing document"
msgstr ""
```

The above shows a string that was extracted from the file `kscribble.cpp` at line 619. `msgid` and `msgstr` are the tags which give the information for the translation; `msgstr` will contain the translated string. There, you have to watch escape sequences such as \n or \t, which have to be included into the translation string. A German translation would therefore look like this:

```
#: kscribble.cpp:619
msgid "Opens an existing document"
msgstr "Öffnet ein existierendes Dokument"
```

That would be all to watch for translation; after you're done, save the file. When make is run within the po directory, the message files will be processed and errors may occur if strings are not translated consistently, e.g. escape sequences are missing. Then edit the according message string again and make sure that make runs without errors.

Additionally, you should be very careful when translating ampersands within text strings. The letters after ampersands are used as keyboard accelerators in conjunction with the ALT key to access menubar or popup menu items to change the keyboard focus to the selected item more quickly. Now, if the same accelerator letter appears in the same keyboard focus area (which would be the main widget on one time, and a dialog at another), each widget after the first one cannot be accessed by the supposed keyboard accelerator. So even translators have a responsibility for the usage of the application under their language. There is also no guarantee that the original letter will occur in the translation, so translators have to choose very carefully and should test the application under their language after they installed the translation to ensure it runs without these malfunctions.

11.4 Translation Team Contacts

The KDE Team also provides numerous contacts to developers that are contributing to the KDE project as translators. Those are organized in language teams and coordinate the translation work.

For an actual list and information who to ask for translating your application, see <http://www.kde.org>.

The information below is taken from the KDE web site and contains the current contact addresses as of March 06, 1999. If you want to join a team please write directly to one of the team coordinators.

The translation of the KDE is organized by Juraj Bednar *mailto:bednar@rak.isternet.sk* and Matthias Elter *me@kde.org*

You can subscribe KDE internationalization mailing list kde-i18n-doc@kde.org by sending a mail to *kde-i18n-doc-request@kde.org* with the word "subscribe" in the subject line.

Before starting any translation work, please contact the according translation team leaders for coordination to avoid double work.

br Breton translation team:

team coordinators: Jañ-Mai DRAPIER *jdrapier@club-internet.fr* website: *http://perso.club-internet.fr/jdrapier*

ca Catalan translation team:

team coordinators: Sebastià Pla *sastia@redestb.es*

cs Czech translation team:

team coordinators: Miroslav Flídr *flidr@kky.zcu.cz*

da Danish translation team:

team coordinators: Erik Kjær Pedersen *erik@binghamton.edu*

de German translation team:

team coordinators: Thomas Diehl *th.diehl@gmx.net* website: <http://www.dtp-service.com/kde/de/> mailing list: send a mail with 'subscribe' in the subject line to: *kde-i18n-de-request@kde.org* Webforum for discussions and user feedback: <http://www.dtp-service.com/discus_d>

el Greek translation team:

team coordinators: Theodore J. Soldatos *theodore@eexi.gr*

eo Esperanto translation team:

team coordinators: Wolfram Diestel *diestel@rzaix340.rz-uni-leipzig.de*

es Spanish translation team:

team coordinators: Boris Wesslowski, Alonso Lara *Boris@Wesslowski.com* website: <http://members.xoom.com/keko5/> mailing list: send a mail with 'subscribe' in the subject line to *kde-es@kde.org*

et Estonian translation team:

team coordinators: Hasso C. Tepper *hasso@ewsound.estnet.ee*

fi Finnish translation team:

team coordinators: Kim Enkovaara *kim.enkovaara@iki.fi*

fr French translation team:

team coordinators: Francois-Xavier Duranceau *Francois-Xavier.Duranceau@loria.fr* website: <http://www.loria.fr/~durancea/kde/wip-apps.html> mailing list: send an empty mail to: *kde-traduc-fr-subscribe@egroups.com*

he Hebrew translation team:

team coordinators: Erez Nir *erez-n@actcom.co.il*

hr Croatian translation team:

team coordinators: Vladimir Vuksan *vuksan@veus.hr*

hu Hungarian translation team:

team coordinators: Marcell Lengyel *miketkf@yahoo.com* website: `<http://sophia.jpte.hu/~kde>`

is Icelandic translation team:

team coordinators: Logi Ragnarsson, *logir@imf.au.dk* Thorarinn R. Einarsson, *thori@mindspring.com* Bjarni R. Einarsson, *bre@netverjar.is* Hrafnkell Eiriksson, *hkelle@rhi.hi.is* Gudmundur Erlingsson, *gudmuner@lexis.hi.is* Richard Allen *ra@hp.is*

it Italian translation team:

team coordinators: Andrea Rizzi *rizzi@kde.org*

ko Korean translation team:

team coordinators: LinuxKorea Co. *kde@linuxkorea.co.kr*

mk Macedonian translation team:

team coordinators: Sasha Konecni *sasha@msi-uk.com*

nl Dutch translation team:

team coordinators: flidr@CyberGate.zcu.cz *flidr@CyberGate.zcu.cz*

no Norwegian translation team:

team coordinators: Hans Petter Bieker *zerium@webindex.no*

pl Polish translation team:

team coordinators: Piotr Roszatycki *dexter@fnet.pl*

pt Portuguese translation team:

team coordinators: Pedro Morais *pmmm@camoes.rnl.ist.utl.pt*

pt_BR Brazil Portuguese translation team:

team coordinators: Elvis Pfützenreuter *epx@netville.com.br*

ro Romanian translation team:

team coordinators: Paul Ionescu *ipaul@romsys.ro*

ru Russian translation team:

team coordinators: Denis Y. Pershin *dyp@inetlab.com*

sk Slovak translation team:

team coordinators: Juraj Bednar *bednar@isternet.sk* maillling list: send a mail with 'subscribe' in the subject line to: *sk-i18n@rak.isternet.sk*

sl Slovenian translation team:

team coordinators: blazzupancic@hotmail.com *blazzupancic@hotmail.com*

sv Swedish translation team:

team coordinators: Anders Widell *d95-awi@nada.kth.se*

tr Turkish translation team:

team coordinators: Gorkem Cetin *gorkem@linux.org.tr*

zh_GB2312 Simplified Chinese translation team:

team coordinators: Wang Jian *larkw@263.net*

zh_TW_Big5 Chinese BIG5 translation team:

team coordinators: Chou Yeh-Jyi *ycchou@ccca.nctu.edu.tw*

Chapter 12

Finding Errors

12.1 Debugging Macros provided by Qt

The Debugging Macros provided by the Qt library can be read on the debug.html page of your Qt Online Reference Documentation, acessable on the link "Debugging Techniques" at the Qt Documentation index page.

The most recently used macros are

- ASSERT(b)
- CHECK_PTR(p)

Thereby, b is a boolean expression. Gives out a debugging warning if b is false; p is a pointer which is checked and gives out a warning, if p is null.

Details can be found in the Qt Online Reference.

12.2 KDE Macros

NOTE: This chapter is a copy of Kalle Dalheimer's *kalle@kde.org* explanation document about the KDEBUG macros included with the KDE libs package as kdebug.html

Last modified: Sat Sep 13 11:56:01 CEST 1997

What is KDebug

KDebug is a system of macros and functions that makes using diagnostic messages in your code more efficient. You can give a message one out of four severity level and an area. You can choose at runtime where diagnostic messages should go and which of them should be printed at all. How to use KDebug in your code

The macro KDEBUG

Using KDebug is very simple. All you have to do is to #include <kdebug.h> at the beginning of every source file in which you want to use diagnostic messages and output the messages by calling the macro KDEBUG. This macro expects three parameters. The first is the severity level. Use one of the following constants:

- KDEBUG_INFO

- KDEBUG_WARN
- KDEBUG_ERROR
- KDEBUG_FATAL

The second parameter is the area. An area is a part of KDE that you define yourself. You can then at runtime choose from which areas diagnostic messages should be printed. Please see the file kdelibs/kdecore/kdebugareas.txt for a list of already allocated area ranges. Choose an area within the range allocated for your application. If your application is not yet in here and you have CVS access, you can allocate a range for your application here, otherwise just mail me. It is probably a good idea to define symbolic constants for the areas you want to use, but this is completely up to you. The third parameter, finally, is the text you want to output. KDebug automatically prepends the logical application name if you output to a file, to stderr or to syslog. A newline is always appended, you need not (and should not) use one yourself. If you need parameters, you can use one of the macros KDEBUG1, ..., KDEBUG9. These allow for one to nine additional arguments. The syntax is exactly the same as with printf, i.e. you have to include format specifiers in your message which get replaced by the additional parameters. An example:

```
KDEBUG3( <idx/KDEBUG_INFO/, kmail_composer, "Message no. %d to %s has %d bytes",
          message_no, aMessage.to(), aMessage.length() );
```

KASSERT

There are also the macros KASSERT, KASSERT1, ..., KASSERT9 which work just like their KDEBUG-counterparts, except that they have an additional bool as their first parameter. Only if this evaluates to false will the message be output. Note: You should not use neither KDEBUG nor KASSERT before the KApplication object is constructed. Note 2: KDebug provides no means for internationalization because it is meant strictly for developers only. If you want to inform the user about an erroneous condition (like "this file is not writable"), use KMsgBox.

Compiler switches

You do not need any special compiler switches in order to use KDebug. But when you ship your product (this mainly applies to people who create distributions like .rpm or .deb packages), you should compile with the switch -DNDEBUG. This will simply remove all the debugging code from your application and make it smaller and faster (e.g. it uses 256K less non-shareable memory).

How to manage diagnostic messages at runtime

You can press Ctrl-Shift-F12 in every KApplication at any time, and the "Debug Settings"-Dialog will appear. Here you can define separately for every severity level what should be done with the diagnostic messages of that level. The following settings are available:

- Output: In this Combobox, you can choose where the messages should be output. The choices are: "File", "Message Box", "Shell" (meaning stderr) and "syslog". Please do not direct fatal messages to syslog unless you are the system administrator yourself. The default is "Message Box".
- File: This is only meaningful when you have chosen "File" as the output and provides the name of that file (which is interpreted relatively to the current directory). The default is kdebug.dbg.
- Area: The areas which should only be output. Every message that is not mentioned here will simply not be output (unless this field remains empty which is the default and means that all messages should be output). Your can enter several areas separated by commas here,

and you can also use area ranges with the syntax start-end. Thus a valid entry could be: 117,214-289,356-359,221. Please do not use whitespace.

Apart from this, you can also tick the checkbox "Abort on fatal errors". In this case, if a diagnostic message with the severity level "KDEBUG_FATAL" is output, the application aborts with a SIGABRT after outputting the message. When you close the dialog with OK, your entries apply immediately and saved in your application's configuration file. Please note that these settings are specific for one singular application! When you press cancel, your entries are discarded and the old ones are restored.

Chapter 13

The KDE File System Standard

This chapter is a copy of the KDE-File System Standard as published on the KDE website at <http://www.kde.org>, written by Richard Moore *rich@kde.org*

KDE File System Standard

This file documents the directory structure that KDE and all KDE compliant applications should use. This is version 0.0.4 of the standard.

13.1 Introduction

The purpose of the KDE FSSTD is to ensure that all resources (icons, mimetypes etc.) needed for KDE applications are stored in a consistent directory structure. Following this structure allows applications to make use of tools such as the KIconLoader class and allows separation of the platform specific data needed by KDE from platform independent data (making installations on multiple architectures possible). In this document directory names have been suffixed with a '/' character. Where the word 'appname' appears in angle brackets <like this> it means that there should be an entry corresponding to every installed KDE application. The word 'lang' is used in the same way to indicate that there should be an entry for every supported language named according to the standard two letter language codes eg. 'fr' for French, 'de' for German etc.

13.2 Directory Layout

The KDE directory structure is as shown below, the top of the KDE installation tree is usually '/opt/kde' and can be found at run time by using the kdedir() method of KApplication (this replaces the KDEDIR environment variable the use of which is now deprecated). This document will refer to this directory as kdedir().

- kdedir()/
  - bin/
    - Application binaries
  - lib/
    - standard kde libraries (libkdecore etc.)
    - *<appname>*/

        - Application specific data that is platform dependent
- include/
    - standard kde header files
- parts/
- cgi-bin/
    - CGI programs for kdehelp
- share/
    - doc/
        - HTML/
        - default -> Link to kdedir()/share/doc/HTML/en
        - *<lang>*/
        - *<appname>*/
        - index.html
        - other application help files
    - config/
    - applnk/
        - System/
        - Utilities/
        - Applications/
        - Games/
        - kfind.kdelnk
        - khelp.kdelnk
        - khome.kdelnk
        - krefresh.kdelnk
    - mimelnk/
        - magic
        - text/
        - audio/
    - partlnk/
        - *<partname>*.kdelnk
    - icons/
        - Icons used in kdelnk files
        - *<appname>*.xpm
        - mini/
        - Mini Icons for kpanel
    - toolbar/

    - Standard toolbar pixmaps (eg. fileopen.xpm)
  * wallpapers/
    - Wallpapers used by kdisplay
  * apps/
    - *<appname>*/
    - toolbar/
    - Toolbar pixmaps
    - pics/
    - Other application pixmaps
    - application specific data (must be platform independent)
    - *<libname>*/
    - pics/
  * locale/
    - *<lang>*/
    - LC_MESSAGES/
    - *<appname>*.mo

13.3 What does this mean to application developers?

A standard KDE application will install files into several places in the above structure. The only required items are the application binary, the application kdelnk file, the application icon and the application help files - all others are optional. The most common things that are installed are:

Type of file	Location
Application binary (required)	kdedir()/bin/
Application kdelnk file (required)	kdedir()/share/applnk/
Application icon (required)	kdedir()/share/icons/<appname>.xpm
Application help files (required)	kdedir()/share/doc/default/HTML/<appname>/<index>.html
Application toolbar pixmaps	kdedir()/share/apps/<appname>/toolbar/
Application platform independent data	kdedir()/share/apps/<appname>/
Application platform specific data	kdedir()/lib/<appname>/

13.4 Application Documentation

I've suggested making putting at least a single page in

kdedir()/doc/default/HTML/<appname>/<appname>.html

a requirement for KDE compliance. The application is free to use the directory to store any help data it requires.

Applications that support more than one language would place the other languages inkdedir()/doc/<lang>/HTML/<appname>/<appname>.html with there being one 'lang' directory for each language code as usual. Arranging the files like this would allow links between the help files of two different applications that both support a given language.

I am not 100% happy with the solution I've suggested as it does not allow any way to fall back to the default language if a required translation is not available.

13.5 What does this mean to library developers?

- kdedir()/share/apps/<libname>/toolbar

Toolbar icons for library widgets.

- kdedir()/share/apps/<libname>/pics

Any other bitmaps for library widgets.

Chapter 14

File System Usage for KDevelop Projects

As the last chapter covered the KDE File System Standard, this chapter deals with what you have to do to use the file system. A KDE project uses the file system at least for installation routines; therefore we will discuss setting installation properties for your project files. Your application may make use of files that are installed afterwards, where it is important to know how to get the relative pathname by the standard. This enables your application to work wherever the KDE file system may be and prevents hard-coding any file information.

14.1 Accessing Files during Runtime

After the installation of your project by end-users, your application may require file information during runtime. During the development process, you will experience at least one error which is caused when running your application within the KDevelop IDE and requiring the application manual by "Help"-"Contents" or pressing the F1 key. This will result in a message box, saying that the index.html file could not be found- if you haven't installed your application on your local KDE file system. Your application asks KDEHelp to open your index page with detecting the installation directory first through `KApplication`'s methods to access the file system, therefore, we will have a look at what `KApplication` offers and make some example usage. Also other classes of `KDE-Core` make use of the KDE File System like `KIconLoader` and `KLocale`, which will be reviewed afterwards.

14.2 KApplication Methods

The KApplication class offers the following methods to access the KDE File System:

```
void invokeHTMLHelp ( QString aFilename, QString aTopic ) const
static const QString& kde_htmldir ()
static const QString& kde_appsdir ()
static const QString& kde_icondir ()
static const QString& kde_datadir ()
static const QString& kde_localedir ()
static const QString& kde_cgidir ()
static const QString& kde_sounddir ()
static const QString& kde_toolbardir ()
```

```
static const QString& kde_wallpaperdir ()
static const QString& kde_bindir ()
static const QString& kde_configdir ()
static const QString& kde_mimedir ()
static QString localkdedir ()
static QString localconfigdir ()
static QString findFile ( const char *file )
```

The methods are generally used with the `KApplication` object of your application, where `KApplication` offers the macro `kapp` to receive the pointer:

#define kapp KApplication::getKApplication()

Therefore, the methods are generally used like this:

```
QString sounddir=kapp->kde_sounddir();
```

This example stores the path of the KDE sounddirectory under a `QString`, where you would append e.g. the sound filename. Then you can process this information and play a sound file that is located there. You should always test for the existence of a file by using `QFileInfo`'s `exists()` method.

Within these methods,

```
void invokeHTMLHelp( QString aFilename, QString aTopic ) const [public]
```

takes a special position to invoke the KDE help. Generally, you should use it everywhere a user needs to access information, e.g. when he is presented a modal dialog. The F1 key will not work to invoke the help contents, also the user should be presented the according help page. To make a good use of it, add a "Help" button to your dialog and create a slot that is used to connect on signal `pressed()`. In this method, use `invokeHTMLHelp()` with the according page and subject; in case your application's documentation isn't written completely yet, leave this open to complete it after the documentation is in sync with the application.

The documentation of `KApplication` says:

Invoke the kdehelp HTML help viewer.

Parameters: aTopic This allows context-sensitive help. Its value will be appended to the filename, prefixed with a "#" (hash) character.

aFilename: The filename that is to be loaded. Its location is computed automatically according to the KFSSTND. If aFilename is empty, the logical appname with .html appended to it is used.

The methods of `KApplication` will retrieve the following path's.

`kde_htmldir()`	`kdedir()/share/doc/HTML`	`Returns the directory where KDE stores its HTML documentation`
`kde_appsdir()`	`kdedir()/share/applnk`	`Returns the directory where KDE applications store their .kdelnk file`
`kde_icondir()`	`kdedir()/share/icons`	`Returns the directory where KDE icons are stored`
`kde_datadir()`	`kdedir()/share/apps`	`Returns the directory where KDE applications store their specific data`
`kde_localedir()`	`kdedir()/share/locale`	`Returns the directory where locale-specific`

		information (like translated on-screen messages) are
kde_cgidir()	kdedir()/cgi-bin	Returns the directory where cgi scripts are stored
kde_sounddir()	kdedir()/share/sounds	Returns the directory where sound data are stored. This directory is for KDE specific sounds. Sound data of Applications should go into kde_datadir()
kde_toolbardir()	kdedir()/share/toolbar	Returns the directory where toolbar icons are stored
kde_wallpaperdir()	kdedir()/share/wallpapers	Returns the directory where KDE wallpaper files are
kde_bindir()	kdedir()/bin	Returns the directory where KDE application binaries
kde_configdir()	kdedir()/share/config	Returns the directory where config files are stored
kde_mimedir()	kdedir()/share/mimelnk	Returns the directory where mimetypes are stored
localkdedir()	$HOME/.kde	Get the local KDE base dir
localconfigdir()	$HOME/.kde/share/config	Get the local KDE config dir

To search for a specific file, use `findFile(const char *file)` which will search several path's of the KDE File System:

1. $KDEDIR,
2. $KDEPATH,
3. "[KDE Setup]:Path=" entry in a config file.

If the file is not found, the QString method isEmpty() will return True

14.3 KIconLoader Methods

QPixmap loadIcon (const QString &name, int w = 0, int h = 0)

QPixmap reloadIcon (const QString &name, int w = 0, int h = 0)

QPixmap loadMiniIcon (const QString &name , int w = 0, int h = 0)

QPixmap loadApplicationIcon (const QString &name, int w = 0, int h = 0)

QPixmap loadApplicationMiniIcon (const QString &name, int w = 0, int h = 0)

bool insertDirectory (int index, const QString &dir_name)

14.4 Setting File Installation Properties

As the above explained where KDE applications should place their files and how to access them at runtime, the following will explain how to set the file properties correctly to ensure the files get installed at the right place. The Makefiles support a set of macros to install your files into the KDE File System and which have to be used for setting the file installation properties.

To set the properties, open your project and select "Project"-"File Properties" which opens the File Properties dialog. The file properties are displayed if you select a filename currently included in the project. First of all, a file has a type property, which can be one of the following:

- **HEADER:** specifies a file as a header file
- **SOURCE:** specifies a file as a source file
- **SCRIPT:** specifies a file as a script file
- **DATA:** specifies a file as a data file that usually gets installed like pixmaps or HTML documentation files
- **PO:** specifies a file as a translation file
- **KDEV_DIALOG:** specifies a file as a dialog file to be interpreted by the dialog library

Further, a file is included in the project, if "Include in Distribution" is checked. This ensures that the file is included in the distribution tarball or package.

If a file has to be installed, you have to enable "Install". This will allow setting the Installation path for the selected file, where the filename is already inserted.

Now, as said above, the Makefile already is capable of a set of macros for the KDE File System Standard. These are used to set the installation path and ensure that the files actually will land in the KDE file system and not somewhere else. Macros that can be used, have to be embraced in round brackets and are marked with the dollar sign in front of the macro. When configure builds the Makefiles on the end-user's system, it will determine values for these macros that match the real directory name and will expand the Makefile.am macro towards the actual destination.

When looking a standard KDE application project, you will see on the file property of your `index.html` file that it already uses a macro to determine where it should go:

$(kde_htmldir)/en/kscribble/index.html

This says, that make should install the file index.html in the kde-html directory, subdirectory en for English, the application subdirectory and the filename. You could as well use another filename if you like to rename the file on the installation destination.

For the destination of your binary you currently have to edit the project's Makefile.am if your destination should be different form the "Applications" section of kpanel:

APPSDIR = $(kde_appsdir)/Applications

Possible values are (as the KDE-File System Standard says):

- Applications
- Games
- Graphics
- Internet
- Multimedia
- Settings
- System
- Utilities

Setting no directory will end your applnk directly in kpanel's root.

The following list contains the macros that can be used in the installation setup for files:

```
kde_htmldir       Where your docs should go to. (contains lang subdirs)
kde_appsdir       Where your application file (.kdelnk) should go to.
kde_icondir       Where your icon should go to.
kde_minidir       Where your mini icon should go to.
kde_datadir       Where you install application data. (Use a subdir)
kde_locale        Where translation files should go to.(contains lang subdirs)
kde_cgidir        Where cgi-bin executables should go to.
kde_confdir       Where config files should go to.
kde_mimedir       Where mimetypes should go to.
kde_toolbardir    Where general toolbar icons should go to.
kde_wallpaperdir  Where general wallpapers should go to.
```

Use these macros in conjunction with the according necessary subdirectories and the filename for setting the installation properties. By default, the currently created HTML documentation files, the kdelnk file, Icon, Miniicon and the translation files (also newly create ones) are already set up for their destination; therefore you don't have to make any changes for your default installation routine that has been set up by the application wizard of KDevelop.

14.5 Organizing Project Data

Another issue in creating projects often appears to the programmer if he has or wants to include additional data that have to be installed with the project. You already know where to install it, but what about organizing it in the source tree ?

A good advice here may be to collect all data in directories that more or less match the KDE File System Standard, e.g. your application needs additional toolbar icons. Creating these icons in the main project directory is potentially not a good idea as they will be difficult to locate in the real file viewer and a removal will result in much work for each icon. Therefore, create your icon with "File"-"New" and choose a subdirectory `toolbar`; if it doesn't exist, it can be easily created with the "select directory" dialog. Existing icons can be copied and included into the project with "Project"-"Add existing file(s)", where you have to choose the files and the destination. When selecting the destination directory, you can create the `toolbar` subdirectory first within the selection dialog. After being finished, press OK and the files will be copied as well as included in the project.

As an example, a toolbar icon should go to the following:

$(kde_datadir)/<*appname*>/toolbar/<*youricon*>.xpm

Pictures or additional icons that are not used as toolbar icons should go to a subdirectory *pics* instead of *toolbar*.

14.6 The `kdelnk` File

The <*appname*>.kdelnk file currently included in your project will install itself in KDE's kpanel structure. You should think it is already created and complete, therefore shouldn't require any further notification. Despite of KDevelop's advanced qualities to help you with creating, programming and designing applications, it cannot determine the exact purpose of your application- and that is the information you have to add to the kdelnk file. As this is a text file, select it from the RFV or the LFV; it will be opened in the Header/Resource window.

The sample kdelnk file would look like this:

```
# KDE Config File
[KDE Desktop Entry]
Type=Application
Exec=kscribble
Icon=kscribble.xpm
DocPath=kscribble/index.html
Comment=
Comment[de]=
Terminal=0
Name=kscribble
Name[de]=kscribble
```

This already contains the basic configuration for the application specific data such as the icon, binary name, application name etc. You see that the section Comment is still empty. There you have to insert the Quick-Tip that will be displayed when the mouse cursor moves over the kdelnk file icon on the desktop or in kpanel. If scribble would be a small drawing program, you would enter e.g.

```
Comment=A simple drawing program
```

Each comment line afterwards will contain the same description translated in the language the brackets symbolize. Ask translators to insert a good translation in their native language or include the kdelnk file when asking for translating the application's po file; the same applies to the name of the application set in the Name lines.

Note: for more information about the purpose of the .kdelnk file, especially its use for commandline processing, see *The KDE Library Reference Guide*

Chapter 15

Programming Issues

Chapter 16

References

The KDevelop Programming Handbook contains information that are taken from various sources on the Internet and by mails to various mailing lists, as:

KDoc documentation: Sirtaj S. Kang *taj@.kde.org*

KDE Internationalization: Matthias Elter *me@kde.org*

KDebug documentation: Kalle Dalheimer *kalle@kde.org*

The KDE File System Standard: Richard Moore *rich@kde.org*

KDE-Developer's mini-HOWTO: David Sweet <*dsweet@chaos.umd.edu*>

Chapter 17

Copyright

Appendix A

Additional Information

A.1 Example Makefile.am for a Shared Library

```
# Example Makefile.am for a shared library.  It makes a library
#  called "example" as libexample.so.2.1.2
# This Makefile.am was taken from the kdelibs distribution and modified
#  to serve as an example.
#
# David Sweet
#

INCLUDES=  $(all_includes)

lib_LTLIBRARIES  = libexample.la

# Note:  If you specify a:b:c as the version in the next line,
#  the library that is made has version (a-c).c.b.  In this
#  example, the version is 2.1.2.
libexample_la_LDFLAGS = -version-info 3:2:1 $(all_libraries)

include_HEADERS = header1.h header2.h\
                  header3.h

# Which headers shouldn't be installed when a   make install  is done?
noinst_HEADERS = version.h

libexample_la_SOURCES = code1.cpp code2.cpp
                        code3.cpp

#  USE_AUTOMOC is great.  This takes care of all of your  moc'ing
#   dependencies.
#  (You still need to include, for example, header1.moc in code1.cpp.)
libexample_la_METASOURCES = USE_AUTOMOC
```

Index