Xandros ASUS
Software Development Kit

xandros
MAKING LINUX WORK FOR YOU

Table of contents

1 Introduction

The Xandros ASUS Software Development Kit is software and documentation to develop applications for ASUS products that include the Xandros operating system. Specifically, it is for creating applications that run on the Launcher, which is more commonly known as Simple mode in the ASUS Eee PC and EP20 computers, for example. Use the software development kit to develop applications that run on these products.

Software development kit

The software development kit is a set of development applications and tools that provide an environment to develop applications for the Launcher in any language supported by ASUS products.

The development platform is the Xandros Desktop - Open Circulation Edition, where Xandros Desktop is a Linux operating system and suite of software, and the Open Circulation Edition is a free version of Xandros Desktop. So you obtain a copy of Xandros Desktop - Open Circulation Edition, version 4.5 specifically, and install it on a computer. Optionally, you can install it on a computer as a virtual machine (VM) using VMware, instead of on a separate computer. When you install Xandros Desktop - Open Circulation Edition, on a computer or as a virtual machine, you are installing the following items:

- Xandros Desktop - Open Circulation Edition version 4.5, which is an operating system and software suite that includes the software development kit. The software development kit includes:
  - Eclipse, to develop your application
  - Qt, to develop your application

- Qt 4 plug-in for Eclipse
- Debian packaging wizard developed by Xandros

You need to use Eclipse, Qt, and the code and procedures as installed with the Open Circulation Edition. If you have Eclipse and Qt already installed on a computer, you can use them, then transfer your files to the Xandros Desktop computer. Similarly, if you know how to package files in the Debian format, then you do not need the packaging wizard included. But you need the Open Circulation Edition to modify files that define where your applicaiton goes in the ASUS product.

The following chapters in this document outline how to create an application for ASUS products:

- Application Development — To create an application that works with ASUS products
- Packaging — To package your application for installation on the ASUS products
- Virtual Machine Use — To use VMware to install Xandros Desktop - Open Circulation Edition and/or to install the ASUS product (for example the Eee PC build) to test your application
- Miscellaneous — Outlines localization, maintaining your package, and provides links to obtain more information

Software required

This document assumes a basic knowledge of Linux operating systems and programming with Qt and C++.

The following software is required:

- Xandros Desktop - Open Circulation Edition version 4.5, to get and use the software development kit. This document is included with it.
- VMware, to install and run Xandros Desktop - Open Circulation Edition as a virtual machine and/or to install the ASUS product as a virtual machine to install and test your application. Use of VMware is optional. You can install Xandros Desktop - Open Circulation Edition on a computer and use your Eee PC to test your new application, in which case you do not need VMware.

2 Application development

This chapter outlines how to develop a new application for use in a Xandros-based ASUS product, create your icons, and set up the product to use the application.

Developing an application

To create an application to work on the Eee PC and/or EP20, you use development tools, such as Eclipse and Qt, and develop files, windows, buttons, and other items in a way that the Eee PC and/or EP20 can understand them. This section outlines user interface guidelines, development of a sample application, and provides the code for common window development.

User interface guidelines

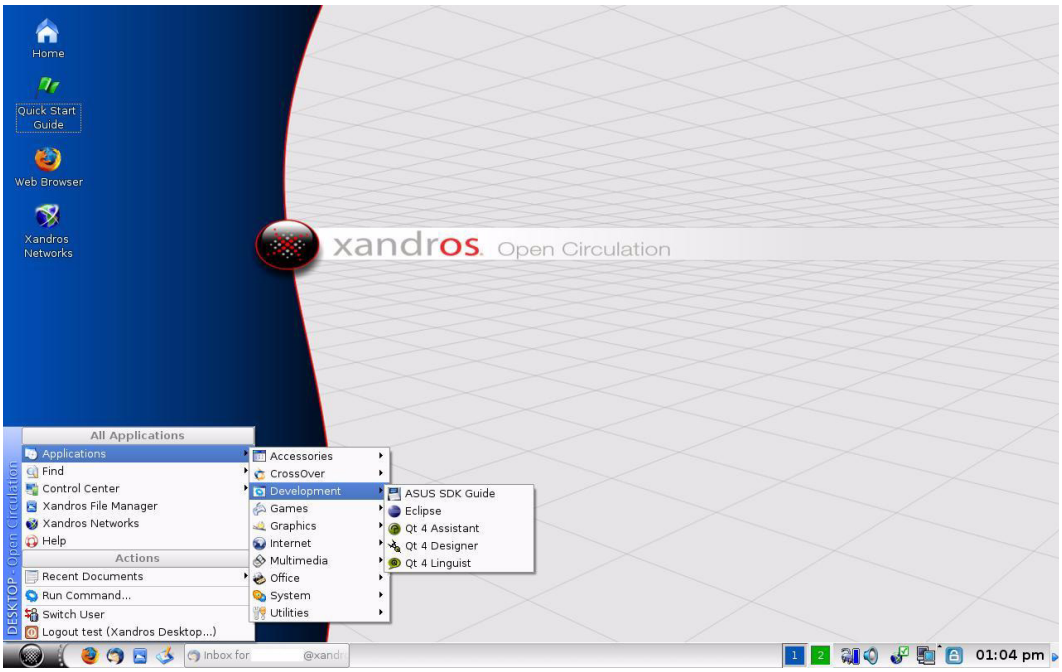
Here are some guidelines to help your application remain consistent with ASUS products:

- Use the Qt Designer application as outlined in the example in this section to develop dialog windows for your application, except for print, open-file, save, color, font, and message windows, for which you use common dialogs, rather than Qt dialogs. The code is provided later in this section.
- Ensure that the main application window and all message windows fit within the screen size. The Eee PC screen is small, for example, so you need to ensure that your windows fit.

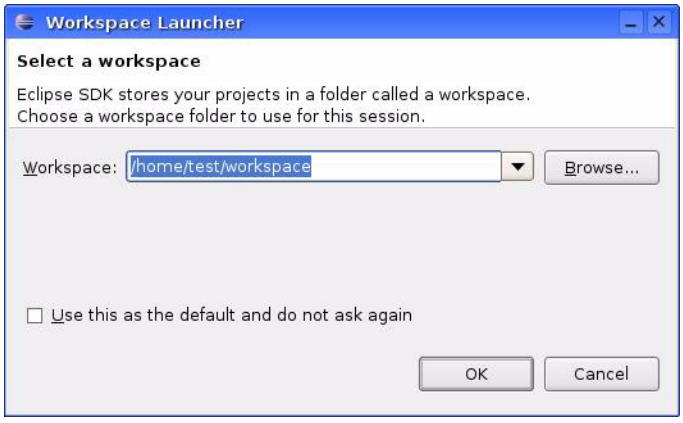
Example

We are going to use a sample address book application to demonstrate how you can create Qt applications using Eclipse.

First, in Xandros Desktop - Open Circulation Edition, start Eclipse by clicking **Launch > Applications > Development > Eclipse**.

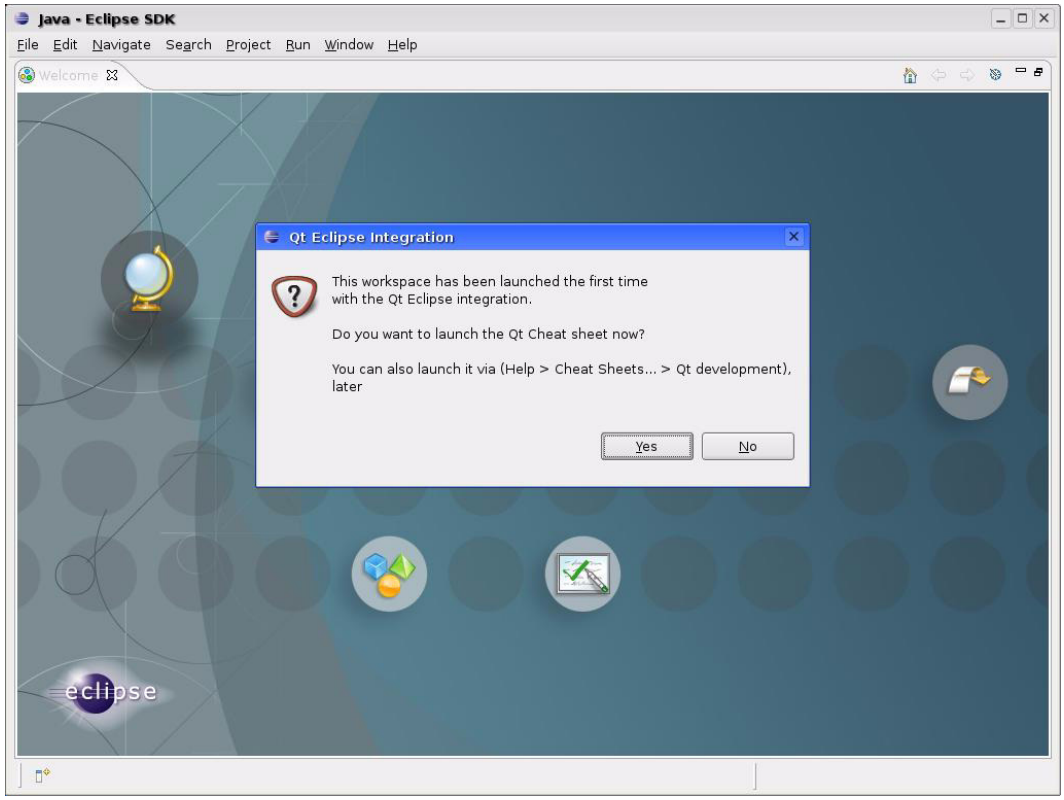


Eclipse prompts you to identify where you want to store your projects. Use the proposed workspace folder, which by default creates a folder called “workspace” in your home folder.

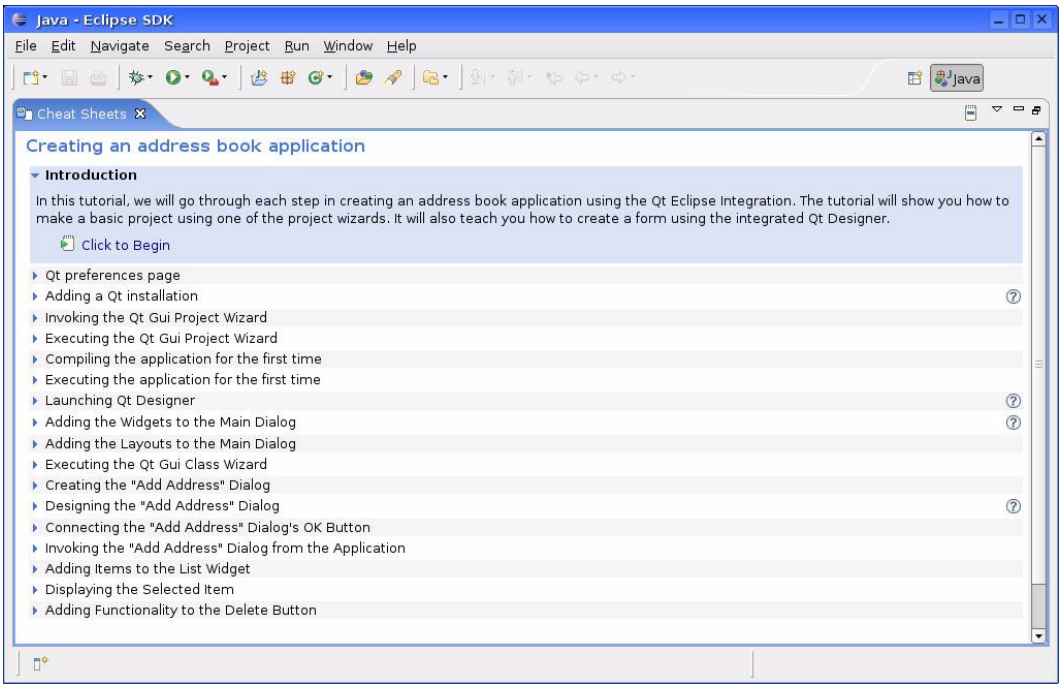


A Qt Eclipse Integration window prompts if you want the Qt Cheat Sheet to be displayed. Click **Yes**. This cheat sheet will guide you through the creation of the sample address book project. If you already had Eclipse open and do not see the

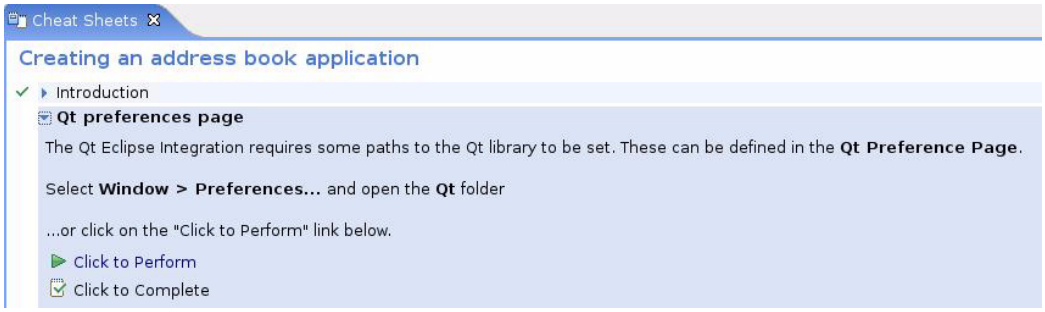
window prompt, get the cheat sheets by clicking **Help > Cheat Sheets**, then **Qt Development**.



Click on **Click to Begin** (shown). You are taken to the next step in the cheat sheet, which asks you to select the **Window > Preferences** menu item and then open the Qt folder.

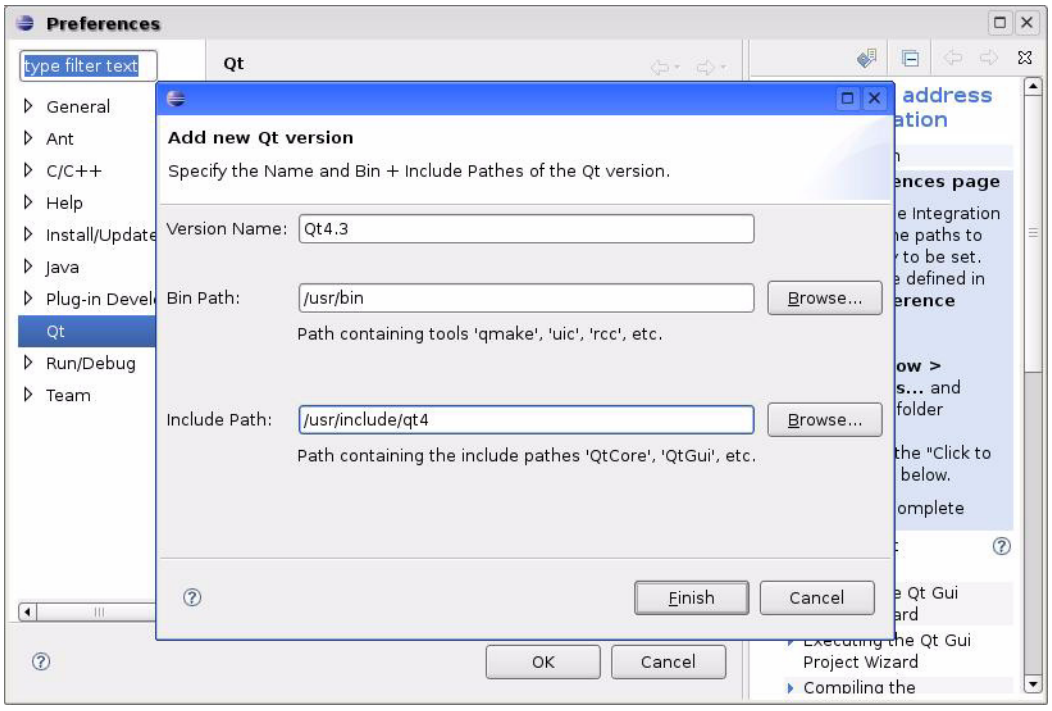


At this point, you have two choices: follow the instructions shown to manually open **Window > Preferences**, or you can click **Click to Perform** and let Eclipse do the work for you. For simplicity, we recommend that you click the **Click to Perform** link. By using this method, the cheat sheet is attached to the Preferences dialog. If you decide to manually open the window, you will have to close the Preferences dialog in order to be able to click the **Click to Complete** link, which is not convenient.



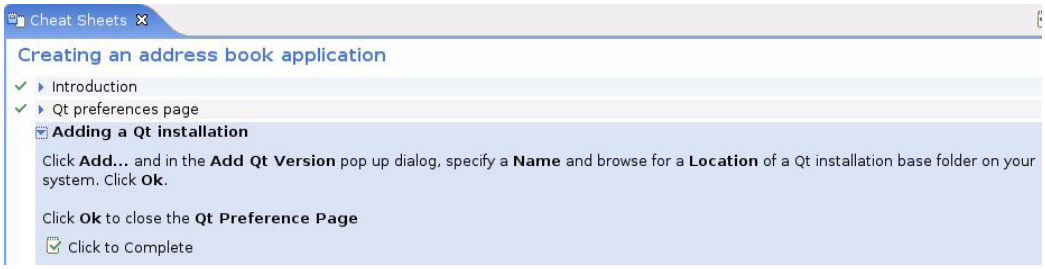
Next, you identify for Eclipse the location of your Qt installation. Click the **Add** button. A window appears with three fields. Enter the following information in the fields:

- **Version Name** — Type “Qt4.3”
- **Bin Path** — Type “/usr/bin”
- **Include Path** — Type “/usr/include/qt4”



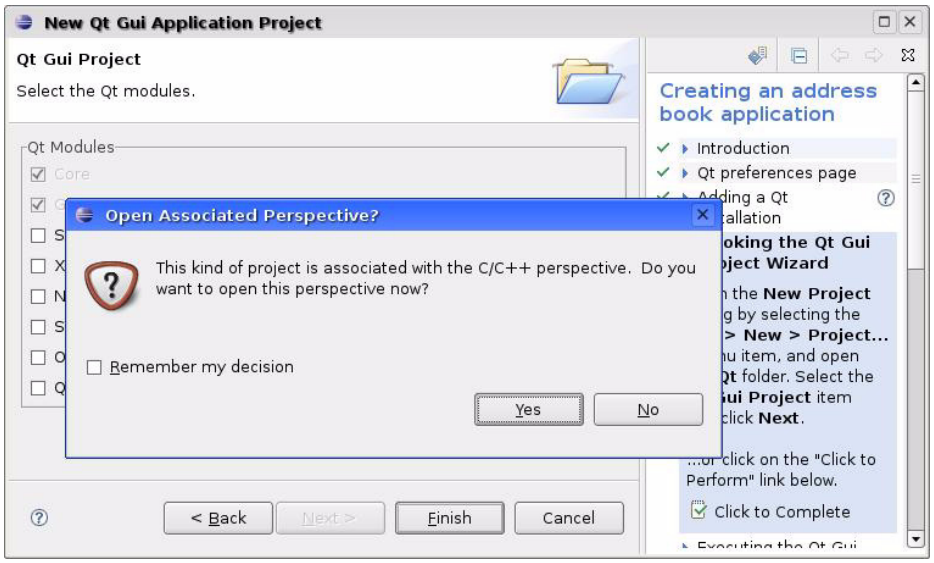
Click **Finish** and then **OK** to close the Preferences window and return to the Eclipse window. Click **Click to Complete.** A check mark appears beside the step.

You see that there are more than a dozen steps remaining. Start working through the steps in the cheat sheet. For example, click **Adding a Qt installation**, then **Click to Complete.**

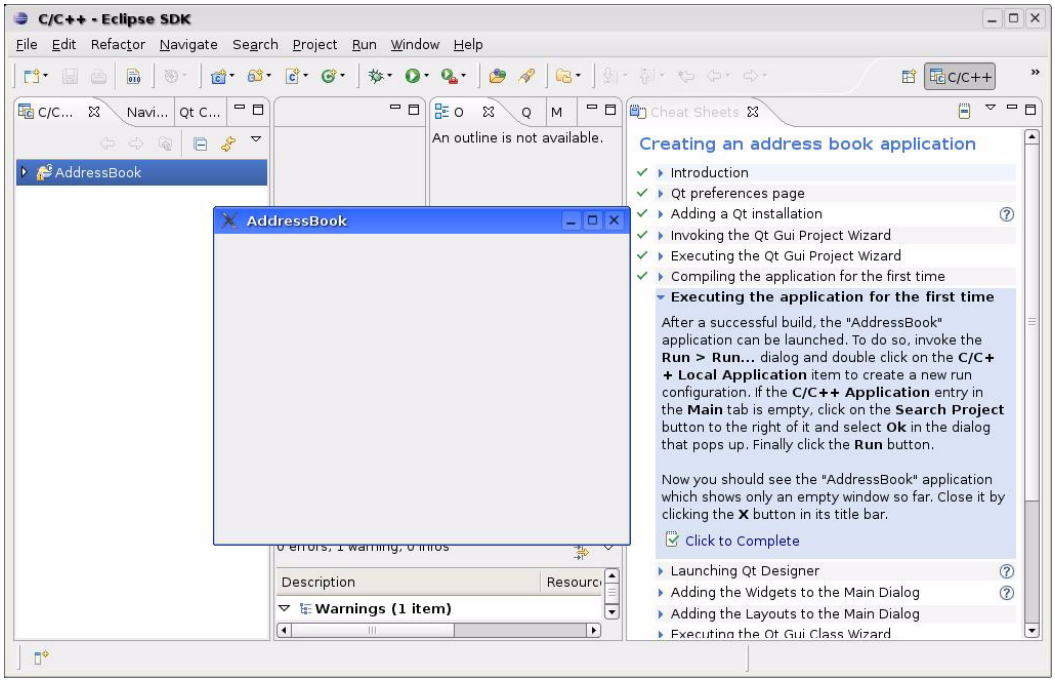


For the project name, call it AddressBook.

In the next step, you are asked if you want to open the C/C++ perspective. Click **Yes**. You will see that the main window now contains a lot more views. In Eclipse, a perspective determines the visible actions and views within the main window. Also, each perspective provides a set of functionality normally associated with a given task. For example, the debugging perspective contains a view that lets you see variables content at run time, which is not shown when you are in the C/C++ perspective.



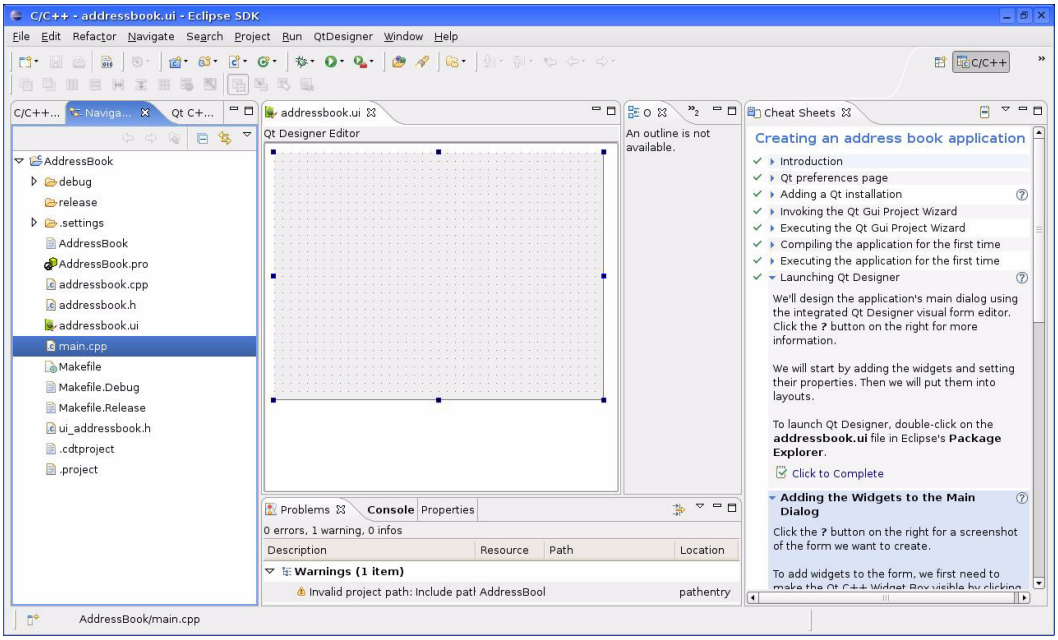
Keep following the cheat sheet until you execute/launch the application. At this point, you are able to run the Address Book application and its main window shows as being empty.



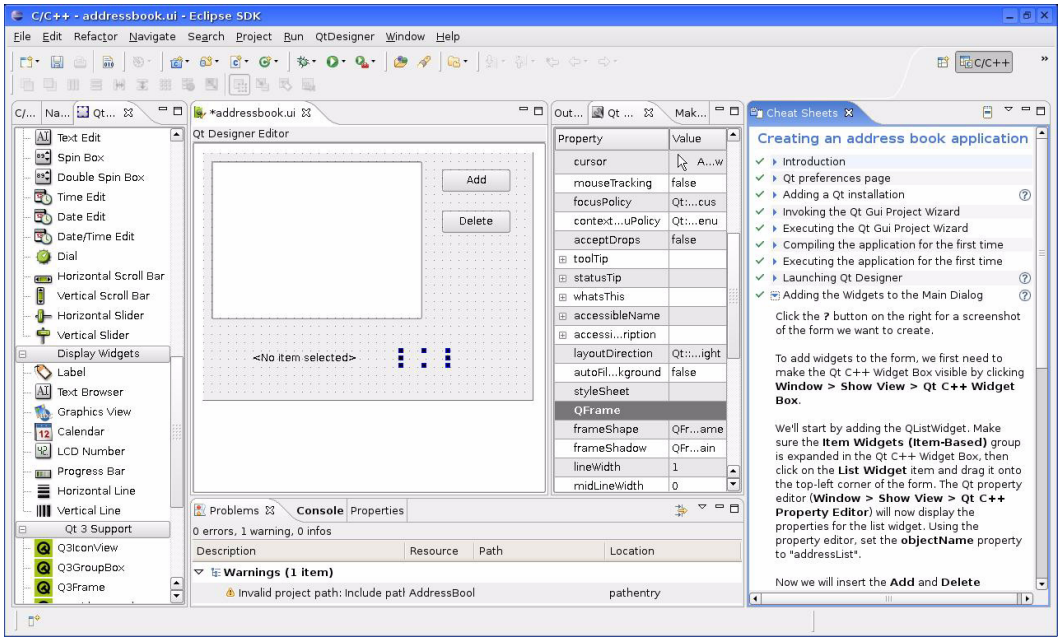
Adding widgets

The next few steps of the cheat sheet guide you through the process of launching the embedded Qt Designer and adding widgets to the main window.

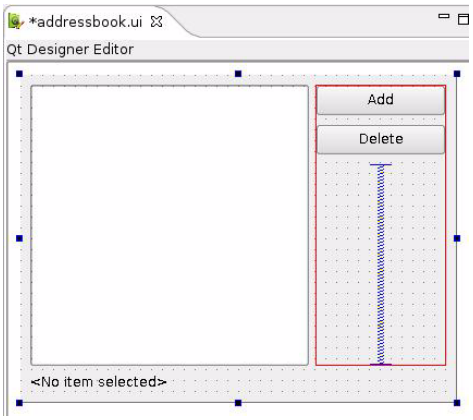
For the **Launching Qt Designer** step, you click the **Navigator** tab, expand the **AddressBook** entry, and double-click the **addressbook.ui** entry to launch Qt Designer within Eclipse (shown).



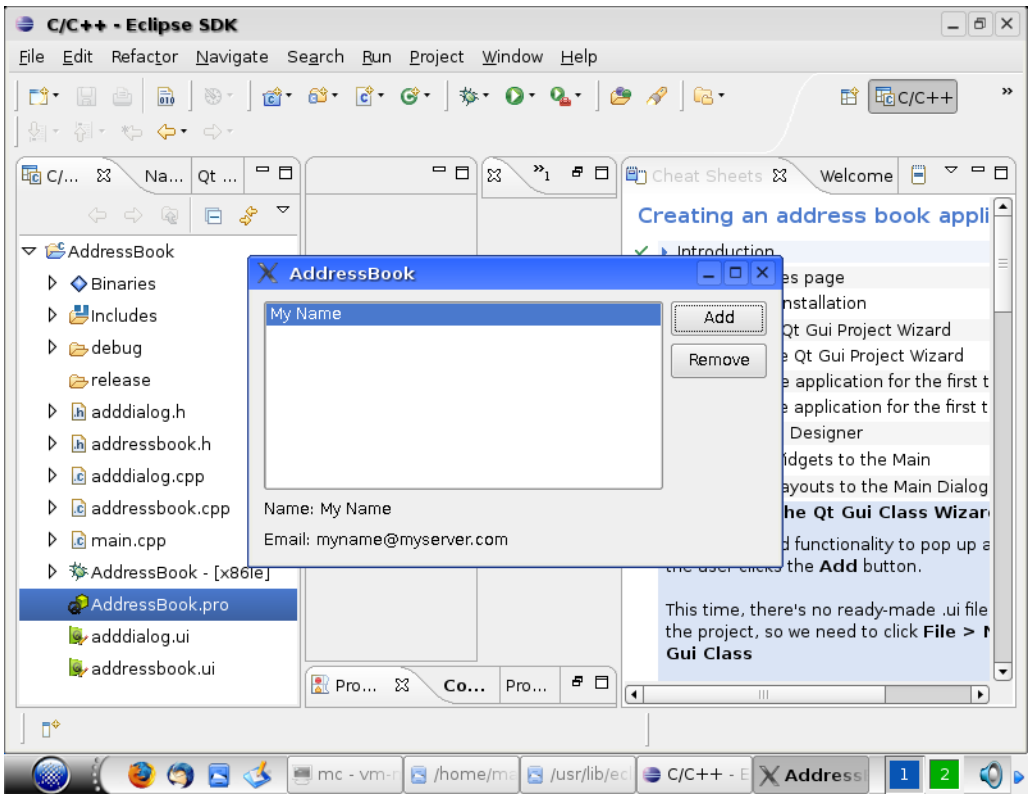
Follow the step to create the widgets. Here is what the window looks like after adding the widgets.



The next step involves adding layouts, which changes the way your application window looks.



After adding the code, the sample application is complete.



The sample project code can be found in /usr/lib/eclipse/plugins/com.trolltech.qtcpp.example/AddressBook

That completes the example. You need to set up your application using Eclipse and Qt Designer and for common dialogs, such as the file open window, you use the code in the next section instead of Qt Designer. When you are finished, open your .pro file in a text editor and add the following lines:

```
target.path = /usr/bin
INSTALLS += target
```

If Eclipse is slow when you are typing, turn off auto-completion as follows. Click **Window > Preferences**, expand the **C/C++** entry, expand **Editor**, then click **Content Assist**. Disable the three check boxes in the **Auto activation** area and click **OK** to apply and exit the window.

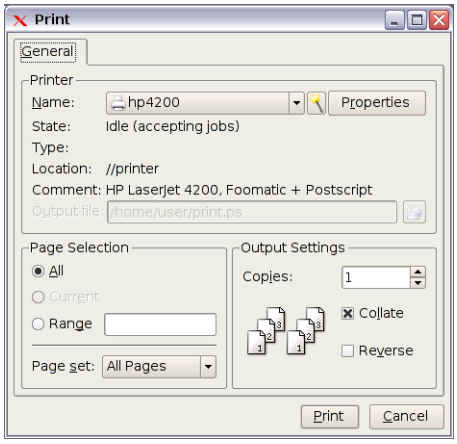
If you click **C/C++ Indexer** at the bottom-right of the window and it takes too much time to create the table of contents of functions, disable it as follows. Click **Window > Preferences**, expand the **C/C++** entry, click **Indexer**, and select the **No Indexer** option from the drop-down list. Click **OK** to apply and exit the window.

Creating common windows

Common print, file-open, save, color, font, and message dialogs are used. For example, if a user can print from your application, they click **File > Print** to launch a common print window that is used for several applications. You create the common print window using the code provided here instead of creating such windows in Qt Designer. Then you integrate the code with your project.

Common print window

For print windows, your application can use the stand-alone printer utility found in Xandros products (kprinter KDE utility) to support a common printer user interface. This utility takes care of the graphical user interface and supports many features and printer options. Printing of saved files or data from standard input (when running with –stdin option) is supported.



Here is an example of how to use the kprinter application:

```
void testApp::print()
{
    // create process
    QProcess *proc = new QProcess(this);

       // create argument list               QStringList args;
    args << "--stdin";

       // start kprinter process with write channel open
```

```
        proc->start("/usr/bin/kprinter", args, QProcess::WriteOnly);

        // write data
    proc->write("Text to print");
        ....
        proc->closeWriteChannel();
}
```

Common file windows

For common file windows, the base of Launcher is a KDE3 environment and it is not possible for an application written on Qt 3 or Qt 4 to use common KDE dialogs just by linking to existing KDE libraries. In order to fix the problem, a library is provided with the software development kit. There are two versions of the library:

- libqtkde for Qt 4 applications (Qt 4 is included with the software development kit)
- libqt3kde for Qt 3 applications

Use of the library is similar to qt3/qt4 common file invoke methods.

To enable qtkde integration in your Qt 4 project under the Eclipse environment do the following steps:

1. Open the .pro project file using text editor mode; right-click the project file and select **Open With > Text Editor**.
2. Add the following lines to the text:

   ```
   INCLUDEPATH += /usr/include/xandros
   LIBS += -lqtkde
   ```

3. Select **Project > Clean**.
4. Select **Project > Build Project**.

Your project is now ready to use the libqtkde library.

C++ header file “libqtkde.h” declares the libqtkde QKDEIntegration and its methods used to implement common dialog functionality:

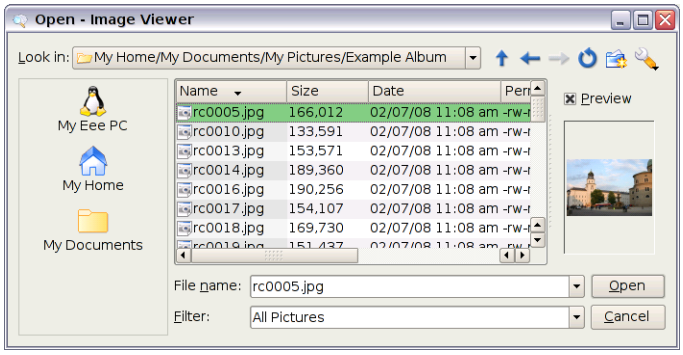
```
#ifndef QKDEINTEGRATION_H
#define QKDEINTEGRATION_H

#include <qstringlist.h>

class QWidget;
class QColor;
class QFont;

class QKDEIntegration
{
public:

static bool enabled();

static QStringList getOpenFileNames( const QString& filter, QString*
workingDirectory, QWidget* parent, const QString& caption, QString*
selectedFilter, bool multiple );
```

```
static QString getSaveFileName( const QString& initialSelection, const
QString& filter, QString* workingDirectory, QWidget* parent, const
QString& caption, QString* selectedFilter );

static QString getExistingDirectory( const QString& initialDirectory,
QWidget* parent, const QString& caption );

static QColor getColor( const QColor& color, QWidget* parent);

static QFont getFont( bool* ok, const QFont* def, QWidget* parent);

static int messageBox1( int type, QWidget* parent, const QString& caption,
const QString& text, int button0, int button1, int button2 );

static int information( QWidget* parent, const QString& caption, const
QString& text, int button0, int button1, int button2 );

static int question( QWidget* parent, const QString& caption, const
QString& text, int button0, int button1, int button2 );

static int warning( QWidget* parent, const QString& caption, const
QString& text, int button0, int button1, int button2 );

static int critical( QWidget* parent, const QString& caption, const
QString& text,              int button0, int button1, int button2 );

static int messageBox2( int type, QWidget* parent, const QString& caption,
const QString& text, const QString& button0Text, const QString&
button1Text,           const QString& button2Text, int defaultButton, int
escapeButton );

static int information( QWidget* parent, const QString& caption, const
QString& text, const QString& button0Text, const QString& button1Text,
const QString& button2Text, int defaultButton, int escapeButton );

static int question( QWidget* parent, const QString& caption, const
QString& text,            const QString& button0Text, const QString&
button1Text, const QString& button2Text, int defaultButton, int
escapeButton );

static int warning( QWidget* parent, const QString& caption, const
QString& text,            const QString& button0Text, const QString&
button1Text, const QString& button2Text, int defaultButton, int
escapeButton );

static int critical( QWidget* parent, const QString& caption, const
QString& text,            const QString& button0Text, const QString&
button1Text, const QString& button2Text, int defaultButton, int
escapeButton );

protected:
...
};

#endif
```

Each of these strings is outlined as follows. Some of them include examples.

QStringList getOpenFileNames(const QString& filter, QString* workingDirectory, QWidget* parent, const QString& caption, QString* selectedFilter, bool multiple)

This static method displays a file open dialog/window and returns a list of selected file names or empty string if none where chosen.



Parameter	Description
filter	A list of filters in QfileDialog format
workingDirectory	A variable to set and return current directory
parent	Parent widget
caption	Dialog's caption to use
selectedFilter	A variable to return selected filter value
multiple	Sets multiple file selection mode

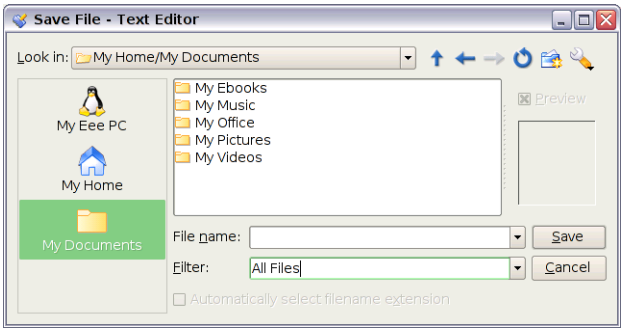
Example

The following example uses getOpenFileNames method to let the user select a file to open.

```
QWidget *pParent = this;
QString strWorkingDir("/home");
QString strFilter;
QStringList strs = QKDEIntegration::getOpenFileNames (
                    "Images (*.png *.xpm *.jpg)",
                    &strWorkingDir,
                    pParent,
                    tr("Choose a file"),
                    &strFilter,
                    false);
if (!strs.isEmpty()) // file selected
{
}
```

QString getSaveFileName(const QString& initialSelection, const QString& filter, QString* workingDirectory, QWidget* parent, const QString& caption, QString* selectedFilter);

This static method displays a save file dialog and returns a selected file name or empty string if none was chosen.



Parameter	Description
initialSelection	Initial filename to use
filter	A list of filters in QfileDialog format
workingDirectory	A variable to set and return current directory
parent	Parent widget
caption	Dialog's caption to use
selectedFilter	A variable to return selected filter value

Example

The following example uses the getSaveFileName method to let the user select a file to save.

```
QWidget *pParent = this;
Qstring strWorkingDir("/home");
QString str = QKDEIntegration::getSaveFileName(
                    "sample.jpg"
                    "Images (*.png *.xpm *.jpg)",
                    &strWorkingDir,
                    pParent,
                    tr("Choose a file"),
                    &strFilter;
if (!str.isEmpty()) // file selected
{
}
```

QString getExistingDirectory(const QString& initialDirectory, QWidget* parent, const QString& caption);

This static method displays a select folder dialog and returns a selected folder name or empty string if none was chosen.

Parameter	Description
initialDirectory	Initial name to use
parent	Parent widget
caption	Dialog's caption to use

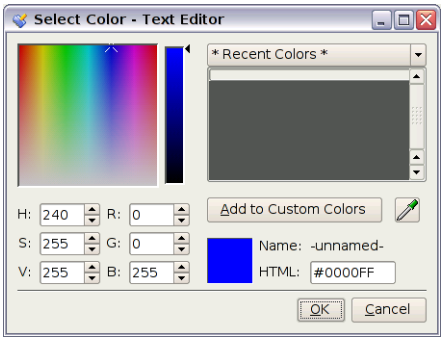
Example

The following example uses the getExistingDirectory method to let the user select an existing folder.

```
QWidget *pParent = this;
QString str = QKDEIntegration::getExistingDirectory(
                    "/home/user"
                    pParent,
                    tr("Choose a folder"));
if (!str.isEmpty()) // folder selected
{
}
```

QColor getColor(const QColor& color, QWidget* parent);

This static method displays a color chooser dialog and returns a selected color.



Parameter	Description
color	Initial color to select
parent	Parent widget

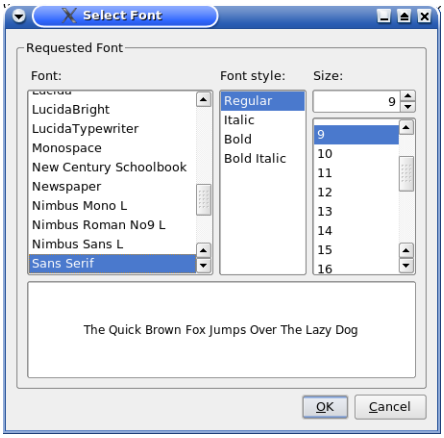
Example

The following example uses the getColor method to let the user select a color using KColorDialog.

```
QWidget *pParent = this;
Qcolor col(123, 0, 0); // red color
QColor newcol = QKDEIntegration::getColor(
                    col,
                    pParent);
if (newcol::isValid()) // new color selected
{
}
```

QFont getFont(bool* ok, const QFont* def, QWidget* parent);

This static method displays a font chooser dialog and returns a selected font.



Parameter	Description
ok	A variable to receive true if user clicks the OK button
def	Initial font to select
parent	Parent widget

Example

The following example uses the getFont method to let the user select a font using KFontDialog.

```
static QFont getFont( bool* ok, const QFont* def, QWidget* parent);

QWidget *pParent = this;
bool bOK;
QFont fnt = QKDEIntegration::getFont(
                    &bOK,
                    pParent->font(); // initial font setting
                    pParent);
if (bOK) // font selected
```

```
{
}
```

int messageBox1(int type, QWidget* parent, const QString& caption, const QString& text, int button0, int button1, int button2);

This static method displays a message box dialog. It returns the identity value of the button clicked.

Parameter	Description

type	Dialog type as one of the following values:
	`QMessageBox::Question`
	`QMessageBox::Information`
	`QMessageBox::Warning`
	`QMessageBox::Critical`

parent	Parent widget
caption	Dialog's caption to use
text	Text displayed by message box

button0, button1, button2	Buttons defined by one of the following identity values:
	`QMessageBox::NoButton`
	`QMessageBox::Ok`
	`QMessageBox::Cancel`
	`QMessageBox::Yes`
	`QMessageBox::No`
	`QMessageBox::Abort`
	`QMessageBox::Retry`
	`QMessageBox::Ignore`
	`QmessageBox::YesAll`
	`QMessageBox::NoAll`

The following example uses the messageBox1 method to ask the user a question and to receive a reply result.

```
QWidget *pParent = this;
if (QKDEIntegration ::messageBox1(
                  QMessageBox::Question,
                  pParent,
                  tr("Do you want to quit?"),
                  QMessageBox::Yes,
                  QMessageBox::No) ==  QMessageBox::Yes)
{
 // quit
}
```

int information(QWidget* parent, const QString& caption, const QString& text, int button0, int button1, int button2);

This static method displays an information message box dialog. It returns the identity value of the button clicked.

Parameters same as previous section.

int question(QWidget* parent, const QString& caption, const QString& text, int button0, int button1, int button2);

This static method displays a question message box dialog. It returns the identity value of the button clicked.

Parameters same as previous section.

static int warning(QWidget* parent, const QString& caption, const QString& text, int button0, int button1, int button2);

This static method displays a warning message box dialog. It returns the identity value of the button clicked.

Parameters same as previous section.

static int critical(QWidget* parent, const QString& caption, const QString& text, int button0, int button1, int button2);

This static method displays a critical message box dialog. It returns the identity value of the button clicked.

Parameters same as previous section.

static int messageBox2(int type, QWidget* parent, const QString& caption, const QString& text, const QString& button0Text, const QString& button1Text, const QString& button2Text, int default-Button, int escapeButton);

This static method displays a message box dialog. It returns the number of the button clicked (0, 1, 2).

Parameter	Description

type	Dialog type as one of the following values:
	`QMessageBox::Question`
	`QMessageBox::Information`
	`QMessageBox::Warning`
	`QMessageBox::Critical`

parent	Parent widget
caption	Dialog's caption to use

Parameter	Description
text	Text displayed by message box
button0Text, button1Text, button2Text	Button text
defaultButton	An index (0, 1, 2) of default button
escapeButton	An index of escape button

static int information(QWidget* parent, const QString& caption, const QString& text, const QString& button0Text, const QString& button1Text, const QString& button2Text, int defaultButton, int escapeButton);

This static method displays a message box dialog. It returns the number of the button clicked (0, 1, 2).

Parameters same a previous section.

static int question(QWidget* parent, const QString& caption, const QString& text, const QString& button0Text, const QString& button1Text, const QString& button2Text, int defaultButton, int escapeButton);

This static method displays a message box dialog. It returns the number of the button clicked (0, 1, 2).

Parameters same as previous section.

static int warning(QWidget* parent, const QString& caption, const QString& text, const QString& button0Text, const QString& button1Text, const QString& button2Text, int defaultButton, int escapeButton);

This static method displays a warning message box dialog. It returns the number of the button clicked (0, 1, 2).

Parameters same as previous section.

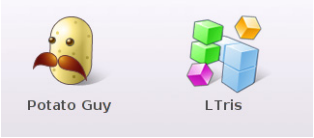
static int critical(QWidget* parent, const QString& caption, const QString& text, const QString& button0Text, const QString& button1Text, const QString& button2Text, int defaultButton, int escapeButton);

This static method displays a message box dialog. It returns the number of the button clicked (0, 1, 2).

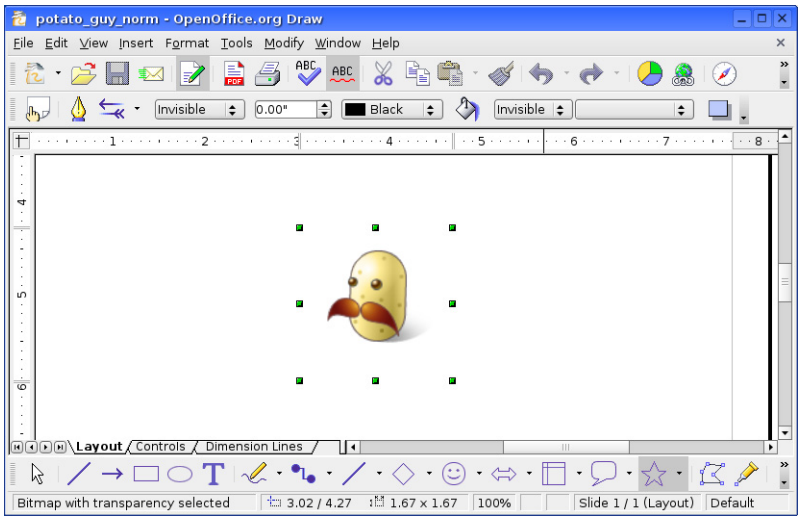
Parameters same as previous section.

Creating your icons

One icon is required so that users can click it to launch your application.



Make your icon 130 x 130 pixels and save it in the .png format. Do not include the name of the application under the icon, just the image.



In the example shown here, an icon is being used in OpenOffice.org Draw. You can set the units of measurement under **Tools > Options > OpenOffice.org Draw > General**, but you cannot set them to pixels in this application. CorelDraw and Paint (on Windows) and GIMP allow you to set the units to pixels.

You can place the image in the following folder, for example:

`/opt/xandros/share/AsusLauncher/new-app.png`

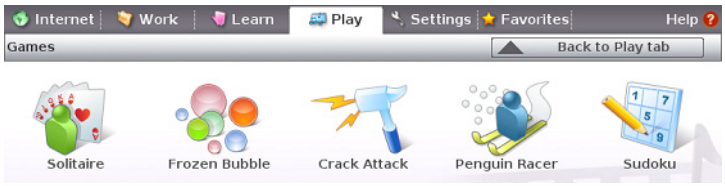
where new-app.png is the name of the new icon.

After you create the icon, you run a script to generate five icons needed (one for normal display and four for color themes). The next section outlines how to run the script.

Adding the application to the ASUS product

Next, you set up the ASUS product to add your application to it.

The Launcher deploys a tabbed interface that is reconfigurable in an XML file called simpleui.rc. This XML file defines the names and icons for all the tabs. All applications that are visible to Launcher have their own components reside in the XML file, and these components are named as XML snippets. An XML snippet defines the name and icons of an application and where the application displays.



On an ASUS computer, when present, the file is located at /var/lib/AsusLauncher/simpleui.rc file. View it in Xandros File Manager by enabling **View > Show All File Systems**, then clicking into the folders on the left side. When this XML file is not found or has errors, Launcher uses the system default /opt/xandros/share/AsusLauncher/simpleui.rc file. A warning message is then displayed once the system enters the tabbed interface. By adding new XML snippets to /var/lib/AsusLauncher/simpleui.rc, the Launcher can be expanded to launch new applications from various tabs. What you will do is create an XML snippet in a .xml file, then merge it.

The following sections describe the required software, the internal structure of the XML snippets, the locations of the snippets, and how new ones can be consolidated into /var/lib/AsusLauncher/simpleui.rc.

Software required

Two software packages are required to add additional XML snippets into Launcher. To install them, update your ASUS product by accessing the **Settings** tab, clicking **Add/Remove Software**, and installing the general updates in the **Settings** tab of the Add/Remove Software window. The packages are:

- **asus-launcher-config** — This package contains three files:
  /etc/AsusLauncher/AsusLauncher.conf
  /opt/xandros/sbin/update-launcher
  /opt/xandros/sbin/create-launcher-icons
- **asus-launcher** — This package is shipped with the system by default, and you update it as outlined here so that Launcher can be made aware of new XML snippets

Creating themed icons

When the cursor is over an application or you click an application in Launcher, a themed icon is rendered for the application. Because there are four color themes in Launcher, four themed icons are required for each application. You run the following script on your icon to generate themed icons:

```
/opt/xandros/sbin/create-launcher-icons
```

For example, if the name of the icon you created for the application is /opt/xandros/share/AsusLauncher/new-app.png, then you run the following command:

```
/opt/xandros/sbin/create-launcher-icons /opt/xandros/share/AsusLauncher/new-app.png
```

This command generates four themed icons in /var/lib/AsusLauncher:

- accessiblity_new-app_hi.png
- business_new-app_hi.png
- student_new-app_hi.png
- home_new-app_hi.png

where these represent the four color themes. We call new-app_hi.png the base name of the themed icons.

Defining new applications in XML snippets

An XML snippet defines the name and icons of an application and defines the tab on which the application displays. You can put your XML snippet code in any file with a .xml extension and in any folder. You can use the <parcel> examples in the /opt/xandros/share/AsusLauncher/simpleui.rc file on your ASUS computer to create your snippet and/or you can use the information outlined here in this section.

A sample XML snippet is as follows:

```
<parcel simplecat="Favorites" extraargs="/opt/xandros/bin/AsusCustomizer"
   icon="add_remove_favorites_norm.png"
   selected_icon="add_remove_favorites_hi.png">
   <name lang="en">Customize</name>
   <name lang="de_DE">Anpassen</name>
   <name lang="nl_NL">Aanpassen</name>
   <name lang="pt_PT">Personalizar</name>
   <name lang="pt_BR">Personalizar</name>
</parcel>
```

Tags and attributes are used. In the example shown, **parcel** and **name** are tags.

The **simplecat** attribute identifies the tab on which the application displays, and the possible values are Internet, Work, Learn, Play, Settings, and Favorites. In the example, the tab is called Favorites, so the application displays in the Favorites tab.

The **extraargs** attribute defines the command (along with the command line options) that Launcher uses when launching the application. An absolute path is used for the command line, otherwise the application is not visible from the tab. Alternatively, an application can be defined by a shortcut attribute, whose value reveals a path to a desktop file (Launcher prepends "/usr/share/applications/" to the path in case of a relative path). Launcher will then parse the desktop file for its command line and possibly the command line options. If the desktop file does not exist, the application

will be invisible from the tabbed interface. In the example, the command line is /opt/xandros/bin/AsusCustomizer.

The **icon** attribute defines the icon to use when the application is not highlighted. No image processing is done on the icon by Launcher, and the icon has to have a size of 130 pixels by 130 pixels. When the icon is defined using a relative path, Launcher prepends "/opt/xandros/share/AsusLauncher/" to it. When the icon is not found, the application is rendered without an image, meaning just the name. In the example, icon /opt/xandros/share/AsusLauncher/add_remove_favorites_norm.png is to be used.

The **selected_icon** attribute defines the base name of the themed icons to use when the application is highlighted. As there are four color themes in Launcher, the base name is prepended with "home_", "business_", "student_", and "accessibility_" to have four icons for the theme sunset, silver, green, and blue respectively. When the icons are defined using a relative path, Launcher prepends "/opt/xandros/share/AsusLauncher/" to the theme names. In case the icons are not found, the application is rendered without an image when highlighted, meaning just a name. Again, these themed icons are used without any processing and they all have to have a size of 130 pixels by 130 pixels. In the example, the resulting themed icons will be /opt/xandros/share/AsusLauncher/{home_ | business_ | student_ | accessibility_}add_remove_favorites_hi.png.

Tag **name**, along with its attribute **lang**, defines the name that Launcher uses for the application at different locales. When the name for a specific locale is not present, Launcher uses the English name as a backup. (The English locale is en, which represents all locales begin with en_) In the example, the English name is "Customize", and "Personalizar" is used for locale pt_BR (meaning Brazilian Portuguese), among others. In case of locale ko_KO, for example, the English name "Customize" is used because there is no corresponding name for locale ko_KO.

In addition to the **parcel** and **name** tags, tags **include** or **exclude** can be used to restrict the locales where the application is visible or invisible. The include tag defines the locales where the application is visible. The exclude tag defines locales where the application is not displayed. You can use the include or exclude tag, but not both. When both tags are present, Launcher ignores the application. The value of these two tags is a space-delimited list of locale names, as depicted in the following example, in which the application is only visible in the locale zh_TW and it_IT.

```
<parcel simplecat="Internet" extraargs="/opt/xandros/bin/eeepc-3g.sh"
   icon="3g_norm.png"
   selected_icon="3g_hi.png">
   <include>zh_TW it_IT</include>
</parcel>
```

Locating XML snippets

The /etc/AsusLauncher/AsusLauncher.conf file defines a list of directories, one directory per line, where XML snippets can be found. When a new XML snippet is installed, one has to make sure the directory where the XML snippet resides is present in the file, and only once. A duplicated directory means an application can appear multiple times in Launcher, and on the same tab. By default, this file is empty.

The name of the XML snippets does not matter but they all need an extension of .xml. To limit the number of files used, you can put many XML snippets into a single file.

Update /etc/AsusLauncher/AsusLauncher.conf to contain the directory with your XML snippet. Run

```
/etc/AsusLauncher/AsusLauncher.conf
```

Updating Launcher

After installing an XML snippet (or a batch of them) and making sure that the directory where the XML snippet resides is present in /etc/AsusLauncher/AsusLauncher.conf, you run the following command as Administrator/root:

```
/opt/xandros/bin/update-launcher
```

This shell script validates the integrity of the XML snippets found in the directories from /etc/AsusLauncher/AsusLauncher.conf and consolidates them into a single XML file. The resulting XML file is validated as well. The shell script discards any XML snippet with errors in it, and if the resulting XML file has errors, Launcher is not updated. When the resulting XML file is syntactically correct, the XML file is moved to /var/lib/AsusLauncher/simpleui.rc, and Launcher is signaled to reload itself.

If for any reason the application is not visible in Launcher, check the integrity of the XML snippet, make sure the directory is in /etc/AsusLauncher/AsusLauncher.conf, re-run /opt/xandros/bin/update-launcher, and watch the output for any error indications.

3 Packaging

After you have developed your application, you prepare, or package it, for inclusion with the ASUS product.

Xandros products are based on a variant of Linux known as Debian. As such, the packaging format used for Debian applies. A packaging wizard, called Package Builder, is included with the software development kit to create the required Debian and other files.

The Debian package file is recognizable by its filename, which is always in the format:

```
<packagename>_<version>_<architecture>.deb
```

For example

```
kcalc_3.4.2.19-5_i386.deb
```

indicates that the KCalc calculator, version 3.4.2.19, is installed on a 32-bit computer (i386) as a Debian package (.deb). So when you package your application, you are creating a .deb file.

Packages consist of the main payload as well as contextual package information. For instance, a typical package can have the payload and scripts to execute upon installation and removal.

Projects that are designed to be built into a package contain a “debian” directory. Within this directory are the input files and templates that comprise the control information within the final package. Every “debian” directory must contain:

- control — Package name, description, architecture, and so on
- rules — Commands for assembling the package
- changelog — Package version and lists of changes for each version

Process for new projects

If you just created your application using Qt and the tools described in this document, use this section. If you have an existing project that is not based on Qt use the next section.

The typical process of creating a new version of a package is to develop and test without using packaging tools. Once you are satisfied with the state of your application, you mark it as a new version. The debian/changelog is then updated with this information, and a formal package is built, tested, and distributed.

The packaging system allows for one file to be "owned" by one package. In other words, packages that contain the same file conflict with each other and cannot be installed simultaneously. You need to design the project to accommodate this situation. For example, language packs can contain the necessary translation files for a program. The translation files themselves have unique names and usually pose no problem in terms of conflicts. But other data files, such as images with text, are sometimes used in a way that makes localization and packaging difficult. A solution is for the images in the various languages to have the same name, but placed in different directories. In that instance, it is up to the program to open the correct image based on the current locale.

C and C++ projects on Linux typically use GNU Makefiles for compiling. Having the Makefiles with proper GNU rules helps packaging immensely. The Makefiles contain rules for "install" and "clean", and contain a default "all" rule to build the project.

When using the suggested development environment, which uses qmake to build the project, the Makefiles are automatically managed by qmake. The required Makefile rules are prepared by default, but you must adjust the default packaging template to match qmake's generated Makefiles. To do this, simply edit the debian/rules file, and replace any instances of $(DESTDIR) with $(INSTALL_ROOT). Normally, there is only one line in the "install" rule that uses this variable.

When qmake is used to manage your Makefiles, it needs to be executed at the appropriate time during the package building phase. To do this, edit the debian/rules file, and add the appropriate qmake command in the "configure-stamp" rule. An example is:

```
configure-stamp:
        dh_testdir
        # Add here commands to configure the package.
        qmake-qt4

        touch configure-stamp
```

Process for existing projects

If you have a pre-existing project that does not use qmake, an additional requirement for packaging is that the “install” rule accepts a $DESTDIR environment variable. The packaging scripts use this rule to install the project to a temporary directory from with the package is created. By default, the scripts call the rule this way:

```
make install DESTDIR=$(TOPDIR)/debian/tmp
```

A typical Makefile install target can look like this (without the $DESTDIR variable):

```
install: mybinary
   install -d $(PREFIX)/bin
   install -m 0755 mybinary $(PREFIX)/bin
```

Now change it to include the $DESTDIR variable:

```
install: mybinary
   install -d $(DESTDIR)/$(PREFIX)/bin
   install -m 0755 mybinary $(DESTDIR)/$(PREFIX)/bin
```

Because the $DESTDIR variable is not normally set during builds, it does not affect normal development.

The point is that packaging is greatly simplified when you take advantage of the default rules behaving as they intended. The default rules executed are:

- make clean
- configure (optional, if autotooled)
- make
- make install (with DESTDIR)

If you create your Makefiles to conform to these simple commands, it is likely that the default package template will work for basic functionality.

Make any changes now to your application, for example to accommodate translations, and use Makefiles.

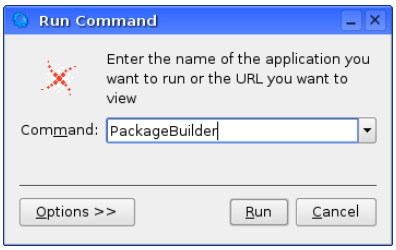
Packaging your application

You create a package from your source code tree. A Package Builder wizard is provided to help you create the initial packaging template. Package Builder is a front-end to dh-make and creates a template directory that you can easily edit for what your product requires.

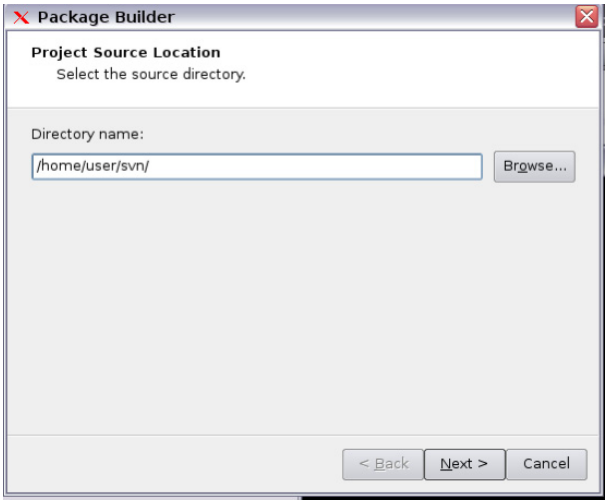
To package your application

1 Click **Launch > Run Command**. The Run Command window opens.

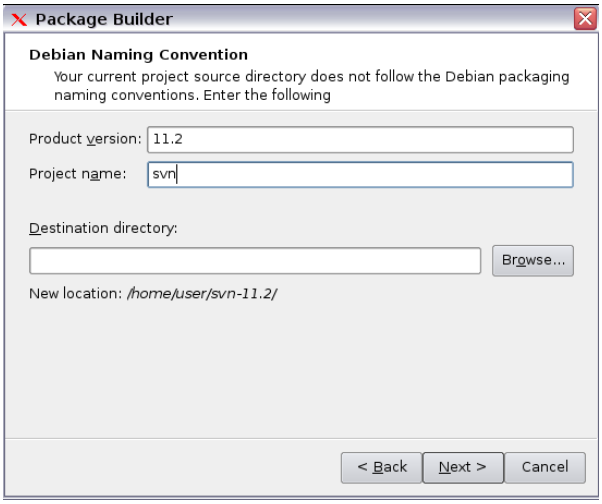
2 Type `PackageBuilder` and press **Enter**. The Package Builde wizard launches.



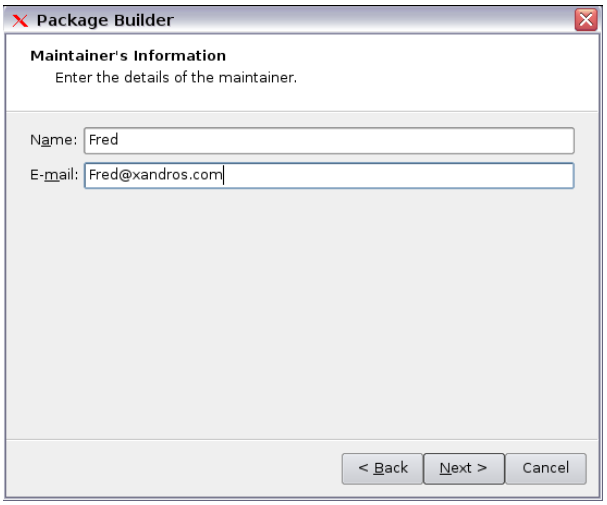
3 Add the location of the source directory. This is the top-level project directory, where the top-level Makefile exists. Then click **Next** to continue in the wizard.



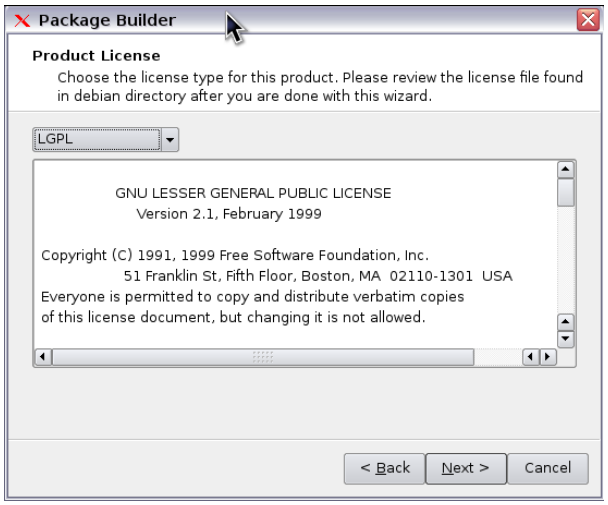
4 Provide the package information: an initial version number and a name for the package. This name needs to be lower-case, and the only punctuation allowed is a hyphen (“-”). Enter a destination directory for the output, such as a folder within your home folder. Then click **Next**.



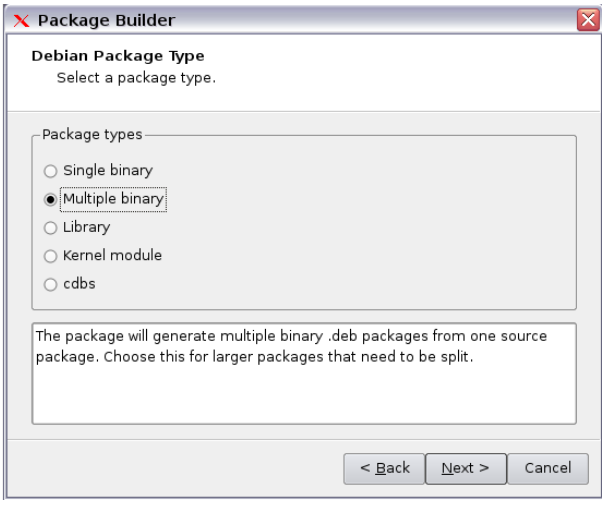
5 Provide developer contact information. This is your name and e-mail address. Then click **Next**.



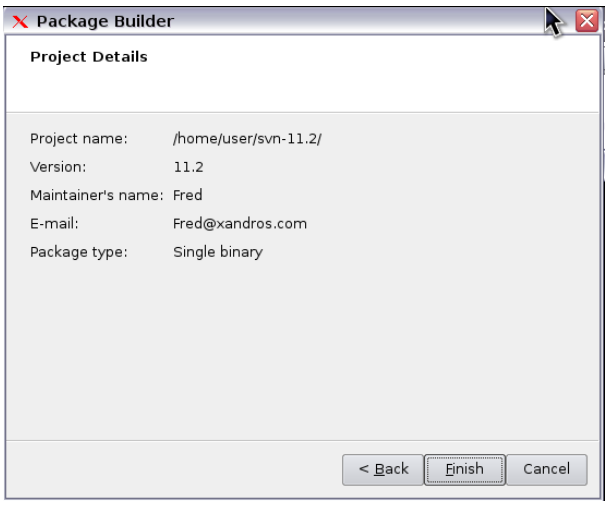
6 Packaging policy is to include a license with every package. By default, the wizard sets the GNU general public license (GPL) as the license, but another license can be entered if required. Click **Next**.



7 The next window determines the structure of the packaging. Most applications fall under the definition of **Single binary**, where there is one package containing binaries. Some applications are better suited to use the **Multiple binary** option, where there are several packages with binaries; an example is a program with several localization packs. The **Library** template is similar to multiple binary packages, except that it automatically creates templates for library development and documentation packages. Similarly, the **Kernel module** template creates a template that is useful if you want to package a kernel module and has optional tools. For the common Debian build system, use the **cdbs** option. It is a separate system from debhelper, and provides similar functionality; its functions are beyond the scope of this document. Click **Next**.



8 Check your selections, and click **Finish** to package your application.



Checking your package for errors

After packaging your application, you can check it for errors. Lintian is software that checks your code for errors and policy violations. You can obtain it free from **http://lintian.debian.org**

Building your package

After the packaging has been done, build the package. Change directories (cd) to the project's top-level directory (the one containing the "debian" directory), then execute:

```
dpkg-buildpackage -rfakeroot
```

When successful, this process outputs a .deb package in the project's parent directory, a .tar.gz file that contains a copy of the current version of the source, and a .dsc file referencing the two previous files. When unsuccessful, see the error message, correct your method, and repeat the build.

Testing your package

To test your work, install it on the ASUS product using the following command:

```
dpkg -i
```

for example

```
dpkg -i kcalc_3.4.2.19-5_i386.deb
```

Maintaining your package

As your project grows, the packaging will grow with it. Even if the project does not, there are changes that need to be managed in the “debian” directory.

Updating the version

As mentioned earlier, the version is kept in the “debian/changelog” file. The most convenient way to update the version, while keeping the format of the file correct, is to use the “dch” (Debian changelog helper) command. Typically, it is done at the top level of the project, which is the directory that houses the “debian” directory, where you execute:

```
dch -i
```

This starts an editor with a template entry for a new version of the package. All of the fields can be edited, but take care to preserve the overall format of the file.

Adding maintainer scripts

Maintainer scripts are executed at each package install, upgrade, and removal. They reside in the “debian” directory as “preinst”, “postinst”, “prerm”, and “postrm”. Templates with an extension “.ex” are provided. To use one of the templates, simply rename it to the proper file name. For example:

```
mv debian/postinst.ex debian/postinst
```

The file can now be edited to suit your needs.

4 Virtual machine use

VMware is software that runs various operating systems in windows on your computer desktop. The installed operating systems are referred to as virtual machines (VMs). You install VMware, install compatible operating systems, and run them. For the current purpose, you can install VMware on whatever computer you are currently using, then install Xandros Desktop - Open Circulation Edition (to develop your application) and/or the ASUS product (to test your application). Use is optional; if you have Xandros Desktop - Open Circulation Edition installed on a computer and you have an Eee PC or EP20 computer, then you do not need VMware.

The ASUS product, such as the Eee PC, includes a DVD that contains the operating system and software in a .iso format, called an ISO image. It is easy to convert that image to one that VMware can use.

This chapter outlines how to install VMware and create a VMware image from an Eee PC ISO.

Installing VMware

A free VMware Player is available, for installation on a Windows or Linux computer, for example. Xandros recommends using a powerful computer with lots of RAM, for example 1 GB or more.

To install VMware Player on Windows

1. Download VMware Player from **http://www.vmware.com/download/player** You need to register.
2. Double-click the .exe file to launch the VMware wizard and install it.
3. After VMware is successfully installed, access the application in the menu, for example **Start > All Programs > VMware > VMware Player**.

To install VMware Player on Linux

1. Download it from **http://vmware.com/download/player** You need to register. In the example here, we use the .tar file.
2. Open a console/terminal window and switch to Administrator/root:

   ```
   su root
   ```

3. View the VMware file name by entering the `ls` command.
4. Uncompress the file, for example

   ```
   tar zxf VMware-player-2.0.2-45731.i386.tar.gz
   ```

   The command to use varies with the version of VMware, for example 32-bit or 64-bit.
5. Access the folder created, for example

   ```
   cd vmware-player-distrib
   ```

6. Launch the installation program by entering the following command

   ```
   ./vmware-install.pl
   ```

7. Accept the defaults in the installation program, except enter no for NAT.
8. After VMware is successfully installed, access the application in the menu, for example **Applications > System > VMware Player**.

Creating a VMware image

You create a VMware image from an ISO file, for example the Eee PC ISO file. This allows you to run the Eee PC as a virtual machine in a window on your computer desktop.

The software development kit provides a simple script to convert the contents of an ISO image into the required VMware disk (VMDK). The script is called "vmware-convert" and its use is outlined in the first procedure here. We also provide instructions on creating an image with extra space.

To convert an ISO image into the VMware format

- Open a console/terminal window and enter the following command:

  ```
  su root
  ```

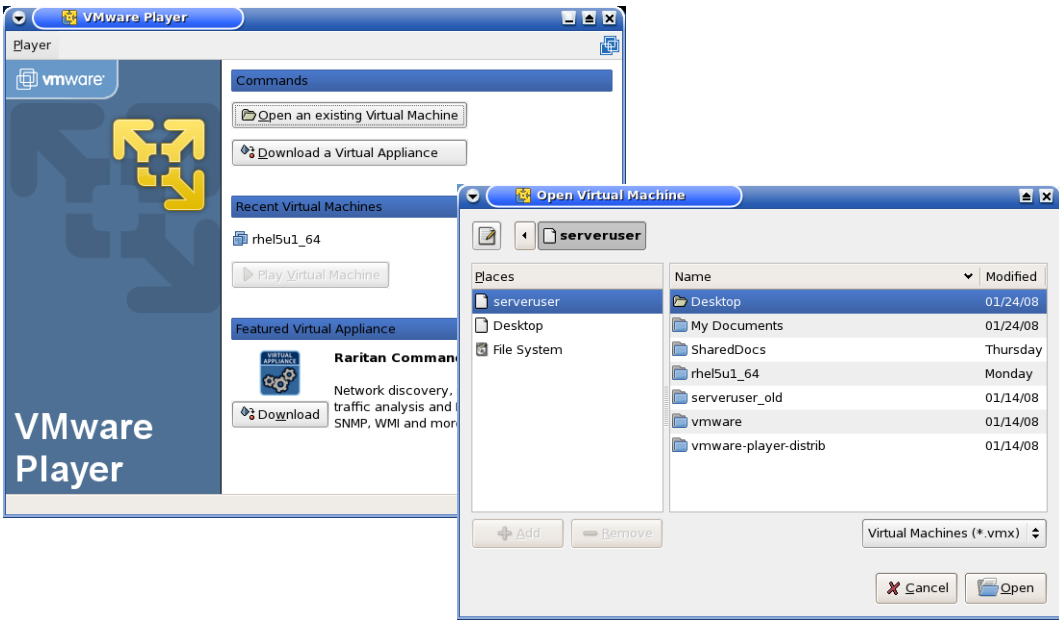
  to change to Administrator/root, then

  ```
  vmware-convert image.iso
  ```

  where you substitute the name of the ISO image for image.iso

The script takes five to ten minutes to execute, and the output is a file called /tmp/asus.vmdk. This file, along with asus.vmx from /usr/share/doc/vmware-convert, can be placed together in a clean folder.

The virtual machine can now be started by opening the asus.vmx file from within VMware. For example, you launch the VMware Player, click **Open an existing Virtual Machine**, then search for the .vmx file to open.



Creating an image with extra space

Space within a VM image is limited. Many developers find a need for more free space. A simple way to free up space is to remove a large application, such as OpenOffice.org and Adobe Reader.

To delete applications

- In a console/terminal window, enter the following command:

  ```
  sudo apt-get remove openoffice.org-common acroread
  ```

  where OpenOffice.org and Adobe Reader have been identified for removal.

  This quickly frees up to an additional 500 MB of space within the VM.

5 Miscellaneous

This chapter outlines translation into other languages and links for more information.

Localizing your application

ASUS products are translated into several languages, so consider translating your application into other languages too. Localization is also referred to as internationalization.

Use these general guidelines:

- Use tr() around every user-visible string
- Use Qt's layout support
- Do not break strings up; use the QString::arg() facility

Use the following specific guidelines.

Use tr() around user-visible text

Example:

```
tr("Hello world")
```

Use the QString::arg() function for dynamic text

Example:

```
tr("%1 out of every %2 of apples are bad.").arg(nCount).arg(nTotal)
```

This allows the translators to reorder the arguments if needed for that language. It also allows the translator to understand the context, compared to breaking up the string. Example:

bad — `tr("The user '") + strUser + tr(' is invalid.")`

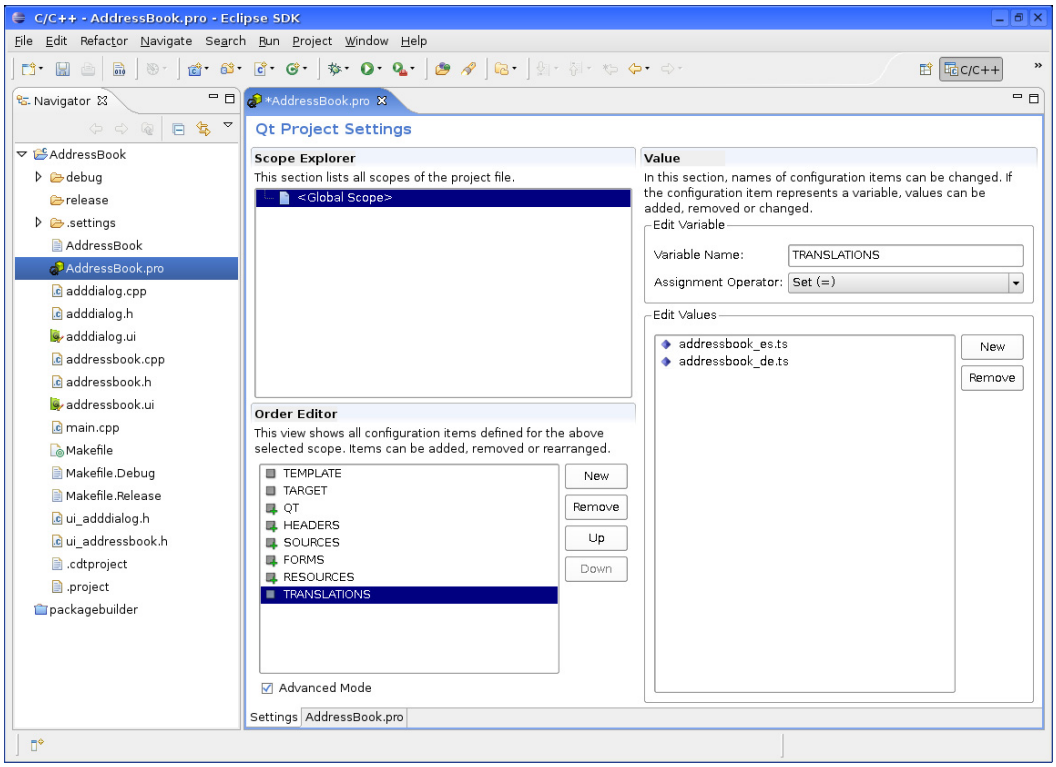
good — `tr("The user '%1' is invalid.").arg(strUser)`

Set the Eclipse IDE for translations

In the Eclipse IDE, open the project's .pro file (double-click). At the bottom, click **Advanced Mode**. Directly above in the **Order Editor**, click **New** and then **Add Variable**. In the top right, clear the **Variable Name** and set it to "TRANSLATIONS". Set the **Assignment Operator** to "Set (=)". In the box below, start adding values in the form "<application name>_<language code>.ts". Example:

```
addressbook_es.ts
addressbook_de.ts
```

This naming convention is only a guideline and not a rule.



Extract the translatable strings

To extract the translatable strings, you need to run the "lupdate" command on the project file within the project directory. For example, run the following command:

```
lupdate addressbook.pro
```

This gives you "addressbook_es.ts" and "addressbook_de.ts" files.

Use the Qt Linguist program to translate

To translate the strings, you use the Qt Linguist program. You can find it in the Launch menu under **Applications > Development > Qt 4 Linguist**.

Convert files to binary files

The localization files now need to be converted into binary translation files before they can be used. From a console window, run the "lrelease" command, for example:

```
lrelease addressbook.pro
```

This gives you a ".qm" file for each ".ts" file.

Load the files

To load the translation files, you add a small amount of code to your project.

1. Open the main.cpp file.
2. After the "QApplication a(argc, argv);" line, add the following lines:

```
QTranslator translator;
translator.load("addressbook_" + QLocale::system().name());
a.installTranslator(&translator);
```

3. Recompile.

Test your files

To test your new translation file, run your application with an altered environment to mimic a different locale. In the Run dialog, click the **Environment** tab. Click **New...** and add a new environment variable of "LC_ALL" and set it to the translation you want to test, for example "es".

Click **Run** to start the application. When successful, you see your translated strings displayed.

For more information, see the Qt internationalization document at **http://doc.trolltech.com/i18n.html**

Links for more information

More information can be found at the following Web sites.

Eclipse

http://www.eclipse.org

Web site of the Eclipse platform.

Introduction to Eclipse for Visual Studio Users

http://www.ibm.com/developerworks/opensource/library/os-eclipse-visualstudio/?ca=dgr-eclipse-1

A document that introduces users familiar with Visual Studio to the workings of Eclipse.

Trolltech

http://www.trolltech.com/

Web site of the company that produces the Qt library.

Qt Online Reference

http://doc.trolltech.com/4.3/index.html

This site contains a complete reference for the Qt 4.3 application programming interface.

GNU Make files

http://www.gnu.org/software/make/manual/

For help on Makefiles.

Debian Policy Manual (for packaging)

http://www.debian.org/doc/debian-policy/

This site is the official packaging policy manual for Debian packages. While some of the manual describes Debian project-specific policy, the majority describes best practices when producing packages.

Debian packages

http://packages.debian.org/

The official Debian software repository is searchable using this site. Once a package is found, you can download it. Upstream packages contain many examples of good packaging.

File system hierarchy standard

http://www.pathname.com/fhs

This web site contains the standard applied to Linux programs about where to install files and the hierarchy rules of the file system.

freedesktop.org specifications

http://www.freedesktop.org/wiki/Specifications

Specifications for common Linux desktop elements can be found here. Those on Desktop Entry and the Icon Theme, for example, are useful to develop for the ASUS Linux platform.