MOTODEV

The Motorola developer network

Quick Start: Your First Application

MOTODEV Studio for Linux

Quick Start: Your First Application
MOTODEV Studio for Linux

version 1.0

July, 2008

For the latest version of this document, visit http://developer.motorola.com.

Motorola, Inc.

http://www.motorola.com

Contents

Chapter 1: Creating the Basic Application

This chapter walks you through the creation of “First Base”, a simple decimal-hexadecimal-binary converter written entirely using native Linux® APIs for the MOTOMAGX™ platform. The application has a simple interface, consisting of three line-edit widgets (“text fields,” on other platforms); as you type digits into one widget, the other two widgets display the converted values. The application prevents incorrect characters from being typed into a field. For instance, you can only enter decimal digits into the Decimal widget; letters and symbols are ignored.

Although the application used throughout this tutorial is a relatively simple one, it illustrates some fundamental technologies that are used by nearly all well-written MOTOMAGX applications. In particular, it shows the use of various MOTOMAGX UI framework objects, including `ZKbMainWidget`, `ZLineEdit`, `ZSoftKey`, `ZFormContainer`, `ZAppInfoArea`, and `ZApplication`. It also illustrates the use of the MOTOMAGX Qt framework. Motorola engineers began with Trolltech’s Qt, version 2.3.6, and modified it to suit the mobile platform. This gives native Linux applications written for the MOTOMAGX platform access to many of the familiar Qt classes, as well as Qt’s powerful signal/slot mechanism for dynamic inter-object communication.

This chapter walks you through the steps needed to create the basic First Base application. Chapter 2 then shows you how to restructure the application so that it follows the Model-View-Controller (MVC) architecture pattern. Although not specific to MOTOMAGX, applications that employ this pattern maintain a clear division between the business logic and the user interface, making it easier to modify either the look of the application or the underlying business rules without affecting the other.

Before You Begin

In order to follow along with this tutorial you should have a basic knowledge of C++ programming. As well, you should have installed and be familiar with MOTODEV Studio for Linux. You need not be familiar with either the MOTOMAGX APIs or Qt.

If you have not yet installed MOTODEV Studio for Linux, you should download it from the MOTODEV web site (http://developer.motorola.com/) and install it now. If you are new to MOTODEV Studio, you should read *Getting Started: MOTODEV Studio for Linux* to acquaint yourself with the tool and its use. This document is also available from the MOTODEV web site.

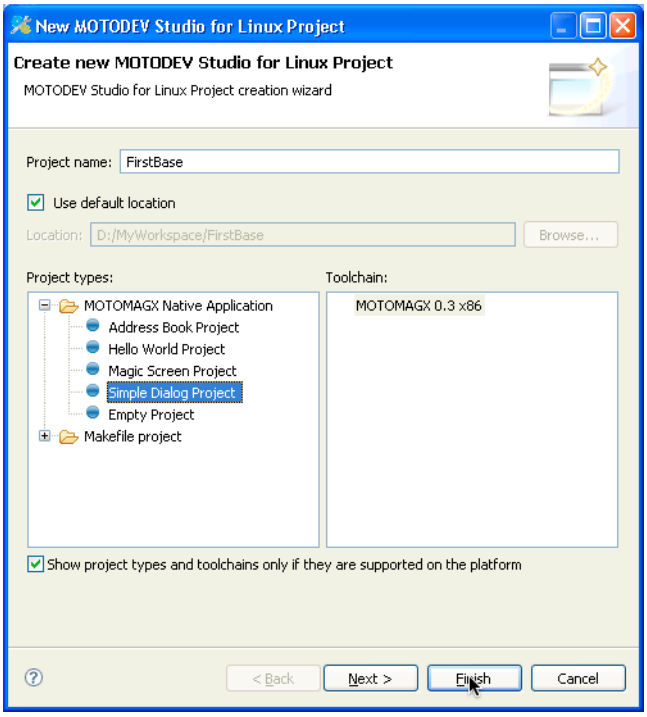
IMPORTANT: This tutorial assumes that you have already read *Getting Started: MOTODEV Studio for Linux*.

Create a New Project

As mentioned in *Getting Started: MOTODEV Studio for Linux*, the easiest way to start a new project is to begin with one of the samples and then modify it. The following steps show you to do this for your project:

1. Launch MOTODEV Studio for Linux.
2. On the **File** menu, point to **New** and then click **MOTODEV Studio for Linux Project**.

   The New MOTODEV Studio for Linux Project dialog appears.



3. Type a name for your project in the **Project name** field. For instance, “FirstBase”.
4. Ensure that **Use default location** is selected. This causes the project to be created in your chosen workspace.
5. Under **Project Types**, click the plus sign next to “MOTOMAGX Native Application” to expose the list of sample applications. Click “Simple Dialog Project”.
6. Click **Finish**.

   Your project is created and, if the “Build automatically” workspace preference[1] is set, it is built automatically. If you do not have automatic builds enabled, build your project manually by right-clicking it in the Project Explorer and choosing **Build Project** from the menu that appears (note that the **Build Project** option does not appear in the menu if automatic building is enabled).

You can verify that your project built correctly by checking the build messages displayed on the Console tab.

1.To check or alter this preference, select Preferences from the Window menu, click the plus sign next to “General” to expand the list, and then click Workspace.

Create a Launch Configuration

Eclipse™—and thus MOTODEV Studio, which is based upon Eclipse—uses two types of configurations to define how your project is built and run:

- A *build configuration* encapsulates compiler, linker, and other tool settings. For example, a debug build configuration might specify that all compiler warnings are to be turned on and might define certain symbols to be used in your code (for instance, `#if DEBUG`). By default, both a release and a debug configuration are defined when a MOTOMAGX project is created, with the debug build configuration set as the default. You can change the active build configuration by right-clicking the project in the Project Explorer and choosing **Build Configurations > Set Active**.
- A *launch configuration* encapsulates all of the information relating to a single execution scenario. This configuration identifies both the project package and the device on which it is to be run. You must create at least one launch configuration for your project, but you may choose to create several: for instance, one for debugging and one for your release build, or one for the emulator and one for an actual device.

Once you create a launch configuration, you may use it to either run or debug your application. Typically, you debug your application using a launch configuration based on a package created with your debug build configuration.

By default, when you build your project it creates a debug package. Follow these steps to create a launch configuration that you can use to debug your application on the MOTOMAGX Emulator:

1 In the Project Explorer, right-click on the FirstBase project[1] and select **Debug As > MOTODEV Application**.

The Debug Launch Configuration dialog is displayed.

2 Enter “FirstBase Debug Emulator” in the **Name** field.

3 Ensure that the project type is “MOTOMAGX native application”, the project is “FirstBase”, and the Application/Package is “MOTOMAGX_0.3_x86_Debug/package.descriptor” (the version number following “MOTOMAGX” may not be 0.3 for your installation, depending upon the MOTODEV Studio version you have installed). The **Device** field should specify the MOTOMAGX Emulator, and the **Deployer** field should be set to the MOTOMAGX Emulator deployer.

4 Click **Apply** to save the new launch configuration, then click **Debug** to launch the application for debugging, thereby verifying that the application runs correctly on the emulator.

The emulator will launch (there will be some confirmation dialogs) and the application will be deployed to it. Because you launched the application for debugging, MOTODEV Studio will switch to the

1.You will notice that this tutorial always directs you to right-click on a project in the Project Explorer and choose actions from the pop-up menu, rather than use the toolbar icons or the main menu bar. This is because Eclipse lets you have several projects open at the same time (a feature that is especially useful when copying code from one project to another), and it is easy to have the wrong project selected when building, cleaning, importing new code, etc. When you select the wrong project, it can be difficult to figure out why things don't work as you expect. If you get into the habit of selecting the target project by right-clicking on it and using the pop-up menu, you will avoid this confusion.

debugging perspective and halt program execution at the first line of executable code in the program; your window should look something like Figure 1.

5 Halt the debugger by clicking the “stop” button ( ) in the Debug view toolbar.

6 Switch back to the MOTODEV Studio for Linux perspective.

7 Close the MOTOMAGX Emulator view and confirm that you want to shut down the emulator.

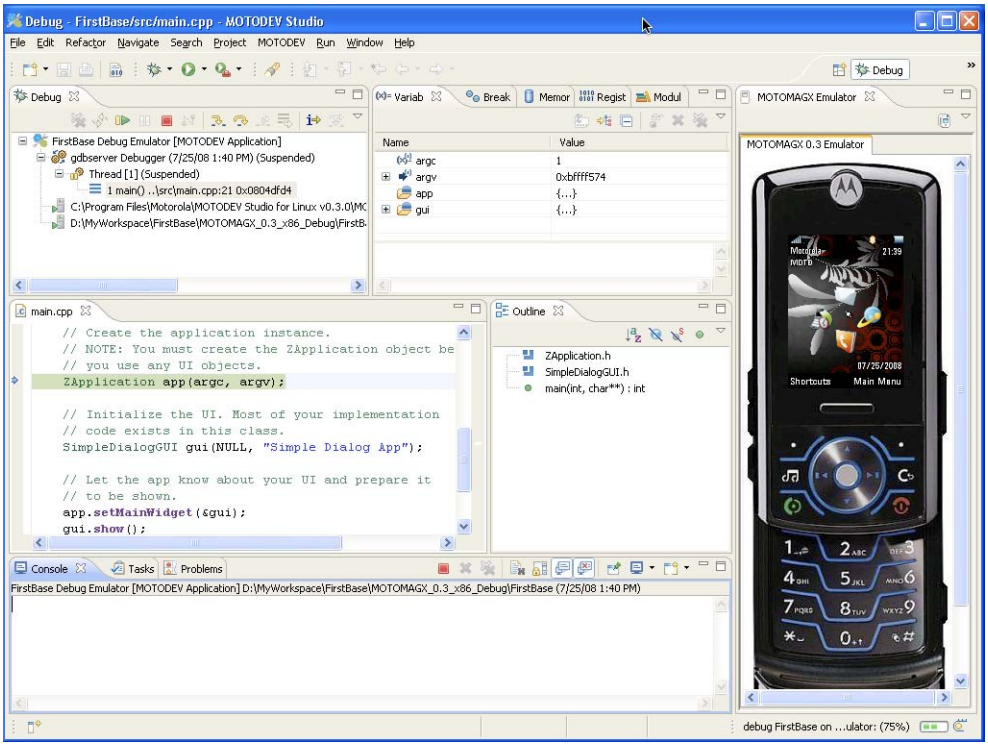


Figure 1: MOTODEV Studio for Linux debug perspective

Rename the Files and Classes

When you create a project based upon the Simple Dialog sample application, three source files are created for you:

- `main.cpp` contains the application's `main()` entry point function. This is where your application's class is instantiated, its primary user interface (UI) widget is set and shown, and control is given over to the application object.
- `SimpleDialogGUI.h` and `SimpleDialogGUI.cpp` contain the definition and implementation of the application's primary UI widget class. You will be makng extensive changes to their contents.

Although you will be making minor changes to `main.cpp`, the name of the file can remain as is. The `SimpleDialogGUI` files, however, should be renamed to reflect the new application's name. While you could delete them and create a new class from scratch, it is simpler to use some of the existing code and delete what you don't need. The following steps walk you through the renaming process:

1. In the Project Explorer, open the `src` folder.
2. Right-click `SimpleDialogGUI.h` and choose **Rename**.
3. Type “FirstBaseUI.h” and press ENTER.
4. Using the same procedure, rename `SimpleDialogGUI.cpp` to `FirstBaseUI.cpp`.

Next, you need to change the UI widget's class name. You can use Eclipse's handy “refactor” feature to change the class name wherever it appears in your project:

1. Open `FirstBaseUI.h`.
2. In the Outline view, click `SimpleDialogGUI`. This will scroll the editor to the class’s definition and select the class name.
3. In the source editor, right-click the selected text (the class name) and choose **Refactor > Rename** from the popup menu.
4. In the **Rename to** field, type “FirstBaseUI”. Specify the scope as “project”, and indicate that you want to update within source code and comments.
5. Click **Preview**. Eclipse displays a dialog indicating that there are a number of potential matches. There will also be two parsing errors, but these can be ignored.
6. Click **Continue**. Eclipse displays the Preview dialog, allowing you to explore the changes that will be made.
7. Click **OK**. Eclipse changes all instances of the widget class to `FirstBaseUI`.

Unfortunately, the Refactor feature ignores `#include` statements. As well, due to a bug in Eclipse the constructor and destructor methods are renamed in the class header file but not in the implementation file. You can use a global search-and-replace operation to catch the remaining instances of “SimpleDialogGUI”:

8. Select **Search** from the **Search** menu.
9. On the **File Search** tab, type “SimpleDialogGUI” into the **Containing text** field.
10. Click **Replace**. The Replace dialog is displayed.
11. Type “FirstBaseUI” into the **With** field, then click **Replace All**. Note that you can make the replacements one-by-one by repeatedly clicking **Replace** instead of **Replace All**.
12. If MOTODEV Studio does not automatically rebuild your project after replacing all remaining occurrences of “FirstBaseUI”, make sure that all your changes have been saved (from the **File** menu, select **Save All**) and build your project. This will ensure that everything you have done so far is correct.
13. Change the line in `main.cpp` that reads:

```
FirstBaseUI gui(NULL, "Simple Dialog App");
```
to
```
FirstBaseUI gui(NULL, "First Base");
```

14 Finally, change all occurrences of `SIMPLEDIALOGGUI_H_` to `FIRSTBASEUI_H_` in `FirstBaseUI.h`.

Add the UI Widgets

In this section, you will create the First Base application's user interface.

Before adding new code, let's take a moment to examine the application's structure. The `main()` function in `main.cpp` looks like this:

```
int main(int argc, char** argv) {
    // Create the application instance.
    ZApplication app(argc, argv);

    // Initialize the UI. Most of your implementation
    // code exists in this class.
    FirstBaseUI gui(NULL, "First Base");

    // Let the app know about your UI and prepare it
    // to be shown.
    app.setMainWidget(&gui);
    gui.show();

    // Give control to the app object (events, etc.)
    return app.exec();
}
```

This is a typical `main()` function for any MOTOMAGX application that has a user interface. It starts by creating a `ZApplication` object, passing any application arguments to its constructor. The `ZApplication` object is important because it handles events (such as key presses) and passes them to the application's UI object.

The second line of code constructs the UI object, passing the object's parent and the application's title. Since `FirstBaseUI` is the top-level widget in the user interface, you pass `NULL` for its parent object.

The third line of code tells the application object that the UI object (`gui`) is the application's main widget. Since `gui` inherits from `ZKbMainWidget`, it already "knows" how to handle events such as navigation keys presses.

The fourth line tells the UI object to become visible once the application is active.

The final line calls `exec()`, which starts the application event loop. This loop continues to run until the application object's `quit()` function is called.

The First Base application's user interface consists of:

- Three line-edit widgets, one each for decimal, hexadecimal, and binary values. To keep things simple, the binary widget will be read-only.
- An `AppInfoArea` widget, which displays keyboard input modes.

- A soft key widget, which allows the user to exit the application.

Since the `FirstBaseUI` class contains the application's functionality, `FirstBaseUI.cpp` and `FirstBaseUI.h` are where you will make your changes.

1 Open `FirstBaseUI.h`.

2 Delete the following #includes, since the First Base application won't be using any of these objects:

```
#include <ZGroupBox.h>
#include <ZPressButton.h>
#include <ZRadioButton.h>
#include <qstring.h>
```

3 In the `FirstBaseUI` class definition, delete all of the private slot functions except for `rightButtonClicked()`. This one remaining function will be called when the right soft key is pressed.

NOTE: If you are new to Qt application development, you may not be familiar with slots. They will be discussed the section "Add Base Conversion Logic" on page 10.

4 Delete all of the current data members ("initialTitle" through "button").

5 Add three new private data members, one for each line-edit widget:

```
ZLineEdit *lineEditDec;
ZLineEdit *lineEditHex;
ZLineEdit *lineEditBin;
```

6 Save your changes. The header file should look similar to this (comments have been removed for the sake of brevity):

```
#ifndef FIRSTBASEUI_H_
#define FIRSTBASEUI_H_

#include <ZKbMainWidget.h>
#include <ZLineEdit.h>

class FirstBaseUI : public ZKbMainWidget
{
Q_OBJECT

public:
  FirstBaseUI(QWidget *parent=0, const char* name=0);
  ~FirstBaseUI();

private slots:
    void rightButtonClicked();

private:
  ZLineEdit *lineEditDec;
  ZLineEdit *lineEditHex;
  ZLineEdit *lineEditBin;
};

#endif
```

Next, edit the `FirstBaseUI` constructor to add the new line-edit widgets and set up the soft keys, as follows:

7 Open `FirstBaseUI.cpp`.

8 Delete all of the code within the `FirstBaseUI` constructor, but leave the constructor itself.

The `FirstBaseUI` constructor should create and initialize the user interface in four sections: the main widget's title area, the application information area, the three line-edit widgets, and the soft keys.

9 Set the main application widget's title, using the name that was passed to the constructor:

```
ZKbMainWidget::setAppTitle(name);
```

10 Create a `ZAppInfoArea` widget to display contextual information and set it as the application's title bar:

```
ZAppInfoArea *aia = new ZAppInfoArea(this);
this->setTitleBarWidget(aia);
```

The `ZAppInfoArea` widget contains keyboard entry status indicators, which will be provided automatically by the system as necessary. You should always add one to any application that permits keyboard entry via widgets such as the `ZLineEdit` widget. Associate the `ZAppInfoArea` widget with the main application widget by calling `setTitleBarWidget()`.

11 Create a form container to hold the line-edit widgets:

```
ZFormContainer *fc = new ZFormContainer(this);
this->setContentWidget(fc);
```

The `ZFormContainer` takes its parent widget as a parameter, and is then associated with the main widget using `setContentWidget()`. Shortly you will add the line-edit widgets to the form container using the `ZFormContainer` object's `addChild()` function.

12 Create and initialize the line-edit widget for decimal numbers.

```
lineEditDec = new ZLineEdit(this);
lineEditDec->setTitle("Decimal:");
lineEditDec->setTitlePosition(ZLineEdit::TitleTop);
lineEditDec->setMaxLength(5);
```

The above code sets the maximum length of the widget to be five characters. This is because the handset's screen isn't wide enough to display the binary representation of decimal numbers larger than 65,535 (1111111111111111 in binary). Also note that this code is placing the widget's title above the widget (`ZLineEdit::TitleTop`) rather than in the default title position (`ZLineEdit::TitleLeft`). The default position would use less vertical screen space for each widget, but would present two problems. First, the width of the input area of the widget is determined by the width of the form container, minus the width of the widget's title. Since the titles will all have differing lengths, the input areas of the three `ZLineEdit` widgets wouldn't align vertically, which is visually distracting. Second, using the default title alignment greatly reduces the amount of width available to display the numbers, which is especially critical for the binary widget.

13 Create and initialize the line-edit widget for hexadecimal numbers:

```
lineEditHex = new ZLineEdit(this);
lineEditHex->setTitle("Hex:");
lineEditHex->setTitlePosition(ZLineEdit::TitleTop);
lineEditHex->setMaxLength(4);
```

14 Create and initialize the line-edit widget for binary numbers. Note that unlike the other two, this widget will only be used to display values.

```
lineEditBin = new ZLineEdit(this);
lineEditBin->setTitle("Binary:");
lineEditBin->setTitlePosition(ZLineEdit::TitleTop);
lineEditBin->setMaxLength(16);
lineEditBin->setReadOnly(true);
```

The `setReadOnly()` function prevents the user from entering characters in the widget, but changes nothing else about the appearance or behavior of the widget. For example, it is visually identical to a widget than can accept input (it has a frame around the input area), and can be selected using the navigation keys.

15 Add the three `ZLineEdit` widgets to the form container:

```
fc->addChild(lineEditDec);
fc->addChild(lineEditHex);
fc->addChild(lineEditBin);
```

16 Create and initialize the left and right soft keys. The left key should be disabled, and the right key should have an “Exit” label:

```
ZSoftKey *pdlgSoftKey = this->getSoftKey();
pdlgSoftKey->setText(ZSoftKey::RIGHT, "Exit");
pdlgSoftKey->setClickedSlot(ZSoftKey::RIGHT,
     this, SLOT(rightButtonClicked()));
pdlgSoftKey->disableClickedSlot(ZSoftKey::LEFT);
pdlgSoftKey->setText(ZSoftKey::LEFT, "");
```

As you might guess, the `setClickedSlot()` call specifies that the `rightButtonClicked()` function should be called when the user presses the right soft key.

17 Save your changes. The full constructor should look like this:

```
FirstBaseUI::FirstBaseUI(QWidget *parent, const char* name) :
  ZKbMainWidget(parent, name) {

  ZKbMainWidget::setAppTitle(name);

  ZAppInfoArea *aia = new ZAppInfoArea(this);
  this->setTitleBarWidget(aia);

  ZFormContainer *fc = new ZFormContainer(this);
  this->setContentWidget(fc);

  lineEditDec = new ZLineEdit(this);
  lineEditDec->setTitle("Decimal:");
  lineEditDec->setTitlePosition(ZLineEdit::TitleTop);
  lineEditDec->setMaxLength(5);

  lineEditHex = new ZLineEdit(this);
  lineEditHex->setTitle("Hex:");
  lineEditHex->setTitlePosition(ZLineEdit::TitleTop);
  lineEditHex->setMaxLength(4);

  lineEditBin = new ZLineEdit(this);
  lineEditBin->setTitle("Binary:");
  lineEditBin->setTitlePosition(ZLineEdit::TitleTop);
  lineEditBin->setMaxLength(16);
  lineEditBin->setReadOnly(true);

  fc->addChild(lineEditDec);
  fc->addChild(lineEditHex);
  fc->addChild(lineEditBin);
```

```
  ZSoftKey *pdlgSoftKey = this->getSoftKey();
  pdlgSoftKey->setText(ZSoftKey::RIGHT, "Exit");
  pdlgSoftKey->setClickedSlot(ZSoftKey::RIGHT,
       this, SLOT(rightButtonClicked()));
  pdlgSoftKey->disableClickedSlot(ZSoftKey::LEFT);
  pdlgSoftKey->setText(ZSoftKey::LEFT, "");
}
```

18 Delete the `FirstBaseUI` destructor's contents, and then modify it to look like this:

```
FirstBaseUI::~FirstBaseUI()
{
  delete lineEditDec;
  delete lineEditHex;
  delete lineEditBin;
}
```

19 Delete all other functions in `FirstBaseUI.cpp` except for `FirstBaseUI::rightButtonClicked()`. That function, which causes the application to exit when the right soft key is clicked, is fine as written.

Note that you can use the outline view to quickly scroll to these extraneous functions.

20 Save your changes.

Although it is not yet complete, now is a good time to compile and run the application (it should build with no errors). The application should display its interface and allow you to navigate from widget to widget. You should be able to enter characters into the Decimal and Hex line-edit widgets, but not into the Binary widget[1]. As yet the application doesn't do anything when you enter characters, and it doesn't prevent you from entering alphabetic characters into the decimal field, but, when you press the right soft key, the application should exit.

Add Base Conversion Logic

The First Base application should convert a number to the other bases as it is entered into a `ZLineEdit` widget. This is accomplished by connecting each line-edit widget's `textChanged()` function to a new function that will accept the changed string and update the other two line-edit widgets. Rather than simply calling functions directly, you can connect widgets to functions using QT's signal/slot mechanism.

Qt extends C++, adding "signals" and "slots" to enable communication between objects. A sending object emits a signal when it changes in some interesting way and wants other objects to know about it. A slot is a function that can be used to receive signals. Qt's signals and slots mechanism is extremely flexible: a given signal can be sent to any number of slots, and a given slot can receive signals from any number of objects. Signals can even be connected to one another, so that the second signal is emitted when the first one is received. Qt's `connect()` function associates a given signal with a particular slot.

Note that slots are normal class member functions; they can be used for purposes other than just receiving signals.

1.Click the handset's "#" key to change the input mode. Note the input mode indicator in the upper left corner of the handset's display.

Using QT's signal/slot mechanism, you can tie a line edit widget to a base conversion function in the `FirstBaseUI` object like this:

```
connect(lineEditDec, SIGNAL(textChanged(const QString &)),
       this, SLOT(decValueChanged(const QString&)));
```

In plain English, the above translates as "Whenever the `lineEditDec` object's `textChanged()` function is called, please call my object's `decValueChanged()` function."

1 Add the following lines to the `FirstBaseUI` constructor (after the existing code):

```
connect(lineEditDec, SIGNAL(textChanged(const QString &)),
       this, SLOT(decValueChanged(const QString &)));
connect(lineEditHex, SIGNAL(textChanged(const QString &)),
       this, SLOT(hexValueChanged(const QString &)));
```

Note that in this version of the First Base application the binary field is read-only. Thus, there is no point in connecting that field to the conversion function.

The final step is to write the `decValueChanged()` and `hexValueChanged()` functions. Qt's `QString` class contains some useful functions that make base conversion a snap.

2 Add the following two new functions to `FirstBaseUI.cpp`:

```
void FirstBaseUI::decValueChanged(const QString &newValue)
{
  bool ok;
  int  num = newValue.toInt(&ok);

  if (ok)
  {
    lineEditHex->setText(QString::number(num, 16));
    lineEditBin->setText(QString::number(num, 2));
  }
  else
  {
    lineEditHex->setText("");
    lineEditBin->setText("");
  }
}

void FirstBaseUI::hexValueChanged(const QString &newValue)
{
  bool ok;
  int  num = newValue.toInt(&ok, 16);

  if (ok)
  {
    lineEditDec->setText(QString::number(num));
    lineEditBin->setText(QString::number(num, 2));
  }
  else
  {
    lineEditDec->setText("");
    lineEditBin->setText("");
  }
}
```

Note that the `toInt()` and `QString::number()` functions assume base ten conversions unless the base is specifically given as a parameter. The `toInt()` function helps guard against invalid entries by determining whether or not a given string represents a valid number in the specified base.

3 Add the function prototypes to the `FirstBaseUI.h` class definition, in the private slots section:

```
void decValueChanged(const QString &);
void hexValueChanged(const QString &);
```

4 Save your changes.

Now is another good time to explore the signals/slots mechanism and the base conversion by compiling and running the application. Note that as yet you are not prevented from entering invalid characters. Doing so is the subject of the next section.

Validate the Input

As you try out the First Base application, you've probably noticed a shortcoming: it works correctly if you enter decimal digits into the Decimal line-edit widget, but if you enter other characters, the values in the other line-edit widgets disappear. This is because the `QString` class's `toInt()` function detects when the string does not represent a valid number in the specified base, and returns an error code. Currently the application responds to an error by clearing the digits from the Hex and Binary line-edit widgets.

While this behavior is better than crashing, ideally the application should validate the characters as they are input by the user, preventing invalid characters from even being entered. Fortunately, Qt provides a simple solution: the `QValidator` base class. The following changes make use of this class to prevent invalid input:

1 In `FirstBaseUI.cpp`, add the following lines after the point where `lineEditDec` is initialized:

```
QValidator *valDec = new QIntValidator(0, 65535, lineEditDec);
lineEditDec->setValidator(valDec);
```

The first line instantiates a `QIntValidator` object and initializes it with three parameters: the minimum and maximum allowable integer values, and the parent of the validator. Specifying the `lineEditDec` widget as the parent of the validator not only binds the two together but also makes the validator subject to automatic memory management so that the validator is deallocated when the widget is.

The `setValidator()` call tells the validator to monitor the `lineEditDec` widget's input, filtering out any non-integer characters and restricting entry to the specified range.

2 Add the following #include statement near the top of the file:

```
#include <qvalidator.h>
```

3 Save your work, then build and test the project.

The Decimal field should now accept only decimal digits, and you are limited to entering integers in the range from 0 to 65535. Note that even when the input mode is set to alphabetic, only numbers can be entered.

The above takes care of the `lineEditDec` widget, but what about `lineEditHex`? If you are new to Qt development, you might assume that you can use a "QHexValidator", but you will find that such a class does not exist. If you are an experienced Qt developer, you would probably want to use

`QRegExpValidator`, but that particular class is not supported in MOTOMAGX. So how can you restrict entry of characters in the `lineEditHex` widget to only those that are valid for hexadecimal numbers? You create a custom validator which inherits from the `QValidator` base class. `QValidator` defines a virtual member function called `validate()` which must be implemented by every subclass. It should accept a string and return one of three values:

- `QValidator::Invalid` if the string is clearly invalid.
- `QValidator::Accept` if the string is clearly acceptable.
- `QValidator::Valid` if the current string is neither clearly invalid nor acceptable as a final result. This constant is unfortunately named[1] and represents an intermediate condition. In the First Base application, this condition should be returned when it detects that the input is a valid hex string but the value falls outside of the specified numeric range.

Follow these steps to create a new class named `HexValidator`:

1 In the Project Explorer, right-click on your project's `src` folder and select **New > Class**. Be sure that you right-click `src`: if you right-click a folder other than your project's source folder, the new class's files will be created in the wrong place in your project.

The New C++ Class dialog is displayed, as shown in Figure 2.

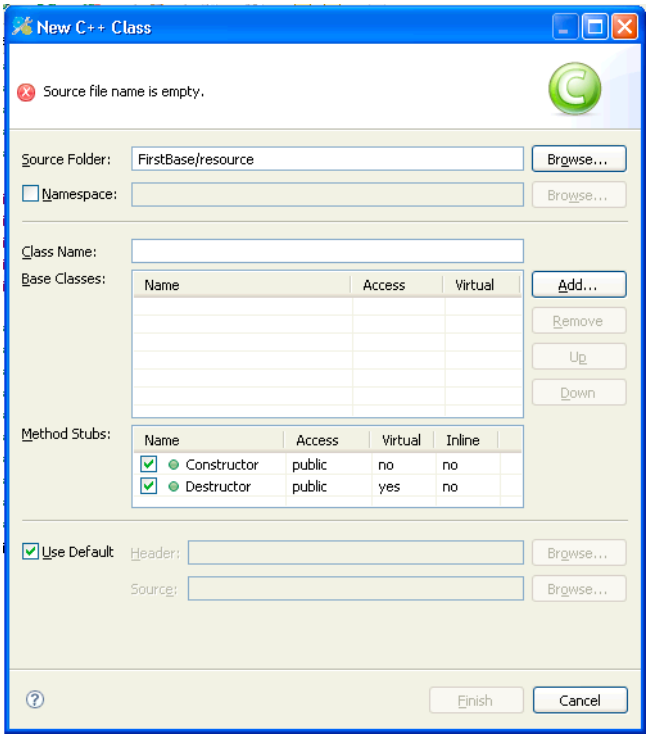


Figure 2: New C++ Class dialog

1.More recent versions of Qt rename `QValidator::Accept` as `QValidator::Acceptable` and rename `QValidator::Valid` as `QValidator::Intermediate`.

2 In the **Class Name** field type “HexValidator”.

3 Be sure that **Use Default** is selected. This causes both the header and implementation files to be created.

4 Click **Finish**.

MOTODEV Studio creates `HexValidator.h` and `HexValidator.cpp` files in your `src` folder, and opens them for editing.

5 Replace the class definition in `HexValidator.h` with the following. This defines a new constructor, a `validate()` function, and a couple of private data members to hold the acceptable range:

```
#include <qvalidator.h>

class HexValidator : public QValidator
{
  Q_OBJECT

public:
  HexValidator(QWidget *parent, const char *name = 0);
  HexValidator(int bottom, int top,
    QWidget *parent, const char *name = 0);
  virtual ~HexValidator();
  QValidator::State validate( QString &, int & ) const;

private:
  // Bottom and top of allowable range (decimal values)
  // If b == t, range checking is not performed.
  int b, t;
};
```

6 Implement the default constructor in `HexValidator.cpp`:

```
HexValidator::HexValidator(QWidget *parent, const char *name)
  : QValidator(parent, name)
{
  b = t = 0;
}
```

7 Implement the new constructor as follows. This constructor makes it easy to create a new validator and specify a valid range of numbers in one step, using the same signature as `QIntValidator` object’s constructor:

```
HexValidator::HexValidator(int bottom, int top,
  QWidget *parent, const char *name)
  : QValidator(parent, name)
{
  if (bottom > top) {
    // If bottom is incorrect, specify no range
    b = t = 0;
  } else {
    b = bottom;
    t = top;
  }
}
```

8 Implement the `validate()` function. It will be called by the system when the validator’s parent needs to validate user input.

```
QValidator::State HexValidator::validate(QString &str, int &unused) const
{
```

```
  bool ok;

  // Convert the string to a dec value.
  long int tmp = str.toLong(&ok, 16);

  if(str.length() == 0)// empty field; acceptable
    return QValidator::Acceptable;
  else if (!ok)// Didn't convert - invalid hex chars
    return QValidator::Invalid;
  else if ( (b != t) && (tmp > t || tmp < b) )
    return QValidator::Valid;
  else     // String is acceptable
    return QValidator::Acceptable;
}
```

9 Use the new `HexValidator` class to validate entries in the `lineEditHex` widget, by adding the following lines to the `FirstBaseUI` constructor after the place where you create and initialize the `lineEditHex` widget:

```
HexValidator *valHex = new HexValidator(0, 65535, lineEditHex);
lineEditHex->setValidator(valHex);
```

10 So that the above will work, add the following #include near the top of the file:

```
#include "HexValidator.h"
```

11 Save your work, then build and run your project. When typing values into the Hex widget, note that you can only enter valid hexadecimal characters,. As well, you'll find that the values you enter must be in the range [0 .. FFFF].

This completes the basic application. The next chapter shows how to modify the application so that it follows the Model-View-Controller design pattern.

Chapter 2: Converting to a Model-View-Controller Architecture

As it now stands, the `FirstBaseUI` class implements both the user interface and the application's processing logic. When you enter a new value into one of the `ZLineEdit` widgets, the widget emits a signal that is received by the other `ZLineEdit` widgets. Each widget, upon receiving the signal, converts the new value to the appropriate base and displays the result. For such a simple application, this combination of UI and processing logic is reasonable. As the complexity of the application grows, however, this mixed design runs the risk of becoming confusing to understand and difficult to maintain. For example, because the processing logic is intermingled with the UI code, changes to the UI might require you to hunt through your code for places where the processing logic must be altered as well.

A better design is one where the processing logic is factored into a class that "knows" nothing about the UI that implements it, and a UI class that holds none of the processing logic. In a classic Model-View-Controller (MVC) architecture, the "model" holds the processing logic. To modify First Base to follow this pattern, you need to create a new class that will represent the model: a class that knows how to convert bases but knows nothing about the user interface. With the addition of this one new class, you establish a clear division between the processing logic and the application's user interface. This not only allows you to more easily extend the user interface, should you so desire, but also gives you a model class that can be reused in other applications.

Qt's signal/slot mechanism really shines with a design such as this. A signal function can be written targeting a slot function without knowing if that function even exists, and the signal can be connected to multiple slots. Similarly, slot functions (which are normal member functions) do not know if any signals are connected to them, and can in fact have many signals connected. You can even connect a signal function directly to another signal function. This high level of encapsulation makes it simple to convert the First Base application into a more modular, maintainable MVC architecture.

The good news is that the application already contains most of the code you need: it just needs to be moved into the proper places. Begin by creating a new "BaseConverter" class to serve as the application's model:

1. In the Project Explorer, right-click on your project's `src` folder and choose **New > Class**.
2. In the **Class Name** field, enter "BaseConverter".
3. Verify that **Use Default** is selected, and then click **Finish** to create the new class.

The design of the model class is straightforward. `BaseConverter` should:

- Be told (via slots) when a new value needs converting.
- Convert the provided string to the appropriate base. In the event of an error, convert it to the empty string.
- Emit a signal with the new string to any implementers.

To keep things simple, the application will use a signal and a slot for each of the three bases it currently supports.

4 Edit `BaseConverter.h` so that, within the header guards, it looks like this:

```
#include <QObject.h>

class BaseConverter : public QObject
{
  Q_OBJECT

public:
  BaseConverter(QObject* = 0);

public slots:
  void setDecValue(const QString &);
  void setHexValue(const QString &);
  void setBinValue(const QString &);

signals:
  void decValueChanged(const QString &);
  void hexValueChanged(const QString &);
  void binValueChanged(const QString &);
};
```

Before implementing the model class, consider how it is going to be used by `FirstBaseUI`, which represents the view and controller portion of the new MVC design. The `ZLineEdit` widgets emit a `textChanged()` signal when their contents are modified; those signals should be connected to slots in the `BaseConverter` object. Like this, for example:

```
BaseConverter *bc = new BaseConverter(this);

connect(lineEditDec, SIGNAL(textChanged(const QString &)),
    bc, SLOT(setDecValue(const QString&)));
```

This means that the conversion logic should reside in the `setDecValue()` slot function. It will receive the candidate string and convert it, checking it for valid characters, just as currently happens in `FirstBaseUI::decValueChanged()`. But rather than directly call a line-edit widget's `setText()` function, it will emit a `decValueChanged()` signal. It is then up to the implementing UI class to connect the `decValueChanged()` signal to a line-edit widget's `setText()` slot. For the Decimal widget, the full code looks like this:

```
BaseConverter *bc = new BaseConverter(this);

connect(lineEditDec, SIGNAL(textChanged(const QString &)),
    bc, SLOT(setDecValue(const QString&)));
connect(bc, SIGNAL(decValueChanged(const QString&)),
    lineEditDec, SLOT(setText(const QString&)));
```

5 Add the following #include statement to `FirstBaseUI.cpp`:

```
#include "BaseConverter.h"
```

6 In the `FirstBaseUI` constructor, delete the existing `connect()` statements and replace them with:

```
BaseConverter *bc = new BaseConverter(this);

// When a lineEdit widget changes, tell the baseConverter model
connect(lineEditDec, SIGNAL(textChanged(const QString &)),
    bc, SLOT(setDecValue(const QString&)));
```

```
connect(lineEditHex, SIGNAL(textChanged(const QString &)),
    bc, SLOT(setHexValue(const QString&)));

// Have the converter model tell the UI about any changes
connect(bc, SIGNAL(decValueChanged(const QString&)),
    lineEditDec, SLOT(setText(const QString&)));
connect(bc, SIGNAL(hexValueChanged(const QString&)),
    lineEditHex, SLOT(setText(const QString&)));
connect(bc, SIGNAL(binValueChanged(const QString&)),
    lineEditBin, SLOT(setText(const QString&)));
```

7 Open `BaseConverter.cpp` and add the following code:

```
#include "BaseConverter.h"
#include <QString.h>

BaseConverter::BaseConverter(QObject *parent) : QObject(parent)
{
  // Nothing to implement
}

void BaseConverter::setDecValue(const QString &newValue)
{
  bool ok;
  int num = newValue.toInt(&ok);

  if (ok) {
    emit hexValueChanged(QString::number(num, 16).upper());
    emit binValueChanged(QString::number(num, 2));
  } else {
    emit hexValueChanged("");
    emit binValueChanged("");
  }
}

void BaseConverter::setHexValue(const QString &newValue)
{
  bool ok;
  int num = newValue.toInt(&ok, 16);

  if (ok) {
    emit decValueChanged(QString::number(num));
    emit binValueChanged(QString::number(num, 2));
  } else {
    emit decValueChanged("");
    emit binValueChanged("");
  }
}

void BaseConverter::setBinValue(const QString &newValue)
{
  bool ok;
  int num = newValue.toInt(&ok, 2);

  if (ok) {
    emit decValueChanged(QString::number(num));
    emit hexValueChanged(QString::number(num, 16).upper());
  } else {
    emit decValueChanged("");
    emit hexValueChanged("");
  }
}
```

8 Now, clean up the old UI member functions that are no longer used. In `FirstBaseUI.cpp` and `FirstBaseUI.h`, delete the `decValueChanged()` and `hexValueChanged()` member functions.

9 Save and compile your application. Then, run it to test the changes. It should behave exactly as it did at the end of the previous chapter.

The First Base application now uses the MVC architecture design pattern and will be much easier to expand and maintain.

Appendix A: First Base Source Code

The following sections contain the full source code of the completed First Base application.

BaseConverter.h

```
#ifndef BASECONVERTER_H_
#define BASECONVERTER_H_

#include <QObject.h>

class BaseConverter : public QObject
{
   Q_OBJECT

public:
   BaseConverter(QObject* = 0);

public slots:
   void setDecValue(const QString &);
   void setHexValue(const QString &);
   void setBinValue(const QString &);

signals:
   void decValueChanged(const QString &);
   void hexValueChanged(const QString &);
   void binValueChanged(const QString &);
};

#endif /*BASECONVERTER_H_*/
```

BaseConverter.cpp

```
#include "BaseConverter.h"
#include <QString.h>

BaseConverter::BaseConverter(QObject *parent) : QObject(parent)
{
   // Nothing to implement
}

void BaseConverter::setDecValue(const QString &newValue)
{
   bool ok;
   int num = newValue.toInt(&ok);

   if (ok) {
     emit hexValueChanged(QString::number(num, 16).upper());
     emit binValueChanged(QString::number(num, 2));
   } else {
     emit hexValueChanged("");
     emit binValueChanged("");
   }
}
```

```
void BaseConverter::setHexValue(const QString &newValue)
{
    bool ok;
    int num = newValue.toInt(&ok, 16);

    if (ok) {
      emit decValueChanged(QString::number(num));
      emit binValueChanged(QString::number(num, 2));
    } else {
      emit decValueChanged("");
      emit binValueChanged("");
    }
}

void BaseConverter::setBinValue(const QString &newValue)
{
    bool ok;
    int num = newValue.toInt(&ok, 2);

    if (ok) {
      emit decValueChanged(QString::number(num));
      emit hexValueChanged(QString::number(num, 16).upper());
    } else {
      emit decValueChanged("");
      emit hexValueChanged("");
    }
}
```

FirstBaseUI.h

```
/****************************************************************************
 *
 * © Motorola, Inc. 2008. All rights reserved.
 *
 ****************************************************************************/

// Guard macro to prevent multiple inclusion of header
#ifndef FIRSTBASEUI_H_
#define FIRSTBASEUI_H_

#include <ZKbMainWidget.h>
#include <ZLineEdit.h>

/****************************************************************************
 *
 * CLASS:   FirstBaseUI
 *
 * DESCRIPTION: This is an example of an application that uses a dialog as
 *       its main widget. It contains several other simple UI widgets
 *       and shows how to configure them.
 *
 ****************************************************************************/
class FirstBaseUI : public ZKbMainWidget {
    // Required Qt macro which provides signal/slot support. This
    // macro MUST be used when you define any objects which are
    // derived from ZWidget.
Q_OBJECT

public:
    FirstBaseUI(QWidget *parent=0, const char* name=0);
    ~FirstBaseUI();
```

```
private slots:
    void rightButtonClicked();
private:
   ZLineEdit *lineEditDec;
   ZLineEdit *lineEditHex;
   ZLineEdit *lineEditBin;
};

#endif
```

FirstBaseUI.cpp

```
/*************************************************************************
 *
 * © Motorola, Inc. 2008. All rights reserved.
 *
 *************************************************************************/

#include <ZFormContainer.h>
#include <ZSoftKey.h>
#include <ZAppInfoArea.h>
#include <qvalidator.h>
#include "FirstBaseUI.h"
#include "HexValidator.h"
#include "BaseConverter.h"

/*************************************************************************
 *
 * METHOD:   FirstBaseUI
 *
 * DESCRIPTION: A dialog widget which contains several sample interface
 *      widgets and initializes the softkeys.
 *
 * SUBCLASSES:ZKbMainWidget
 *
 *************************************************************************/
FirstBaseUI::FirstBaseUI(QWidget *parent, const char* name) :
    ZKbMainWidget(parent, name) {

   ZKbMainWidget::setAppTitle(name);

   ZAppInfoArea *aia = new ZAppInfoArea(this);
   this->setTitleBarWidget(aia);

   ZFormContainer *fc = new ZFormContainer(this);
   this->setContentWidget(fc);

   lineEditDec = new ZLineEdit(this);
   lineEditDec->setTitle("Decimal:");
   lineEditDec->setTitlePosition(ZLineEdit::TitleTop);
   lineEditDec->setMaxLength(5);
   QValidator *valDec = new QIntValidator(0, 65535, lineEditDec);
   lineEditDec->setValidator(valDec);

   lineEditHex = new ZLineEdit(this);
   lineEditHex->setTitle("Hex:");
   lineEditHex->setTitlePosition(ZLineEdit::TitleTop);
   lineEditHex->setMaxLength(4);
   HexValidator *valHex = new HexValidator(0, 65535, lineEditHex);
   lineEditHex->setValidator(valHex);

   lineEditBin = new ZLineEdit(this);
   lineEditBin->setTitle("Binary:");
```

```
    lineEditBin->setTitlePosition(ZLineEdit::TitleTop);
    lineEditBin->setMaxLength(16);
    lineEditBin->setReadOnly(true);

    fc->addChild(lineEditDec);
    fc->addChild(lineEditHex);
    fc->addChild(lineEditBin);

    ZSoftKey *pdlgSoftKey = this->getSoftKey();
    pdlgSoftKey->setText(ZSoftKey::RIGHT, "Exit");
    pdlgSoftKey->setClickedSlot(ZSoftKey::RIGHT,
         this, SLOT(rightButtonClicked()));
    pdlgSoftKey->disableClickedSlot(ZSoftKey::LEFT);
    pdlgSoftKey->setText(ZSoftKey::LEFT, "");

    BaseConverter *bc = new BaseConverter(this);

    // When a lineEdit widget changes, tell the baseConverter model
    connect(lineEditDec, SIGNAL(textChanged(const QString &)),
        bc, SLOT(setDecValue(const QString&)));
    connect(lineEditHex, SIGNAL(textChanged(const QString &)),
        bc, SLOT(setHexValue(const QString&)));

    // Have the converter model tell the UI about any changes
    connect(bc, SIGNAL(decValueChanged(const QString&)),
        lineEditDec, SLOT(setText(const QString&)));
    connect(bc, SIGNAL(hexValueChanged(const QString&)),
        lineEditHex, SLOT(setText(const QString&)));
    connect(bc, SIGNAL(binValueChanged(const QString&)),
        lineEditBin, SLOT(setText(const QString&)));
}

/*************************************************************************
 *
 * METHOD:   ~FirstBaseUI
 *
 * DESCRIPTION: Destructor - free allocated instance variable objects
 *
 *************************************************************************/
FirstBaseUI::~FirstBaseUI() {
    delete lineEditDec;
    delete lineEditHex;
    delete lineEditBin;
}

/*************************************************************************
 *
 * METHOD:   rightButtonClicked
 *
 * DESCRIPTION: Slot method. Tells app to exit when called.
 *
 * CALLED BY:Right softkey
 *
 *************************************************************************/
void FirstBaseUI::rightButtonClicked() {
    qApp->exit(0);
}
```

HexValidator.h

```
#ifndef HEXVALIDATOR_H_
#define HEXVALIDATOR_H_

#include <qvalidator.h>

class HexValidator : public QValidator
{
   Q_OBJECT

public:
   HexValidator(QWidget *parent, const char *name = 0);
   HexValidator(int bottom, int top,
     QWidget *parent, const char *name = 0);
   virtual ~HexValidator();
   QValidator::State validate( QString &, int & ) const;

private:
   // Bottom and top of allowable range (decimal values)
   // If b == t, range checking is not performed.
   int b, t;
};

#endif /*HEXVALIDATOR_H_*/
```

HexValidator.cpp

```
#include "HexValidator.h"

HexValidator::HexValidator(QWidget *parent, const char *name)
   : QValidator(parent, name)
{
   b = t = 0;
}

HexValidator::HexValidator(int bottom, int top,
   QWidget *parent, const char *name)
   : QValidator(parent, name)
{
   if (bottom > top) {
     // If bottom is incorrect, specify no range
     b = t = 0;
   } else {
     b = bottom;
     t = top;
   }
}

HexValidator::~HexValidator()
{
}

QValidator::State HexValidator::validate(QString &str, int &unused) const
{
   bool ok;

   long int tmp = str.toLong(&ok, 16);

   if(str.length() == 0)// empty field; acceptable
     return QValidator::Acceptable;
```

```
   else if (!ok)// Didn't convert - invalid hex chars
     return QValidator::Invalid;
   else if ( (b != t) && (tmp > t || tmp < b) )
     return QValidator::Valid;
   else// String is acceptable
     return QValidator::Acceptable;
}
```

main.cpp

```
/*************************************************************************
 *
 * © Motorola, Inc. 2008. All rights reserved.
 *
 *************************************************************************/

#include <ZApplication.h>
#include "FirstBaseUI.h"

/*************************************************************************
 *
 * METHOD:   main
 *
 * DESCRIPTION: Application entry point
 *
 *************************************************************************/
int main(int argc, char** argv) {
    // Create the application instance.
    // NOTE: You must create the ZApplication object before
    // you use any UI objects.
    ZApplication app(argc, argv);

    // Initialize the UI. Most of your implementation
    // code exists in this class.
    FirstBaseUI gui(NULL, "First Base");

    // Let the app know about your UI and prepare it
    // to be shown.
    app.setMainWidget(&gui);
    gui.show();

    // Give control to the app object (events, etc.)
    return app.exec();
}
```