2

Symbian OS User Interfaces

2.1 Introduction

UIQ and Series 60 are the only two user interfaces available for Symbian OS for which third-party developers can write C++ applications. Both provide a framework, built on top of Symbian OS, that can be reused by application writers, and a set of standard applications, for instance PIM applications, multimedia and email. The purpose of this chapter is to compare and contrast the two UIs, in particular their application framework APIs.

Series 60 is one of a range of developer platforms created by Nokia, and it has several versions. Versions 1.0, 1.1 and 1.2 (collectively referred to as the Series 60 Platform 1.x) are based on Symbian OS v6.1. Series 60 Platform 2.0, which this chapter describes, is based on Symbian OS v7.0s. The main features introduced in Platform 2.0 that affect application UIs are skins and bidirectional text support. Skins allow users to customize the UI, by changing the background bitmap, icons and color scheme. Skins are described later in this chapter. Bidirectional text support allows languages that are written from right to left, for instance Hebrew and Arabic, to be edited and displayed. It also affects the ordering and alignment of controls throughout the UI.

All Series 60 phones use a navigation controller that allows navigation in four directions, a confirmation key, and two hardware buttons, called softkeys, beneath the screen. These buttons make Series 60 phones easy to use with one hand. Users can input text using the phone's keypad and can optionally use a predictive text input system.

UIQ is produced by UIQ Technology AB, a subsidiary of Symbian Ltd. UIQ 1.0 was released in September 2000, and there have been several releases since. This chapter describes version 2.1, which runs on Symbian OS v7.0.

Symbian OS C++ for Mobile Phones, Volume 2. Edited by Richard Harrison
© 2004 Symbian Software Ltd ISBN: 0-470-87108-3

UIQ phones have a large touch-sensitive screen (either 6 × 8 cm or 4 × 6 cm), and use a pen as their main input device. However, like Series 60, all UIQ phones provide a hardware confirmation key and other hardware keys for navigation, minimally up and down, and optionally left and right, which make it possible to browse the phone's contents with one hand. The touch-sensitive screen allows text input methods like handwriting recognition and an on-screen virtual keyboard.

To a user these UIs have major differences, but to a programmer they have a lot in common. They have the same underlying framework, which means that application UIs written for both UIs have the same structure, based around application, document, app UI and view classes.

2.2 The Common Framework

Qikon and Avkon are the names of the UI-specific application framework layers implemented on top of the common Symbian OS UI framework, which is called Uikon. Qikon and Avkon implement framework classes defined in Uikon which must be overridden further by application writers. They also define many UI-specific controls, like dialogs, list boxes and editors that can be reused in applications.

Uikon provides the base classes for the three fundamental UI classes: the application (`CEikApplication`), the document (`CEikDocument`) and the application UI (`CEikAppUi`). All Series 60 and UIQ applications minimally need to define their own classes that derive from these. However, they must not derive directly from the Uikon base classes, but from the UI-specific implementations of them.

These implementations have the same name as the Uikon classes, but in UIQ the `CEik` prefix is replaced with `CQik`, and Series 60 uses `CAkn`:

	Application	**Document**	**App UI**
Symbian OS (Uikon)	`CEikApplication`	`CEikDocument`	`CEikAppUi`
Series 60 (Avkon)	`CAknApplication`	`CAknDocument`	`CAknAppUi/` `CAknViewAppUi`
UIQ (Qikon)	`CQikApplication`	`CQikDocument`	`CQikAppUi`

The convention of using a `Qik` or `Akn` prefix is used throughout UIQ and Series 60 to identify UI-specific classes, headers and libraries. In general, a type, header or library without such a prefix is part of Symbian OS, and is present in both UIs. There are some exceptions to this, notably `avkon`, which appears in a few filenames like `avkon.lib` which is the main Series 60 UI library, and `avkon.hrh` which holds the Series 60 resource constants.

With the exception of the app UI, there are very few differences between the Uikon, Qikon and Avkon implementations of these three classes. The next three subsections describe these differences.

2.2.1 The Application

This represents the application's properties, including its UID, capabilities and caption in an appropriate language. Minimally, it must implement two required functions: `CreateDocumentL()` to create the document, and `AppDllUid()` to return the application's UID.

2.2.1.1 `ini` Files

An `ini` file can be used by an application to store its initialization information. For instance, in UIQ it might store the current folder and the zoom level. If one exists, the `ini` file is opened and read each time the application is launched. To avoid the impact this has on application startup time, Series 60 by default does not support them.

The UI framework opens an application's `ini` file by calling `CEikApplication::OpenIniFileLC()`. `CAknApplication` overrides this function to leave with `KErrNotSupported`. If a Series 60 application needs to use an `ini` file, then it must implement `OpenIniFileLC()` itself.

2.2.1.2 Application Switching

In both UIQ and Series 60, applications can be left running in the background when a new application is launched. In UIQ this is enforced by the lack of a **Close** or **Exit** menu option in all applications. In Series 60 users have a choice: all applications should provide an **Exit** option, but users can switch between applications instead, for example by using the fast swap window, which is launched by holding down the menu key.

When an application is launched, `CEikApplication::PreDocConstructL()` is called by the UI framework to initialize the application. Both `CAknApplication` and `CQikApplication` override this to first check whether a new instance of the application really should be launched. If there is already a running instance, then instead of launching a new one, the existing instance is brought to the foreground.

2.2.2 The Document

This is responsible for storing and restoring the modifiable data that applies to a particular instance of the application. Minimally, it must instantiate the application UI by implementing a function called `CreateAppUiL()`. Because of this responsibility, all applications, even those that do not need to store any data, must create an instance of a document class.

2.2.2.1 *Responding to Low Memory*

Because UIQ applications cannot be closed down by users, it is up to the OS to free up RAM when a shortage occurs.

UIQ does this by calling the document's `SaveL()` function with a notification code of `MSaveObserver::EReleaseRAM`, for all running applications except the one in the foreground. `CQikDocument::SaveL()` saves the application's data, then sends an `EEikCmdExit` command to the app UI, which should respond by exiting the application.

An application's derived document class should override `SaveL()` if this default behavior is inappropriate. For instance, UIQ also calls `SaveL()` when data storage has run out, with an `MSaveObserver::EReleaseDisk` notification code. `CQikDocument::SaveL()` ignores these messages. You may choose to respond to this, for instance by deleting any data your application no longer needs or by canceling any operation that is filling up the disk.

2.2.2.2 *Document File Creation*

To further reduce application startup time, Series 60 by default disables document file creation and opening. In other words, `CAknDocument::OpenFileL()` returns NULL. `OpenFileL()` is called by the framework during application initialization, and returns the file store containing the application's document. Series 60 applications must override `CAknDocument::OpenFileL()` if they need to store and restore data.

2.2.3 The Application UI (app UI)

The main job of the app UI is to handle commands. Commands are generated from a variety of sources, for instance the softkeys and the **Options** menu in Series 60, and the menu bar and the toolbar in UIQ. The app UI also handles key, pointer and other events, and owns and manages the controls in the UI, including the views. Input handling is described in Section 2.7.

2.2.3.1 *Views*

In both UIQ and Series 60, many applications use more than one view to display their data in different ways. In multiple-view applications, the app UI is responsible for:

- creating and destroying views
- registering and deregistering views with the view server
- activating views

- adding views to the control stack and removing them (for views that need to handle input)
- setting a default view
- passing view-specific events to the correct view for handling.

There are several differences between the UIs in the way these tasks are carried out.

Views in UIQ are derived from `CCoeControl` and `MCoeView`. In multiple-view applications in Series 60, views are derived from `CAknView` and the app UI should be derived from `CAknViewAp-pUi` rather than from `CAknAppUi`. A `CAknView` is constructed from an `AVKON_VIEW` resource, and defines its own softkeys and **Options** menu pane.

In both UIs, view creation and registration are carried out in the derived app UI's `ConstructL()` function. In UIQ this function must also include a call to `CQikAppUi::BaseConstructL()` and in Series 60 to `CAknAppUi::BaseConstructL()`.`BaseConstructL()` reads the resource file and initializes various UI-specific elements of the app UI, for instance in Series 60 a key sounds object, which plays a sound when the app UI receives key events, and in UIQ the toolbar.

In UIQ, the derived app UI's `ConstructL()` should also include a call to `CCoeAppUi::RegisterViewL()` to register views (and there should be matching calls to `DeregisterView()` in its destructor). In Series 60, `CAknViewAppUi::AddViewL()` is called instead of `RegisterViewL()`.`AddViewL()` also causes the app UI to take ownership of the view; `CAknViewAppUi`'s destructor takes care of view deregistration and destruction.

In both UIs, if an application has multiple views, one of them should be set as the default view using `CCoeAppUi::SetDefaultViewL()`. This also should be called in the derived app UI's `ConstructL()`, because after calling `ConstructL()` the framework activates the default view, and if you haven't set one, the first registered view will be activated.

The default view is the view that is activated when the application is launched. In UIQ, it is also the view that is activated each time the application is brought to the foreground by being switched to. When an application is switched away from in UIQ, its active view is deactivated, and therefore it should save all outstanding changes (or discard any unimportant ones). The next time the application is brought to the foreground, the default view is restored.

In contrast, in Series 60, when an application goes into the background and is then brought to the foreground again, no view switch occurs. The application's previously active view remains active. Only if the application has no active view (this is the case when the application is launched) is the default view activated. Series 60 views therefore do not need to save their changes when the application is switched away from.

There is more information about views in Chapter 6.

2.2.3.2 Data Management

In both UIs it is possible in principle for applications to create multiple document files, although none of the built-in applications do this. In UIQ, categories (called **folders** in the UI) are the recommended way for users to manage data. Series 60 provides standard dialogs for file/folder and memory selection (`CAknFileSelectionDialog` and `CAknMemorySelectionDialog`) and for file creation (`CAknFileNamePromptDialog`) which can be used in applications. They can be defined using resource structs declared in `commondialogs.rh`.

A principle of UIQ is that applications, not users, are responsible for saving and managing data. Data management is done using categories. Each application can have its own set of categories. UIQ's standard applications use **Business**, **Personal** and **Unfiled** categories, but users can edit these names, add new ones or remove them. Categories are managed using the app UI's category model, `CQikCategoryModel`.

2.2.3.3 Access to UI Components

As well as handling commands, the app UI gives access to the UI components that generate commands.

`CQikAppUi::SetToolbarL()` and `RemoveToolbarL()` are used to add and remove the toolbar. `CAknAppUi::StatusPane()` gives access to the Series 60 status pane, and `CAknAppUi`'s `Cba()` function (CBA is short for control button array) gives access to the button group container that holds the left and right softkeys.

Toolbars, status panes and softkeys are described in more detail later in this chapter.

2.2.3.4 Zooming

In UIQ, **Zoom** is recommended as a standard **Edit** menu option for views that allow text input. When **Zoom** is selected, the standard UIQ zoom settings dialog should be launched by calling `CQikZoomDialog::RunDlgLD()`. This supports small, medium and large zoom levels.

Each zoom level corresponds to a zoom factor, which is the factor by which font sizes are reduced or increased. A 100% magnification has a zoom factor value of 1000, 200% has the value 2000 and so on. An application's three zoom factor values may be persisted using an `ini` file called `zoom.ini`, located in the same folder as the application. If no such file exists, the zoom factor is read from a resource, which contains default zoom factors.

`CQikAppUi::ZoomFactorL()` gets the application's zoom factors.

In Series 60 it is up to the application whether or not to support zooming. A zoom option is not a requirement and the app UI does not provide any special support for it.

2.2.3.5 `SetMopParent()`

App UIs and views in Series 60 should call a function called `SetMopParent()` on any controls they own. `SetMopParent()` is defined in `CCoeControl` so is part of the core OS, but because it was added in v7.0s it is unavailable in UIQ. Its purpose is to inform controls of their owning component. Not calling it may cause scroll indicators, skins and other features to work incorrectly.

2.3 The Screen Layout

Series 60 and UIQ each support a wide variety of devices. Phone manufacturers can customize the UI's look and feel, for instance the icons, color schemes, fonts and text, but all phones running on the same platform have the same basic screen layout and UI components.

2.3.1 Customizing the Look and Feel

Symbian OS includes an abstract look and feel layer that is implemented by the UI in a component called Uiklaf. This component is used by Uikon to get information about the UI. For instance, Uiklaf specifies the UI's standard fonts and the appearance of borders around controls. Also, in some phones, applications briefly fade just before moving into the background. This behavior is specified in Uiklaf.

All Uiklaf classes have a `Laf` prefix. They are only intended to be used internally by Uikon.

As an aside, fading is used in other situations. For instance in both UIs, the foreground application is faded when a dialog is displayed, and in Series 60 only, it is faded when a menu pane is displayed.

2.3.2 Series 60

In Series 60, the screen is divided into three areas or panes (Figure 2.1). These are referred to as the status pane, main pane, and control (or softkey) pane.

2.3.2.1 *Status Pane*

The status pane, usually displayed at the top of the screen, displays information about the foreground application including its title and

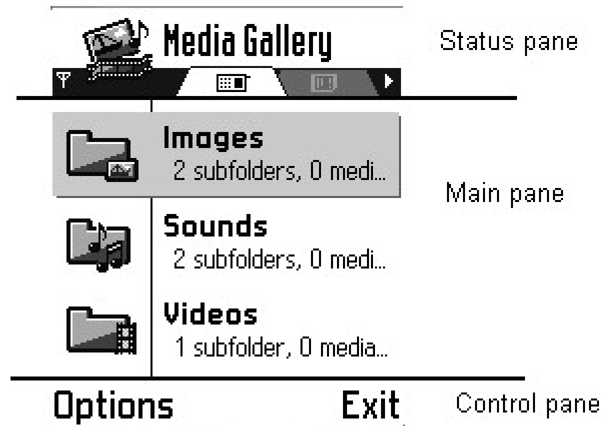

Figure 2.1 Series 60 screen layout

icon, as well as general information about the phone, for instance the signal strength.

It consists of six subpanes. These are laid out from left to right (or from right to left in some cases, for instance in Arabic and Hebrew layouts, in which the ordering of many controls is reversed):

- signal pane (signal strength indicator)
- context pane (displays the application's icon)
- title pane (displays the application's title)
- navigation pane (may contain tabs, images, or text, or can be empty)
- battery pane (battery strength indicator)
- small indicator pane (contains connectivity and some other indicators).

Of these, only the title pane, context pane and navigation pane can be customized by an application writer.

By default, the navigation pane is empty, but it can contain a tab group, as shown in Figure 2.1. This indicates the currently active view in a multiple view application, or the current page in a multi-page dialog and whether more views or pages exist.

In single view applications, the navigation pane can contain a label or image instead of a tab group. This may also be useful when the item in the view is part of a sequence. In this case, left and right arrow bitmaps can show that there is a next or previous item. For instance, the Calendar application displays the date in a navigation label and uses this for navigation.

When the application is in a text editing state, the navigation pane automatically displays an edit indicator which shows whether the input

mode is numeric or alphabetic, and upper or lower case (phones that use non-Western languages have different input modes).

The application's initial status pane is specified in the application's EIK_APP_INFO resource that is required in all applications' resource files.

A status pane is defined by a STATUS_PANE_APP_MODEL resource and its subpanes by SPANE_PANEs. The following types of subpane resource exist:

- TITLE_PANE. This can optionally contain an icon instead of text, but not both. The default is the application's caption read from its aif file.
- CONTEXT_PANE. This contains an icon, representing the application. This icon is displayed alongside the title. The default is the application's icon read from its aif file.
- NAVI_DECORATOR. This can contain different types of controls, including a tab group (TAB_GROUP), a text label (NAVI_LABEL), or an image (NAVI_IMAGE).

In some situations, you may want to hide the status pane, for instance in games that require a full screen display. You can access it using CAknAppUi::StatusPane(), and it can be hidden or exposed by calling CEikStatusPane::MakeVisible().

2.3.2.2 Control Pane

The control pane contains the softkey labels and the scroll indicator, if required. A softkey label is a string, usually a single short word identifying the action associated with the softkey. The labels change depending on the state of the application. Often, the left-hand softkey is labelled **Options** and activates a menu pane. It can also be used for issuing affirmative commands, for instance **Ok, Select** and **Yes**, while the right-hand softkey is used for **No, Back, Cancel** and **Exit** commands.

The two softkeys are defined using a CBA resource struct containing an array of two CBA_BUTTONs which define the left- and right-hand softkeys. These structs are declared in uikon.rh.

Series 60 declares many standard CBA resources in avkon.rsg, with names beginning with R_AVKON_SOFTKEYS. You can reuse these or define your own.

For example, here is Series 60's resource definition for **Yes** and **No** softkeys:

```
RESOURCE CBA r_avkon_softkeys_yes_no
    {
    buttons =
```

```
        {
        CBA_BUTTON {id=EAknSoftkeyYes; txt=text_softkey_yes;},
        CBA_BUTTON {id=EAknSoftkeyNo; txt=text_softkey_no;}
        } ;
}
```

The `txt` and `id` fields are used to define the softkey label and the command ID it invokes. Note that the **Yes**/**No** text is defined in a separate file containing the localized text strings.

The command IDs issued by the softkeys are defined in `avkon.hrh`. Some common ones are:

- `EAknSoftkeyOk`
- `EAknSoftkeyCancel`
- `EAknSoftkeySelect`
- `EAknSoftkeyOptions`
- `EAknSoftkeyBack`
- `EAknSoftkeyYes`
- `EAknSoftkeyNo`
- `EAknSoftkeyDone`
- `EAknSoftkeyClose`
- `EAknSoftkeyExit`.

Softkeys can be defined in several places:

- the `cba` field in an `EIK_APP_INFO` resource
- the `buttons` field in a `DIALOG` resource
- the `cba` field in an `AVKON_VIEW` resource
- the `softkeys` field in an `AVKON_LIST_QUERY` resource
- the `softkeys` field in an `AVKON_MULTISELECTION_LIST_QUERY` resource.

The application's initial softkeys are specified in the `EIK_APP_INFO` resource; often, these are **Options** and **Exit** (`R_AVKON_SOFTKEYS_OPTIONS_EXIT`). The **Options** menu pane is defined by `EIK_APP_INFO`'s `menubar` field.

2.3.2.3 Main Pane

This is the area between the status pane and the control pane and is generally the area available for the application to draw to – it is the area returned by the app UI's `ClientRect()` function.

2.3.3 UIQ

The screen in UIQ is divided into five areas (Figure 2.2), one of which, the toolbar, is optional.

2.3.3.1 Application Picker

This is used to switch between applications. The rightmost icon switches to the application launcher, which gives access to all installed applications. Users can change the applications that are displayed in the Application picker.

2.3.3.2 Menu Bar

UIQ has no control pane or softkeys. Instead it uses a menu bar that is always visible beneath the application picker. Each view typically has its own menu bar. Most UIQ menu bars have two left-aligned menu titles and a right-aligned **Folders** menu. The leftmost title should be the name of the application. This provides the principal menu pane with standard functions like **New**, **Find** and **Send as**. The next menu title to the right, which can be omitted if not required, normally provides standard **Edit**

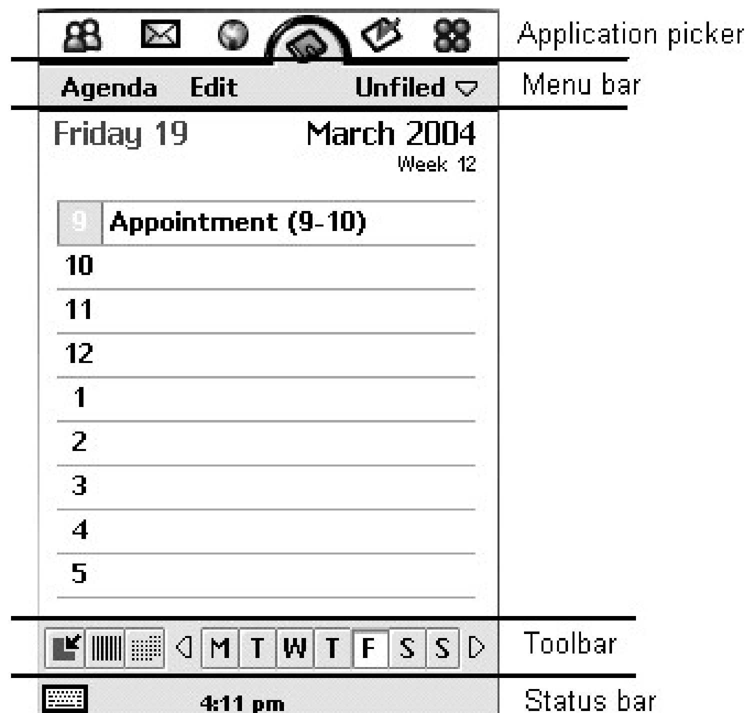

Figure 2.2 UIQ screen layout

commands like **Cut**, **Copy**, **Paste** and **Zoom**. UIQ menu panes should be kept short and cascading menu panes are deprecated. If the menu pane contains more items than will fit on the screen, a scroll bar is added to it, although this should be avoided.

Note that UIQ menus do not include an **Exit** or **Close** option, except in debug builds, where it can be useful to check that the application frees all resources when closed. They also should not provide a **Save** option; UIQ applications should save their data without user intervention.

For more guidelines on how to design menus, see the UIQ Style Guide in the UIQ SDK documentation.

2.3.3.3 Toolbar

The toolbar is an optional view-specific bar at the bottom of the screen that holds frequently used controls. These are often command buttons, so an alternative name for it is the button bar.

The application's initial toolbar is specified in the `toolbar` field in the application's `EIK_APP_INFO` resource. It can be changed using `CQikAppUi::SetToolbarL()`, or removed using `CQikAppUi::RemoveToolbarL()`.

The toolbar is defined in a resource file using a `QIK_TOOLBAR` resource, and toolbar buttons by `QIK_TBAR_BUTTON`s. By default these are command buttons, but this can be changed using `QIK_TBAR_BUTTON`'s `type` field. You can customize their behavior, alignment, and spacing (although these have default values) and they can contain text or a bitmap or both. The following code defines a toolbar button containing a standard 'Go back' bitmap (Figure 2.3):

```
QIK_TBAR_BUTTON
     {
     id=EMyAppDone;
     flags=EEikToolBarCtrlHasSetMinLength;
     alignment=EQikToolbarRight;
     bmpfile="z:\\System\\Data\\quartz.mbm";
     bmpid=EMbmQuartzBackarrow;
     bmpmask=EMbmQuartzBackarrowmask;
     }
```

Controls other than buttons can be used in the toolbar by using `QIK_TBAR_CTRL` resources instead.

Note that an optional UI component called the tab screen can be displayed in the same area as the toolbar, but is defined separately. This is described in Section 2.4.7.

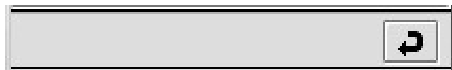

Figure 2.3 UIQ toolbar with standard Go back button

2.3.3.4 Status Bar

In general the purpose of the status bar is to display status information such as the battery level and signal strength, and to hold the button that activates the virtual keyboard. Its contents are defined by the phone manufacturer; it cannot be modified by third-party developers.

2.4 Common UI Components

This section describes how UI components that exist in both UIs are defined, created and used, including the differences between the two implementations.

2.4.1 Menu Bars

Menus are defined in resource files, and consist of the following four components:

- The menu bar, which in UIQ is a horizontal bar containing the menu titles. In Series 60 it is never displayed. It is defined using a `MENU_BAR` resource (`CEikMenuBar` in C++).
- Menu titles. These specify a menu pane and in UIQ the text to display in the menu bar. They are defined by `MENU_TITLE` resources. Note that in Series 60, the labels that are displayed in the control pane are not defined in the `MENU_TITLE` resource but in the `CBA` resource that defines the softkeys.
- Menu panes, which are vertical lists of menu items displayed when the user selects a menu title in UIQ or the **Options** softkey in Series 60. They are defined by `MENU_PANE` resources (`CEikMenuPane` in C++). Both Series 60 and UIQ support cascading menu panes, although they are deprecated in UIQ.
- Menu items, which are items in the menu pane that may be selected by the user. They are defined by `MENU_ITEM` resources, and are associated with a label that appears in the menu pane and a command ID that is issued when the item is selected.

The menu bar in UIQ is always visible. It supports multiple menu titles, each of which is associated with a menu pane. In Series 60, the control pane is used instead of a menu bar. Nevertheless, a menu bar still needs to be specified in most Series 60 applications, because it defines the menu pane that is activated by the **Options** softkey.

In both Series 60 and UIQ, the initial menu bar that is used when the application is launched is specified in the `menubar` field in the `EIK_APP_INFO` resource struct.

Different views typically need different menu bars, so that when the view changes, so should the menu bar. In UIQ, menu bar switching is done in the view's ViewActivatedL() function using code like this:

```
MEikAppUiFactory* factory = iEikonEnv->AppUiFactory();
factory->MenuBar()->ChangeMenuBarL(0, R_NEW_MENUBAR, EFalse);
```

The second parameter of CEikMenuBar::ChangeMenuBarL() is the resource ID of the new view's menu bar. The other two parameters are not used in UIQ.

In Series 60, if you need to dynamically change the menu bar you can do this using code like:

```
MEikAppUiFactory* factory = iEikonEnv->AppUiFactory();
factory->MenuBar()->SetMenuTitleResourceId(R_NEW_MENUBAR);
```

Unlike UIQ, Series 60 does not implement CEikMenuBar::ChangeMenuBarL(); you must use SetMenuTitleResourceId() instead.

Normally, however, Series 60 uses a different scheme for switching between view-specific menu bars. Series 60 views are defined by AVKON_VIEW resources and the menu bar and softkeys associated with the view are specified in the resource:

```
STRUCT AVKON_VIEW
    {
    LLINK hotkeys=0;
    LLINK menubar=0;
    LLINK cba=0;
    }
```

In the view's ConstructL(), you call CAknView::BaseConstructL(), passing it the ID of an AVKON_VIEW resource. When the view is activated, its menu bar and softkeys are automatically used.

In Series 60, dialogs can define their own menu bar. The Series 60 dialog base class, CAknDialog, owns a menu bar, and the menu bar's resource ID is specified when you construct the dialog. The softkeys used by the dialog are defined in the DIALOG resource's buttons field. When the dialog is launched, its softkey labels overwrite any labels previously displayed and its menu bar is added to the control stack so that it receives key events before any existing menu bar. Note that Series 60 dialogs that do not need an **Options** menu pane can be derived from CEikDialog instead.

A form (CAknForm) is a type of dialog derived from CAknDialog. Forms provide their own default menu bar and menu pane. List query dialogs (CAknListQueryDialog) are also derived from CAknDialog.

They are defined by an AVKON_LIST_QUERY resource, which uses a field called softkeys rather than buttons to define its softkeys.

Dialogs and forms are described later in this chapter.

2.4.2 Menu Panes

The contents of menu panes may also need to change according to the state of the application. The function to make menu items available or unavailable is CEikMenuPane::SetItemDimmed(). In UIQ, calling this function with ETrue causes the item to remain in the menu, but it appears dimmed. In Series 60, to save screen space and make menus quicker to navigate, the menu item is removed. Calling the function again with EFalse undims it or causes it to reappear. CEikMenu-Pane::DeleteMenuItem(), on the other hand, removes a menu item altogether, and CEikMenuPane::AddMenuItemL() adds one. Both of these functions have variants that allow multiple menu items to be added or deleted at once. Note that UIQ deprecates dynamically adding and removing menu items because this can confuse users; you should dim and undim them instead. If the user selects a dimmed menu item, an infoprint explaining why the item is unavailable may optionally be used.

Menu items are added, deleted, dimmed and undimmed in the app UI's implementation of DynInitMenuPaneL(). This function is called by the framework just before the menu pane is displayed. For Series 60's menu bar-owning dialogs, DynInitMenuPaneL() is implemented by the dialog rather than the app UI. CAknDialog defines a DynInit-MenuPaneL() function which is empty, but CAknForm overrides it to dim any items that are not required.

Related items can be grouped in a menu pane using separators. This is done in the menu pane's resource definition by specifying

```
flags=EEikMenuItemSeparatorAfter;
```

for the MENU_ITEM preceding the separator. Both Series 60 and UIQ support this flag. EEikMenuItemSeparatorAfter is defined with other menu item flags in uikon.hrh.

2.4.2.1 Cascading Menu Panes

These are panes within another pane. Figure 2.4 shows an example.

A cascading menu pane is defined by a MENU_ITEM's cascade field. As they are considered to add complexity, UIQ deprecates them.

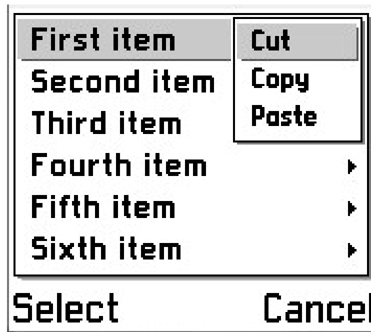

Figure 2.4 Cascading menu pane in Series 60

2.4.3 Dialogs

In both UIs, dialogs appear at the bottom of the screen. They always fill the width of the screen, and have variable height, although in UIQ they never cover the application picker or the status bar. Unlike UIQ, Series 60 does not use dialog buttons; the softkeys are used instead.

All dialogs are ultimately derived from class `CEikDialog` and are usually defined by a `DIALOG` resource, or sometimes by a `FORM` resource in Series 60. The most important fields in a `DIALOG` resource are the `flags`, the `buttons` and the `items`. The `items` in a dialog (or a Series 60 form) are defined by `DLG_LINE` resources. Each dialog line can contain one of a variety of controls. The available controls vary between the two UIs.

The pattern of use for a dialog is the same in both UIs. You construct it using a C++ constructor, call the dialog's `ExecuteLD()` function, specifying the ID of the `DIALOG` resource that defines it, then test `ExecuteLD()`'s return value, which indicates which button or softkey was pressed to dismiss it. For convenience, some dialogs wrap up construction and execution in a static function called `RunDlgLD()`.

2.4.3.1 Title Bar

Dialogs in UIQ, unlike those in Series 60, have a title bar which serves several purposes:

- It holds the dialog's title (defined by `DIALOG` struct's `title` field).
- It can be dragged, so the user can see what is underneath the dialog.

Figure 2.5 A UIQ dialog with a help button, showing a drop-down edit menu

- If the dialog contains a text editor, UIQ adds a button that when pressed activates a drop-down menu pane providing cut, copy and paste functions (see Figure 2.5).
- It can optionally contain a help button which, when pressed, launches a context-specific help topic.

The help button is added if the dialog provides a help context. It can do this either by calling `CEikDialog::SetHelpContext()` or by overriding `CEikDialog::GetHelpContext()`. The UIQ SDK documentation provides more information on the context-sensitive help system.

2.4.3.2 Dialog Flags

Uikon's dialog flags have names beginning with `EEikDialogFlag` and are defined in `uikon.hrh`. They control the behavior and layout of dialogs, but note that not all of them are relevant to both UIs. A useful flag for Series 60 is `EEikDialogFlagFillAppClientRect` which causes the dialog to fill the main pane. In UIQ, the `EEikDialogFlagWait` flag makes the dialog modal; in other words, pen taps outside the dialog are ignored.

Series 60 defines standard sets of flags suitable for Series 60-specific dialogs, in `avkon.hrh`. For instance `EAknErrorNoteFlags` is used for error notes.

2.4.3.3 Dialog Buttons

UIQ supports multiple buttons inside a dialog. They are usually defined using a `DLG_BUTTONS` resource, which is an array of `DLG_BUTTON` resources, or you can use a standard UIQ button resource, for instance `R_EIK_BUTTONS_CANCEL_OK`, defined in `eikcore.rsg` for **Cancel** and **Done** buttons.

Dialog buttons in UIQ are arranged by default in a single horizontal row, although it is possible to use two rows or to arrange the buttons

vertically. One button should be set as the default. This is the button that is activated by the **Confirm** hardware key and is drawn with a highlight. It is normally the button labelled **Done**, or equivalent. A button is set as the default by specifying the `EEikLabeledButtonIsDefault` flag in its `DLG_BUTTON` resource definition.

Series 60 does not use `DLG_BUTTON` resources and has no default button. The softkeys are used instead.

2.4.3.4 Dialog Lines

In both UIs, the main body of a dialog is defined by an array of `DLG_LINE` resources. The most important fields in a `DLG_LINE` resource are the `type`, the `id`, the `prompt` and the `control`.

The type of the control in the dialog line must always be specified, because both UIs have a control factory that uses it to construct the correct C++ class.

- Common control types have an `EEikCt` prefix and are defined in the `TEikStockControls` enumeration in `uikon.hrh`. Note that not all of these controls are supported by both UIs.
- Series 60-specific control types have an `EAknCt` prefix and are defined in `avkon.hrh`.
- UIQ-specific control types have an `EQikCt` prefix and are defined in the `TQikStockControls` enumeration in `qikstockcontrols.hrh`.

The `id` uniquely identifies the control in the program code and also needs to be specified. If you need to get pointers to the dialog's controls in your application code, you can do this using `CEikDialog::Control()` which takes the control's `id` as its parameter. You need to cast the return value to the correct type. The `id` should be defined in the `.hrh` file so that it can be used in both the C++ code and the resource file.

Controls in dialogs can have a prompt, which is a caption displayed beside the control. If the prompt and the control combined are too wide for the dialog, UIQ breaks the caption into multiple lines, while Series 60 truncates it. In UIQ, you can alternatively place the caption above the control by specifying `EQikDlgItemCaptionAboveControl` in the dialog line's `itemflags` field.

UIQ controls can optionally have a trailer (using the `DLG_LINE`'s `trailer` field). This is an additional caption in bold, displayed after the control, which is sometimes used to hold units of measurement. This is not supported in Series 60.

Series 60 has support for bitmaps in dialog lines. Use the `DLG_LINE`s' `bmpfile`, `bmpid` and `bmpmask` fields to specify the `mbm` file, bitmap and mask. If specified, the bitmap is displayed beside the prompt.

2.4.3.5 *Dialog Pages*

Both UIs support multi-page dialogs. In Series 60, tabs in the navigation pane identify the current page, and the navigation controller is used to switch between them. Series 60 always displays two tabs for multi-page dialogs (Figure 2.6), even if the dialog has more than two pages.

Note that in Series 60, tabs in the navigation pane are also used for view switching in applications with multiple views. In this case, the navigation pane can contain up to four tabs. Because dialog pages are navigated in the same way as application views, when porting a multiple-view application from UIQ to Series 60 it may be possible to redesign it to use a dialog-based architecture, where each dialog page represents a view.

In UIQ, the dialog page tabs are displayed at the bottom of the page, just above the dialog buttons. UIQ displays as many tabs as will fit on the screen. As in Series 60, left and right navigation arrows are automatically added to the ends if not all the tabs will fit (see Figure 2.7).

Dialog pages are defined using the `DIALOG` struct's `pages` field. This is assigned an array of `PAGE` resources. `PAGE` resource structs exist in both UIQ and Series 60, but in Series 60 the page tab can contain a bitmap instead of text and the page can contain a form.

Figure 2.6 Navigation tabs for a multi-page dialog in Series 60

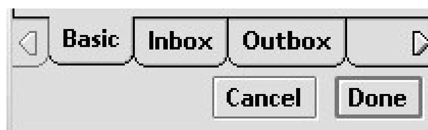

Figure 2.7 Navigation tabs for a multi-page dialog in UIQ

2.4.3.6 *UIQ-specific Dialogs*

UIQ implements a set of standard dialogs that can be reused in third-party applications. For convenience all of these dialogs define a static function called `RunDlgLD()` that wraps up construction and execution.

Name	Class	Link against
Add/delete/edit folders	CQikEditCategoriesDialog	qikdlg.lib
Find text	CEikEdwinFindDialog	eikcdlg.lib
Set zoom level	CQikZoomDialog	qikdlg.lib
Send as	CQikSendAsDialog	qikdlg.lib
Set password	CEikSetPasswordDialog	eikcdlg.lib
Verify password	CEikVerifyPasswordDialog	eikcdlg.lib
Set location	CEikTimeDialogSetCity	eikcdlg.lib
Set date and time	CEikTimeDialogSetTime	eikcdlg.lib
Set summer times	CEikTimeDialogSetDst	eikcdlg.lib
Set date and time formats	CEikTimeDialogOptionFormat	eikcdlg.lib
Set workdays	CEikTimeDialogOptionWorkday	eikcdlg.lib
Info window	Use iEikonEnv->InfoWinL()	eikcore.lib
Alert window	Use iEikonEnv->AlertWin()	eikcore.lib
Query window	Use iEikonEnv->QueryWinL()	eikcore.lib

2.4.3.7 Series 60-specific Dialogs

Series 60 extends the Uikon dialog hierarchy with its own dialog types, notably CAknDialog, which adds support for a menu bar, and its derived classes. These include:

- query dialogs (CAknQueryDialog and several derived classes)
- selection and markable list dialogs (CAknSelectionListDialog and CAknMarkableListDialog)
- forms (CAknForm).

Series 60 also implements some dialogs that do not need a menu bar, so are derived from CEikDialog instead:

- note dialogs (CAknNoteDialog and derived classes) and associated wrappers.

Query Dialogs

Query dialogs are used to request confirmation or other input from the user. There are various types, all derived from CAknQueryDialog. The main ones are:

- Confirmation queries. These request user confirmation, so often the dialog uses **Yes** and **No** softkeys.
- Data queries. These use an editor, for instance an EDWIN for text input, or a NUMBER_EDITOR for numeric input. There are several

Figure 2.8 A message query dialog

subtypes of data query. They often use **Ok** and **Cancel** buttons, and have their own edit indicator to show what kind of input is accepted.

- List queries. These display a popup single or multiple selection list, and support a wide range of list box types, for instance single or double line, with or without icons.
- Message queries. These display a message and optionally have a title. Figure 2.8 shows an example.

This table lists the classes and resources associated with query dialogs.

Name	**Class**	**Resource struct**
Confirmation query	`CAknQueryDialog`	`AVKON_CONFIRMATION_QUERY`
Text query	`CAknTextQueryDialog`	`AVKON_DATA_QUERY`
Integer query	`CAknNumberQueryDialog`	`AVKON_DATA_QUERY`
Date or time query	`CAknTimeQueryDialog`	`AVKON_DATA_QUERY`
Duration query	`CAknDurationQueryDialog`	`AVKON_DATA_QUERY`
Floating point query	`CAknFloatingPointQueryDialog`	`AVKON_DATA_QUERY`
Multiple line query	`CAknMultiLineDataQueryDialog`	`AVKON_DATA_QUERY`
Single selection list query	`CAknListQueryDialog`	`AVKON_LIST_QUERY`
Multiple selection list query	`CAknListQueryDialog`	`AVKON_MULTISELECTION_LIST_QUERY`
Message query	`CAknMessageQueryDialog`	`AVKON_MESSAGE_QUERY`

Notes:

- The control type for all query dialogs is `EAknCtQuery`.
- The confirmation and data query resource structs have a `layout` field. Its possible values are defined in `avkon.hrh`, for instance `ENumber-Layout` for integer queries and `EConfirmationQueryLayout` for confirmation queries.

- Single selection list query dialogs are constructed with an integer pointer. If the dialog's `ExecuteLD()` returns true, this pointer holds the index of the selected item. Multiple selection list query dialogs use a pointer to an integer array instead.
- Query dialogs can play a tone when they execute. The tone type is specified when the dialog is constructed.
- `CAknQueryDialog::SetEmergencyCallSupport()` enables emergency telephone calls to be made even when a numeric query dialog is executing. By default, this is disabled.
- Text query dialogs may use predictive text entry by calling `CAkn-QueryDialog::SetPredictiveTextInputPermitted()`. The default is not to use it.

List query dialogs support the following types of list box:

Name	Class	Control type
Single line list box	`CAknSinglePopupMenuStyle ListBox`	`EAknCtSinglePopupMenu ListBox`
Single line list box with icons	`CAknSingleGraphicPopupMenu StyleListBox`	`EAknCtSingleGraphicPopupMenu ListBox`
Single line list box with titles	`CAknSingleHeadingPopupMenu StyleListBox`	`EAknCtSingleHeadingPopup MenuListBox`
Single line list box with icons and titles	`CAknSingleGraphicHeading PopupMenuStyleListBox`	`EAknCtSingleGraphicHeading PopupMenuListBox`
Double line list box	`CAknDoublePopupMenuStyle ListBox`	`EAknCtMenuDoublePopupMenu ListBox`
Double line list box with icons	`CAknDoubleLargeGraphic PopupMenuStyleListBox`	`EAknCtDoubleLargeGraphic PopupMenuListBox`

Selection and Markable List Dialogs

`CAknSelectionListDialog` and a derived class, `CAknMarkable-ListDialog`, are convenient list box dialogs with an optional search box (permanent or popup) and a predefined menu bar. They occupy the whole of the main pane (see Figure 2.9).

They are defined using a `DIALOG` resource with a `LISTBOX` control in the first dialog line and, optionally, a search box in a second line. The

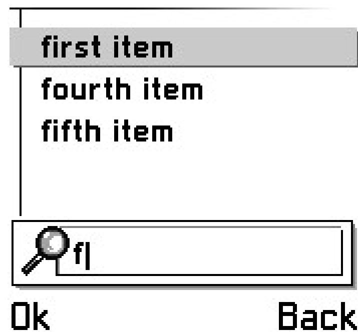

Figure 2.9 Selection list dialog with search box

LISTBOX can be of any single or double list box type: the possible values are defined in avkon.hrh, starting with EAknCtSingleListBox.

The dialog needs to define appropriate softkeys and a menu pane. For instance, the markable list dialog uses an **Options** rather than **Ok** softkey, with **Mark** and **Mark all** menu options. The resource ID of the menu bar is specified when the dialog is constructed. Standard menu panes are:

- R_AVKON_MENUPANE_SELECTION_LIST
- R_AVKON_MENUPANE_SELECTION_LIST_WITH_FIND_POPUP
- R_AVKON_MENUPANE_MARKABLE_LIST
- R_AVKON_MENUPANE_MARKABLE_LIST_WITH_FIND_POPUP.

The comments in header file AknSelectionList.h describe how to define and use these dialogs.

Note Dialogs and Wrappers

Note dialogs display a warning, a question or a progress indicator. They do not need a menu bar and most are timed so do not even need softkeys. Because of this, the note dialog base class, CAknNoteDialog, is derived from CEikDialog rather than CAknDialog.

You can define your own notes using an AVKON_NOTE resource and assigning it to the control field in the dialog's DLG_LINE, but because note dialogs are so commonly used, note wrappers, which use predefined resources, are provided.

Wrappers make note dialogs very easy to use by providing a default resource definition with values for the duration, tone, animation and softkeys. The confirmation note wrapper provides a default label (**Done** or equivalent), but for the others you specify the label when calling the dialog's ExecuteLD().

The following table lists all of the note dialogs and their associated wrapper classes.

Name	Dialog class	Wrapper class
Confirmation note	CAknNoteDialog	CAknConfirmationNote
Information note	CAknNoteDialog	CAknInformationNote
Warning note	CAknNoteDialog	CAknWarningNote
Error note	CAknNoteDialog	CAknErrorNote
Progress note	CAknProgressDialog	n/a
Wait note	CAknWaitDialog	CAknWaitNoteWrapper
Permanent note	CAknStaticNoteDialog	n/a

Notes:

- Series 60 defines standard sets of flags for note dialogs in `avkon.hrh`, for instance `EAknErrorNoteFlags`.
- All note dialogs should contain an `AVKON_NOTE` resource, whose type is `EAknCtNote`.
- The layout of note dialogs varies, so you need to specify the right value in the `AVKON_NOTE`'s layout field, for instance `EProgressLayout` for a progress note.
- Like query dialogs, note dialogs can play a tone when they execute.
- `AVKON_NOTE`s can contain a label (in singular and plural versions), an image, an icon and an animation. You can select whether to use the singular or plural version of the text label by calling `CAknNoteDialog::SetTextPluralityL()` and you can set the number in the label dynamically using `CAknNoteDialog::SetTextNumberL()`.

Global Notes and Queries

Global note and query dialogs are similar to standard note and query dialogs except that they are displayed even if the application that launched them is switched away from or closed down. They can be used by UI or engine components, and are not defined by resources. They are used only in unusual circumstances, for example to display the error that caused an application to shut down.

A global note is a `CAknGlobalNote`. After initialization, call `ShowNoteL()`, specifying the type of note (enumerated in `TAknGlobalNoteType`). There are three types of global queries: global confirmation queries (`CAknGlobalConfirmationQuery`), global list queries

(`CAknGlobalListQuery`) and global message queries (`CAknGlob-alMsgQuery`).

Forms

A form is a type of dialog specific to Series 60 that looks like a list box, but which allows list items to be edited. Forms support two modes of operation – view mode and edit mode – which users can switch between. Forms can alternatively support edit mode only.

Forms are defined in resource files using a `FORM` resource which can be assigned to the `form` field in a `DIALOG` or to the `form` field in a `PAGE`. In either case, the softkeys are defined in the dialog.

`FORM` flags are defined in `uikon.hrh`:

- `EEikFormShowEmptyFields` – shows lines without a value (this is the default).
- `EEikFormHideEmptyFields` – hides lines without a value.
- `EEikFormShowBitmaps` – displays bitmaps in dialog lines.
- `EEikFormEditModeOnly` – the form supports edit mode only.
- `EEikFormUseDoubleSpacedFormat` – the control is displayed beneath the line containing the bitmap and label (Figure 2.10).

Forms have a default menu bar and pane, which are used if you don't specify a menu bar resource when calling `CAknForm::ConstructL()`. The default menu pane includes standard options including **Save**, **Add field** and **Delete field**.

A form should minimally implement `SaveFormDataL()`, which is called when the user chooses **Save**, and `DoNotSaveFormDataL()`,

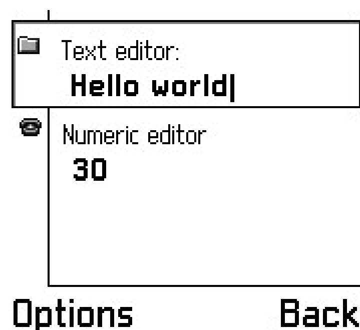

Figure 2.10 A double-spaced form in edit mode with bitmaps and prompts

which is called when the user chooses not to save their changes – this may be implemented to reset the form to contain default values.

Popup fields can only be used inside forms: see Section 2.5.1.2.

2.4.4 List Boxes

List boxes display information in columns. All list boxes in UIQ and Series 60 are ultimately derived from `CEikListBox`. The main types of list boxes defined by Uikon are:

- text list boxes – `CEikTextListBox`
- column list boxes – `CEikColumnListBox`
- hierarchical list boxes – `CEikHierarchicalListBox`.

The Series 60 UI is designed with an emphasis on viewing rather than creating data and lists are the main way in which data is displayed. Series 60, unlike UIQ, defines many of its own specialized list box types, described below.

In UIQ, you may need to create your own list boxes. A `LISTBOX` resource is used to define a list box. You can customize it using flags defined in `uikon.hrh`, for instance `EEikListBoxMultipleSelection`. Its contents can be set either in the resource, or in C++, through the list box model (`CTextListBoxModel::SetItemTextArray()`). You may also need to set a list box observer so that you are notified when an item is selected.

In Series 60, if the list box is in a dialog, you could use a list query dialog, or a selection or markable list box dialog instead. In UIQ, **choice lists** are often used in dialogs. These are described in Section 2.5.2. UIQ does not define any standard list box dialogs.

2.4.4.1 Series 60-specific List Boxes

Series 60 defines the following list boxes:

Name	Class	Control type
Single list box	`CAknSingleStyleListBox`	`EAknCtSingleListBox`
Single numbered list box	`CAknSingleNumberStyleListBox`	`EAknCtSingleNumberListBox`
Single list box with titles (titles are displayed to the left of the item text, in a different font)	`CAknSingleHeadingStyleListBox`	`EAknCtSingleHeadingListBox`

Name	Class	Control type
Single list box with graphics	`CAknSingleGraphicStyleListBox`	`EAknCtSingleGraphicListBox`
Single list box with titles and graphics	`CAknSingleGraphicHeading StyleListBox`	`EAknCtSingleGraphicHeading ListBox`
Single numbered list box with titles	`CAknSingleNumberHeading StyleListBox`	`EAknCtSingleNumberHeading ListBox`
Single list box with increased spacing between list items	`CAknSingleLargeStyleListBox`	`EAknCtSingleLargeListBox`
Double list box	`CAknDoubleStyleListBox`	`EAknCtDoubleListBox`
Double numbered list box	`CAknDoubleNumberStyleListBox`	`EAknCtDoubleNumberListBox`
Double list box where items can contain the time, including optional am/pm text	`CAknDoubleTimeStyleListBox`	`EAknCtDoubleTimeListBox`
Double list box with increased spacing between list items	`CAknDoubleLargeStyleListBox`	`EAknCtDoubleLargeListBox`
Double list box with graphics	`CAknDoubleGraphicStyleListBox`	`EAknCtDoubleGraphicListBox`
Settings list box (the user can change the value of each item)	`CAknSettingStyleListBox`	`EAknCtSettingListBox`
Numbered settings list box	`CAknSettingNumberStyleListBox`	`EAknCtSettingNumberListBox`

The contents of each list box column are of the same type, so for instance there are separate columns for numbers, titles, icons and the main list item text. List box contents are defined in the resource file by list item strings that use tab characters as column delimiters. All of the list boxes in the table above are declared in the header file `aknlists.h`, and comments in `aknlists.h` describe the list item string format for each.

Icons are identified in list item strings by their index into the list box's icon array, so before you can use a list box with graphics, the

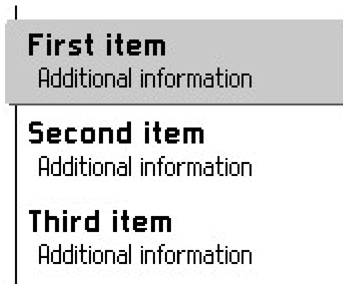

Figure 2.11 A double list box

list box's icon array must have been set using CColumnListBox-Data::SetIconArray(). This applies also to list query dialogs and selection list dialogs, which provide a SetIconArrayL() function.

The difference between single and double list boxes is that double list boxes can have two lines of text per item in the main list item text column, while single list boxes have one. Figure 2.11 shows an example of a double list box.

All the list boxes in the table above except for Settings list boxes can support multiple selection, depending on flags specified in the LISTBOX resource, for instance EAknListBoxMarkableList and EAknList-BoxMultiselectionList.

2.4.4.2 Grids

Grids are a type of two-dimensional list box with columns and rows. They are more commonly used in Series 60 than in UIQ, so Series 60 provides special support for them.

CAknGrid is the base class for Series 60 grids. Because it is derived from CEikListBox and its model (CAknGridM) is derived from CText-ListBoxModel, grids are used in a similar way to lists.

A grid is defined by a GRID resource and constructed using CAkn-Grid::ConstructFromResourceL(), or in C++ using CAknGrid::ConstructL().

You can customize the grid's layout either by using a GRID_STYLE resource definition or by calling CAknGrid::SetLayoutL(). Layout variables are:

- the orientation (horizontal or vertical)
- the scroll type, for instance whether to loop back to the first item in the row or column when the last item has been reached, or to stop scrolling, or to move to the start of the next row
- the number of rows and columns to display

- the size of gaps between rows and columns
- the size of grid items.

Cells in the grid can contain text, graphics or both. If the grid contains graphics, you need to set its icon array, in a similar way to a list box, using `CFormattedCellListBoxData::SetIconArrayL()`. To set and get the currently selected item(s) use `CAknGrid::SetCurrentDataIndex()` and `CAknGrid::CurrentDataIndex()`.

Popup and multiple selection grids are possible. Popup grids are created using a `CAknPopupList`, which provides a title and softkeys. For a multiple selection grid, specify the `EAknListBoxMarkableGrid` flag on construction.

2.4.5 Editors

UIQ and Series 60 define some standard editors that can be reused in third-party applications, including inside dialogs. They are listed in the table below. Common resource structs are defined in `uikon.rh` or `eikon.rh`, UIQ's in `Qikon.rh`, and Series 60's in `avkon.rh`.

Name	**UIQ class name, control type, resource struct**	**Series 60 class name, control type, resource struct**
Integer editor	CQikNumberEditor EQikCtNumberEditor QIK_NUMBER_EDITOR	CEikNumberEditor EEikCtNumberEditor NUMBER_EDITOR
Floating point editor	CQikFloatingPointEditor EQikCtFloatingPointEditor QIK_FLOATING_POINT_EDITOR	CEikFloatingPointEditor EEikCtFlPtEd FLPTED
Time editor	CQikTimeEditor EQikCtTimeEditor QIK_TIME_EDITOR	CEikTimeEditor EEikCtTimeEditor TIME_EDITOR
Popup calendar	CEikCalendar n/a R_EIK_ONE_MONTH_CALENDAR	n/a
Date editor	CQikDateEditor EQikCtDateEditor QIK_DATE_EDITOR	CEikDateEditor EEikCtDateEditor DATE_EDITOR
Time and date editor	CQikTimeAndDateEditor EQikCtTimeAndDateEditor QIK_TIME_AND_DATE_EDITOR	CEikTimeAndDateEditor EEikCtTimeAndDateEditor TIME_AND_DATE_EDITOR
Duration editor	CQikDurationEditor EQikCtDurationEditor QIK_DURATION_EDITOR	CEikDurationEditor EEikCtDurationEditor DURATION_EDITOR

(*continued overleaf*)

Name	UIQ class name, control type, resource struct	Series 60 class name, control type, resource struct
Time offset editor	n/a	CEikTimeOffsetEditor EEikCtTimeOffsetEditor TIME_OFFSET_EDITOR
Plain text editor	CEikEdwin EEikCtEdwin EDWIN	CEikEdwin EEikCtEdwin EDWIN
Global text editor	CEikGlobalTextEditor EEikCtGlobalTextEditor GTXTED	CEikGlobalTextEditor EEikCtGlobalTextEditor GTXTED
Rich text editor	CEikRichTextEditor EEikCtRichTextEditor RTXTED	CEikRichTextEditor EEikCtRichTextEditor RTXTED
Secret editor	CEikSecretEditor EEikCtSecretEd SECRETED	CEikSecretEditor EEikCtSecretEd SECRETED
PIN editor	n/a	CAknNumericSecretEditor EAknCtNumericSecretEditor NUMSECRETED
Color selector	CQikColorSelector EQikCtColorSelector QIK_COLOR_SEL	n/a
Sound selector	CQikSoundSelector EQikCtSoundSelector n/a	n/a
IP editor	CQikIpEditor n/a n/a	CAknIpFieldEditor EAknCtIpFieldEditor IP_FIELD_EDITOR
Slider	CQikSlider EQikCtSlider QIK_SLIDER	CAknSlider EAknCtSlider SLIDER
Phone number editor	n/a	CAknPhoneNumberEditor EAknPhoneNumberEditor PHONE_NUMBER_EDITOR

Notes:

- The UIQ integer editor uses spinner arrows to change the number.
- The UIQ time, date and duration editors use popup time pickers or calendars.
- The secret editor is alphanumeric in UIQ, and alphabetic in Series 60. For numeric passwords in Series 60, use the PIN editor.

- UIQ supports horizontal and vertical sliders; Series 60 supports horizontal only. Series 60 supports different slider layouts and the slider can display a value, a percentage or a fraction.

2.4.6 Progress Bars

In both UIs, a progress bar can be defined in a resource file using a `PROGRESSINFO` resource struct. This has a control type of `EEikCtProgInfo`, which corresponds to `CEikProgressInfo` in C++. The progress bar can contain text showing the progress as a fraction or a percentage (Figure 2.12), or it can be divided up by vertical lines.

To update and redraw the progress bar, call `CEikProgressInfo::IncrementAndDraw()` or `CEikProgressInfo::SetAndDraw()`.

In Series 60, progress bars are usually displayed using the progress note dialog (`CAknProgressDialog`).

Figure 2.12 UIQ progress bar

2.4.7 Navigation Tabs

In Series 60, tabs in the navigation pane can be used to switch between application views. Figure 2.13 shows an example.

UIQ also supports navigation tabs which are located at the bottom of the screen, in the area occupied by the toolbar (see Figure 2.14). They are not used to switch view in UIQ, but to switch between pages in the same view. A page may be a scrollable container, for instance.

Series 60 navigation pane tabs can be confused with multi-page dialog tabs, which are also displayed in the navigation pane and are used for switching between dialog pages. These were described earlier in this chapter.

Series 60 navigation pane tabs are defined by `TAB`s in a `TAB_GROUP` resource, each of which has an ID and can contain text or a bitmap or

Figure 2.13 A tab group in the Series 60 Profiles application

Figure 2.14 Two navigation tabs in the UIQ Agenda

both. The tab group consumes left and right key events and the application should change the view depending on which tab is active.

You can customize the number of tabs visible in the pane, with a maximum of four. If not all tabs are visible, navigation arrows are displayed (see Figure 2.13).

To set the navigation pane's tab group you first need access to the status pane (both the app UI and the view provide a `StatusPane()` function), and through this you can get a handle to the navigation pane. The following code initializes the navigation pane with a `TAB_GROUP` called `R_MY_TAB_GROUP`.

```
CAknNavigationControlContainer* naviPane =
STATIC_CAST(CAknNavigationControlContainer*, StatusPane()->
    ControlL(TUid::Uid(EEikStatusPaneUidNavi)));
TResourceReader reader;
iCoeEnv->CreateResourceReaderLC(reader, R_MY_TAB_GROUP);
CAknNavigationDecorator* naviTabGroup = naviPane->CreateTabGroupL(reader);
naviPane->PushL(*naviTabGroup); // set the tab group to be visible in the
                                // navigation pane
```

You can set the application's initial navigation pane by defining a `STATUS_PANE_APP_MODEL` resource that contains a `NAVI_DECORATOR` pane. This can be set via the `status_pane` field in the application's `EIK_APP_INFO` resource.

The equivalent of navigation pane tabs in UIQ is the tab screen. This is constructed either in code (`CQikTabScreen`) or from a `QIK_TABSCREEN` resource. As in Series 60, tabs in the tab screen can contain text or a bitmap or both. Each tab is associated with a page and tab/page pairs are added to the tab screen using `CQikTabScreen::AddTabPageL()`. The tab screen can contain an additional control that is not a tab, for instance a **Done** button.

2.4.8 Messages and Notifications

Every control in a GUI application has a member called `iEikonEnv`, which is a pointer to the GUI environment, `CEikonEnv`. `CEikonEnv` is common to both UIs and provides many useful functions, including access to information and busy messages.

`CAknEnv` is a Series 60 extension to `CEikonEnv`. A `CAknEnv` instance is created at the same time as `CEikonEnv` and is owned by `CEikonEnv`. All applications have access to it through a global pointer called `iAvkonEnv`, although `iEikonEnv` is much more useful to third-party applications.

The following messages and notifications are common to both UIs.

- Info messages. These are concise messages displayed by default in the top right-hand corner of the screen for three seconds. They are used to

notify the user of a minor error, or that something without visual impact has happened. They are generated using `iEikonEnv->InfoMsg()`.

- Busy messages. These are flashing text messages that notify the user that the application is busy. They are generated using `iEikonEnv->BusyMsgL()`. They do not have a fixed duration; they end when `iEikonEnv->BusyMsgCancel()` is called.
- Info windows. These are used to notify the user of more serious errors, for instance that a file is corrupt. An info window is a dialog with a title, a description and a button labeled **Continue** (UIQ), or **Back** (Series 60), as shown in Figure 2.15. They are generated using `iEikonEnv->InfoWinL()`.
- Alert windows. These are similar to info windows except that resources are guaranteed to be available for them. Also, in UIQ, the title contains the word **Information**. They are generated using `iEikonEnv->AlertWin()`.
- Query windows. These are similar to info windows, except they have buttons labelled **Yes** and **No** and they return a value that indicates which button was pressed (see Figure 2.16). They are generated using `iEikonEnv->QueryWinL()`.

Note that in Series 60, note and query dialogs can alternatively be used for notification.

Figure 2.15 A UIQ info window

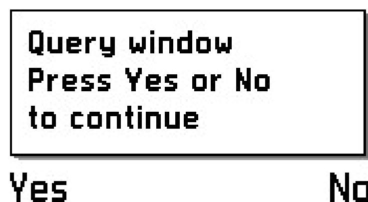

Figure 2.16 A Series 60 query window

2.5 UI-specific Components

2.5.1 Buttons

UIQ's touch-sensitive screen allows it to use buttons extensively. UIQ uses three types of button: command buttons, option buttons and check boxes. Series 60 does not support any of these, although similar behavior can be achieved using other controls.

2.5.1.1 Command Buttons

Command buttons are often used in UIQ's toolbar and in dialogs. Different types are available:

- Standard command buttons (class `CEikCommandButton`/resource struct `CMBUT`) contain text, or an icon, or both. Their control type is `EEikCtCommandButton`.
- Two picture command buttons (`CEikTwoPictureCommandButton`/`PICMBUT`) contain text, or an icon that changes when the button is pressed, or both. Their type is `EEikCtTwoPictureCommandButton`.
- Bitmap buttons (`CEikBitmapButton`/`BMPBUT`) contain no text, just one or two bitmaps. Their type is `EEikCtBitmapButton`.
- Text buttons (`CEikTextButton`/`TXTBUT`) contain text only. Their type is `EEikCtTextButton`.
- Menu buttons (`CEikMenuButton`/`MNBUT`) contain text or an icon, or both, and launch a popout menu pane. Their type is `EEikCtMenuButton`.

A command button can be dimmed if unavailable using its `SetDimmed()` function. Properties of the button and its layout can be set using flags defined in `uikon.hrh`. For instance, a button can latch, meaning that it stays set after it has been pressed and released, or it can stay set, regardless of whether it is pressed.

Standard command IDs issued by buttons are defined in `uikon.hrh`, for instance `EEikBidCancel` and `EEikBidOk`.

In Series 60, the softkeys are normally used instead of command buttons.

2.5.1.2 Option Buttons

Option buttons, sometimes referred to as radio buttons, are used to select one from a list of options. In UIQ they are usually held in either a vertical or a horizontal list:

	Vertical option button list	Horizontal option button list
Class name	CQikVertOptionButtonList	CEikHorOptionButtonList
Resource struct	QIK_VERTOPBUT	HOROPBUT
Control type	EQikCtVertOptionButtonList	EEikCtHorOptionButList

Option buttons are defined by `OPBUT` resources which have an `id` and a `label`. The option button list classes' `SetButtonById()` function is used to change the selected button.

Option buttons can be used in menus as well as dialogs. This is done using the `MENU_ITEM`'s flags field, specifying one of the flags `EEikMenuItemRadioStart`, `EEikMenuItemRadioMiddle` or `EEikMenuItemRadioEnd` (defined in `uikon.hrh`) for each of the menu options to be included in the option button list. An option button in a menu item is set or unset using `CEikMenuPane::SetItemButtonState()`.

Radio button behavior is available in Series 60 using a popup field (`CAknPopupField`). This is a control with two possible modes: label mode and selection list mode. In label mode, the control displays a text label. When it receives an **Ok** or **Confirm key** event, it switches to selection list mode and displays a popup selection list containing radio buttons, causing the form to redraw its contents around the list. Figure 2.17 shows an example. Note that popup fields can be used only in forms, and the form must be in edit mode before the popup selection list can be displayed. When the field is in label mode and it is selected, an alternative to pressing **Confirm** is to use the left and right navigation keys to cycle through the available options.

Optionally, the popup field can allow users to enter new values, using a button in the popup list usually labeled **Other** ... (or its equivalent).

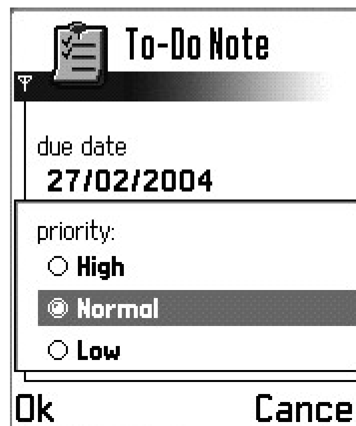

Figure 2.17 Popup selection list from Series 60 To-Do application

This makes them similar to combo boxes (see Section 2.5.2). Selecting this button invokes an editor, allowing the user to add an entry to the list. The `EAknPopupFieldFlagAllowsUserDefinedEntry` flag should be specified in the resource to allow this, or you can call `CAknPopupField::SetAllowsUserDefinedEntry()`.

Popup fields support different data types, for instance text, integers, dates and times, so an appropriate editor must be used to create new entries. The editor, and the initial array of values to display in the popup selection list, are specified by calling `CAknPopupField::SetQueryValueL()`, with an object that implements the `MAknQueryValue` interface.

An exception to this is a text-only popup field, `CAknPopupFieldText` (for example, the selection list shown in Figure 2.17). Its initial values are specified in its resource definition (a `POPUP_FIELD_TEXT` struct, with a control type of `EAknCtPopupFieldText`), which removes the need to call `SetQueryValueL()` and significantly simplifies its construction.

2.5.1.3 *Check Boxes*

If multiple list options can be selected, then check boxes should be used instead of option buttons.

Figure 2.18 shows a dialog from UIQ's Agenda application that uses seven dialog lines, each of which has a prompt that is the day of the week and a control of type `EikCtCheckBox`, which in C++ corresponds to a `CEikCheckBox`.

Series 60 supports multiple selection in markable list dialogs and list boxes.

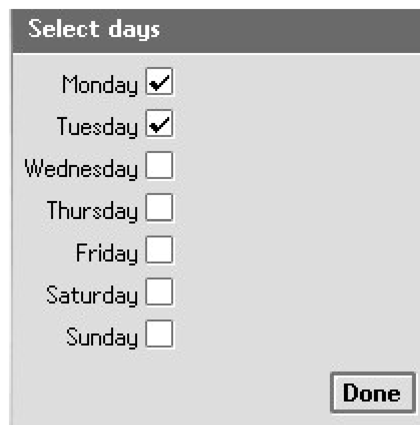

Figure 2.18 Check boxes in UIQ

2.5.2 Choice Lists and Combo Boxes

Choice lists are a type of single selection text list box that display a single list item only, but can use a popout box to display the whole list. In UIQ they are often used where space is limited, for instance in dialogs. Choice lists are not supported in Series 60; the custom list dialogs provide similar behavior.

A choice list is defined using a `CHOICELIST` resource. This has a control type of `EEikCtChoiceList` and corresponds to a `CEikChoiceList` in C++. You can specify its width using the `maxdisplaychar` field, otherwise it defaults to the width of the widest item. The list's contents are set either in the resource file using an `ARRAY` resource, or in C++ using `CEikChoiceList::SetArrayL()`.

You can customize the list through `CEikChoiceList`, including its alignment, font and borders, and you can disallow the list popout, using `CEikChoiceList::AllowPopout()`. `SetCurrentItem()` and `CurrentItem()` set and get the selected item.

Combo boxes are a combination of choice list and text editor. They allow new items to be added and existing items to be edited. The `COMBOBOX` resource defines a combo box. It has a control type of `EEikCtComboBox`, which corresponds to a `CEikComboBox` in C++. A `COMBOBOX` defines the width of the edit box in characters, the maximum number of characters that can be entered by the user and the maximum number of entries in the array.

A `CEikComboBox` is used in a similar way to a `CEikChoiceList`. The array can be initialized from a resource `ARRAY`, or from a C++ array (by calling `CEikComboBox::SetArrayL()`), and the selected item is set and retrieved using `SetTextL()` and `GetText()`.

Series 60 does not support combo boxes; its closest equivalent is popup fields in forms.

2.5.3 Scroll Bars

In UIQ, users can scroll either using the hardware navigation keys, or using scroll bars. UIQ supports both vertical and horizontal scroll bars. Scroll bars allow different types of scrolling; for instance, the scroll bar thumb can be dragged, or the scroll arrows can be pressed. As an alternative to using scroll bars, scroll arrows alone can be used. These save screen space by avoiding the need for a horizontal or vertical scroll bar (see Figure 2.19).

Series 60 does not support scroll bars, but instead displays scroll indicators in the control pane, between the softkey labels. Their color tone changes to indicate how much information remains to be scrolled. Series 60 does not support horizontal scrolling.

Figure 2.19 'Floating' scroll arrows in UIQ

In either UI, if you want scroll indicators for a list box or text editor, you must create them yourself and you must call the list box's or editor's `UpdateScrollBarsL()` function when the scroll bar needs updating, for instance when items are added to or removed from the list.

Scroll bars and arrows can be created as follows:

```
listbox->CreateScrollBarFrameL();
// Set the horizontal scroll bars to be invisible and vertical scroll bars
//  to be visible, when needed
listbox->ScrollBarFrame()->SetScrollBarVisibilityL
      (CEikScrollBarFrame::EOff, CEikScrollBarFrame::EAuto);
// Make the scrollbar floating (removing the scroll bars)
listbox->ScrollBarFrame()->SetScrollBarManagement
      (CEikScrollBar::EVertical,CEikScrollBarFrame::EFloating);
// Give the floating scrollbar standard scroll arrows
listbox->ScrollBarFrame()->SetScrollBarControlType
      (CEikScrollBar::EVertical,EQikCtArrowHeadPageScrollBar);
```

UIQ provides a control container called `CQikScrollableContainer` that is useful for managing scroll bars for a list of controls that do not all fit on the screen.

2.6 Skins

Skins (also called themes) are used to customize a phone UI's appearance at runtime. Both UIs support them, but they are implemented differently. Support for skins was introduced in Series 60 v2.0 and in UIQ 2.1.

In both UIs, skins specify a color scheme and optionally a bitmap to display in the background. In UIQ the background bitmap, if specified, is displayed only in the application launcher while in Series 60 it is displayed in all applications. UIQ skins can customize the system sounds, for instance the sounds made when an email or SMS is received. Series 60 skins cannot do this, but can customize the icons used throughout the UI, including application icons.

2.6.1 Skins in UIQ

Skin-aware applications in UIQ must do all drawing using the logical colors enumerated in `TLogicalColor`, rather than physical colors,

and controls should implement CCoeControl::HandleResourceChange(). This should respond to messages of type KEikMessageColorSchemeChange, which indicates a color scheme change, by updating any colors to use the new values, for instance:

```
void CMyAppView::HandleResourceChange(TInt aType)
    {
    if (aType == KEikMessageColorSchemeChange)
            {
            Window().SetBackgroundColor(iEikonEnv->
            ControlColor(EColorControlBackground, *this));
            }
    CCoeControl::HandleResourceChange(aType);
    }
```

2.6.2 Skins in Series 60

In Series 60, skins have a greater impact on applications than in UIQ. Skins are automatically applied to menus, note and query dialogs, the control and status panes, the fast swap window and, if present, the Chinese FEP. Other UI components may support skins, depending on whether the application is skin-aware, and the control is skin-enabled.

To make an application skin-aware, call:

```
BaseConstructL(CAknAppUi::EAknEnableSkin);
```

in the app UI's ConstructL(). In a skin-aware application, skins are applied to all standard Series 60 controls, for instance search boxes, list boxes and grids. If BaseConstructL() is not called with this parameter, only the controls mentioned in the first paragraph are drawn using skins.

To draw non-standard, custom controls and container controls using skins, you need to use the skin-drawing utility class AknsDrawUtils (the Akns prefix stands for Avkon Skins). Before you can do this, you first need to get the currently active skin by calling AknsUtils::SkinInstance(), and non-container controls also need to call AknsDrawUtils::ControlContext() to get the skin control context, described below.

If the skin does not provide a resource for a particular UI component, then it should be drawn using a graphics context as in Series 60 1.x, instead of using AknsDrawUtils. This is the case for all controls, even in skin-aware applications, if the user selects the default Series 60 skin. The return values from AknsDrawUtils draw methods indicate whether a skin was used for drawing.

As in UIQ, applications may need to check for skin change events in their HandleResourceChange() function. In Series 60, these are identified by KAknsMessageSkinChange messages.

2.6.2.1 Skin Control Context

The skin control context provides the ID of the background image to use from the active skin, the rectangle to draw it in, and image attributes, for instance whether to draw the bitmap tiled or stretched. The skin instance and skin control context are needed by all `AknsDrawUtils` draw functions.

It is up to the container control to provide a skin control context to its contained controls. Skin control contexts are objects derived from `MAknsControlContext`, for instance:

- `CAknsBasicBackgroundControlContext`
- `CAknsListBoxBackgroundControlContext`
- `CAknsFrameBackgroundControlContext`.

The container owns the context and returns a pointer to it in its implementation of `MopSupplyObject()`. For example,

```
TTypeUid::Ptr CMyAppContainer::MopSupplyObject(TTypeUid aId)
     {
     if(aId.iUid == MAknsControlContext::ETypeId)
             {
             return MAknsControlContext::SupplyMopObject(aId,
                 iSkinControlContext);
             }
     return CCoeControl::MopSupplyObject(aId);
     }
```

An object provider chain must exist from the container to all the controls that need the control context; the chain is set up by calling `CCoeControl::SetMopParent()` for all controls in the chain.

When constructing a skin control context, the ID of the background bitmap must be specified. These IDs are defined in `aknsconstants.h`. For instance, `KAknsIIDQsnBgAreaMain` identifies the bitmap to use in the main pane.

2.7 Handling User Input

The main types of user input that an application needs to handle are commands, key events and, in UIQ, pointer events.

2.7.1 Commands

When a command is generated, for instance by a user selecting a menu item, or pressing a Series 60 softkey or a UIQ toolbar button, the framework passes the ID of the command to the app UI of the foreground

application, by calling its `HandleCommandL()` function. The app UI can handle the command itself, or can pass it to a view or other control for handling.

Most command IDs are application-specific, but both UIs provide some standard ones. In `avkon.hrh`, Series 60 defines the command IDs generated by the standard softkey resource definitions, for instance `EAknSoftkeyOptions` and `EAknSoftkeyExit`, which are issued by the `R_AVKON_SOFTKEYS_OPTIONS_EXIT CBA`.

UIQ defines the command IDs generated by standard toolbar and dialog buttons in `uikon.hrh`, for instance `EEikBidCancel`. It also defines some command IDs in `Qikon.hrh`, including those generated by the set zoom dialog.

`EEikCmdExit` is defined in Symbian OS in `uikon.hrh` and is generated by the system when it needs to close down applications. It must be handled by app UIs in both Series 60 and UIQ.

2.7.2 Key Events

A keypress generates three types of key event: a key down (with an event code of `EEventKeyDown`), one or more standard key events (`EEvent-Key`) and a key up (`EEventKeyUp`). On receiving any of these types of key event, the UI framework calls `CCoeControl::OfferKeyEventL()` for all controls in the foreground application's control stack, starting with the control at the top of the stack, until a control consumes it. Applications conventionally ignore up and down key events and consume only standard events.

The parameters of `OfferKeyEventL()` are the key event and the event code. The key event contains the key code (`TKeyEvent::iCode`) and any modifier keys (`TKeyEvent::iModifiers`), for instance Shift, that were held down, and specifies whether the key was held down long enough to repeat (`TKeyEvent::iRepeats`).

In both UIs, it is rare for applications to need to handle modifier keys. The only modifier key supported in Series 60 is Shift (`EModifierShift`), which is generated by pressing the Edit key. In UIQ, modifier keys may not be supported, depending on the active FEP, so it is not recommended for applications to rely on them.

In Series 60, some applications treat long and short key presses differently. For instance, the Series 60 Calculator clears all user input if the Clear key is pressed and held down, in other words if its `TKeyEvent::iRepeats` value is greater than zero.

Note that in Series 60, although the left and right softkeys generate key events (`EKeyCBA1` and `EKeyCBA2`), applications should in general handle the generated commands instead. The mapping between softkeys and command IDs is defined in the application's resource file.

As well as the standard alphanumeric keypad keys, applications may need to handle some of the special Symbian OS key codes defined

in e32keys.h and some UI-specific ones. Series 60's are defined in uikon.hrh and UIQ's in quartzkeys.h. The following table lists the most common key codes that applications need to handle.

Series 60	UIQ	Purpose
EKeyLeftArrow, EKeyRightArrow	EQuartzKeyFourWayLeft, EQuartzKeyFourWayRight	Used to navigate horizontally, for instance between navigation tabs.
EKeyUpArrow, EKeyDownArrow	EQuartzKeyTwoWayUp or EQuartzKeyFourWayUp, EQuartzKeyTwoWayDown or EQuartzKeyFourWayDown	Used to navigate vertically, for instance in lists and menu panes. EQuartzKeyTwoWayUp and EQuartzKeyTwoWayDown may sometimes be used to implement horizontal navigation.
EKeyOK	EQuartzKeyConfirm	Generated by pressing the Confirm/Selection key to open or activate an item. In Series 60, in some circumstances, it may be used to activate a short context-sensitive popup menu instead. EKeyEnter is not generated in Series 60 or UIQ, except on the emulator.
EKeyBackspace	EKeyBackspace	In Series 60, this is generated by the Clear key and is used for deletion.

2.7.2.1 FEPs

A FEP (short for front end processor) is a DLL that allows users to input characters that are not directly available on the phone's keypad. In both UIs, the FEP communicates directly with editors, so no action is required by third-party developers to use a FEP.

In UIQ, the FEP is the main source of text input. UIQ supports multiple FEPs on a phone, which the user can switch between, but only one can be enabled at any time. Most UIQ phones provide at least a handwriting recognition FEP and an on-screen virtual keyboard.

Series 60 phones support a single FEP only that is always loaded and available to applications. It supports multitap text entry, where one or more numeric key presses outputs a single alphanumeric character, and predictive text entry, where each key press causes a single character to be predicted and output by consulting a multiple language dictionary.

The Series 60 FEP distinguishes between short and long key presses. For instance, in numeric input mode, a short press of the # key inputs the hash character, but a long press changes the FEP to alphabetic input mode. It is also aware of the input capabilities of the target editor, so for instance will automatically change to numeric input mode when a phone

number editor has focus. In the Asia Pacific region, it has separate modes for inputting text in different languages. Sometimes a device may require both. For example, in Hong Kong, both Chinese input and English input may be needed on the same phone.

2.7.3 Pointer Events

As with key events, the UI framework ensures that pointer events are sent to the right control for handling. `CCoeControl::HandlePointerEventL()` takes a `TPointerEvent` parameter that packages information about the event. This includes the event's type, for instance pen down and pen up, any keyboard modifiers, and the event's position on the screen.

In UIQ, items are selected and activated by a single pen tap, so pen down events (of type `TPointerEvent::EButton1Down`) and drag events (`TPointerEvent::EDrag`) generally change the selection, and pen up events (`TPointerEvent::EButton1Up`) activate the selection.

Some controls need to handle pointer drag events, for example sliders (`CQikSlider`). To do this you must first enable them, by calling `CCoeControl::EnableDragEvents()`. Repeat pointer events (`TPointerEvent::EButtonRepeat`) may also need to be handled, for instance by spinner arrows in a numeric editor or the hours and minutes controls in a popout time picker. In order to receive repeating pointer events, the control must first call `RequestPointerRepeatEvent()` on its window, specifying the initial delay before the first repeat event is generated. Then it should be called repeatedly, specifying the delay before the next repeat event.

As in key event handling, modifiers are usually ignored.

2.8 Summary

The aim of this chapter was to show that despite the obvious differences to a user, the Series 60 and UIQ application frameworks have a lot in common.

- Both are built on top of Uikon, the common UI framework, which means applications have the same structure and have common base classes.
- In both UIs, menus are generally defined using standard Uikon resources and C++ classes.
- Dialogs are widely used in both UIs. Although Series 60 defines many specialized dialog types, all of them are ultimately derived from Uikon's dialog base class `CEikDialog`, so follow a standard pattern of use.

- Most editors either are common to both UIs or have a near equivalent in the other UI.
- Buttons are widely used in UIQ, but not in Series 60, although equivalent behavior can usually be achieved using other means.
- Command and event handling is similar in both UIs, although some of the command IDs and key event codes differ, and UIQ applications need to handle pointer as well as key events.