Series 60 Application Framework Handbook

Contents

Legal Notice

1. Introduction

1.1 Purpose and scope

1.1.1 Purpose

The purpose of this document is to provide information to those who write Symbian OS application UIs (App UIs) for the Avkon products or other products that use Series 60 UI.

1.1.2 Scope

This document is intended for people who work with application design for devices using the Series 60 UI.

1.2 Related Documents

Series 60 UI Style Guide.

-

1.3 Glossary

The following terms and abbreviations are used within this document.

AVKON	Series 60 extensions and modifications to Symbian's Uikon and other parts of the Symbian OS Application Framework.

2. Application Architecture

Applications are normally split into two parts, the engine and the UI, to aid maintainability and flexibility. The application engine, also known as the application model, deals with the algorithms and data structures needed to represent the applications data. The application UI, sometimes called the app, deals with the on screen presentation of the application data and the overall behaviour of the application. This section describes the different application UI architectural approaches available in Avkon, situations in which each approach is appropriate and information on how to implement code to use them.

Application UIs can be simple with only one main screen, e.g. a calculator, or complex with many screens, e.g. a messaging application. Three architectural approaches have been identified for writing an application UI:

Traditional Symbian OS Control Architecture

Dialog Based Architecture

View Architecture

The choice of application architecture will depend on the application complexity, the view navigation and communications requirements and the screen layout requirements.

Whichever architecture is used, the top level application UI class of each application will be derived from a single application UI base class. The base class does not enforce any choice of UI architecture that must be added by the application UI developer.

2.1 Traditional SYMBIAN OS Application Architecture

Symbian OS applications are traditionally written using CCoeControl derived custom view controls placed on the applications control stack to act as application views. These controls can be created and destroyed, shown and hidden as required by the application to provide the appropriate behaviour.

This approach is very well suited to applications in Avkon. Since many of the Avkon applications will be based on existing UIs written using traditional methods, it is appropriate to follow the same style with Avkon.

This is the most flexible approach to application UI construction. Its main disadvantage over View Architecture is that it is not using a system-provided view management system. This is also its main advantage, because the View Architecture view management system is quite restrictive.

There are many good examples of this type of application architecture as all existing released applications (and all Crystal applications) have been developed this way. There is a lot of experience and support for writing applications with this type of UI architecture. If you can find an existing application with a UI written this way, which is similar to your requirements, you would be advised to use its code and architecture as your starting point.

View switching is achieved by creating, destroying or changing the visibility of the main view controls. Key events are directed to the current control by the app UI placing and removing controls on the control stack, and by manual redirection of key events to controls.

More complex view handling can be achieved using this architecture; e.g. the ER5 web and messaging applications are written this way. The messaging application is a particularly good example, containing multiple internal views as well as externally available editor views. If one of these applications requires views to be switched by an external application, a client/server system is normally needed. The client/server system can also be used to give powerful multi-client access to the application’s data, which is a common requirement when view switching is needed.

2.2 Dialog Architecture

Dialog architecture refers to the case where the dominant model for views in the application is that of dialogs. One particular case of this is where the main view is run as a dialog. Multipage dialogs may be used to give a set of “views” in conformance to the Series 60 UI concept.

One benefit of using dialogs is that the content and layout can be changed in resource files, without rebuilding any C++ code.

Note that nested dialogs can use quite a lot of stack space if not written carefully.

Avkon has automatic status pane tab handling built in to its multi-page dialogs. This contrasts with traditional and view architectures for which the user must drive the navipane tabs.

Refer to DlgApp for an example of an application using the dialog architecture.

2.2.1 Using dialogs for the main view

Where dialog architecture is used for the main view, it is recommended that the dialog be run as a modeless dialog.

2.2.1.1 Resources

Use of a dialog as a main view will usually required using the entire available client rectangle, and will often be a multi-page dialog.

The example here has both of these features:

```
RESOURCE DIALOG r_dlgapp_main_dialog
          {
          flags=EEikDialogFlagNoDrag | EEikDialogFlagNoTitleBar |
                      EEikDialogFlagFillAppClientRect |
                      EEikDialogFlagCbaButtons | EEikDialogFlagModeless;
          buttons=r_dlgapp_softkeys_options_home;
          pages=r_dlgapp_main_pages;
          }
```

The modelessness of the dialog is set by setting EEikDialogFlagModeless. Flags are set to achieve the filling of the client rectangle and to suppress a title bar. CBA (i.e. Softkey) is flagged to be used with EEikDialogFlagCbaButtons.

2.2.1.2 Constructing and Running

This is quite straightforward:

```
void CDlgappAppUi::ConstructL()
          {
          BaseConstructL();
          iAppView=new(ELeave) CDlgAppMainView;
          iAppView->ExecuteLD(R_DLGAPP_MAIN_DIALOG);
          AddToStackL(iAppView);
          }
```

ExecuteLD() returns immediately after being called. The dialog has to be added to the control stack with the AddToStack because modeless dialogs to not do this for themselves.

The application extends the dialog just as it would normally extend a CCoeControl-derived view in order to achieve the desired functionality.

2.2.2 Mixing Dialog and other Application Architectures

Dialog-based architecture can be combined successfully with the other architecture types as long as dialog-based views are only used at the tips of the navigation structure.

Avkon supports view switching within an application and, more significantly, view switching between applications. This feature raises issues for modal dialogs, such as Avkon Forms and Queries. When there is such a modal dialog open, possibly with new data entered, and then one of an application's views is switched to (presumably by another application), then the data associated with editor controls may have to be disposed of.

2.3 View Architecture

View architecture is being made available in the V6 DFRDs. Avkon has modified it to be more suitable for use in Series 60 applications. It is a system that allows applications to register "views", with one view being active in each running application at any one time. It does not dictate what a view is, however it does provide support for a view being a display page on the screen.

The use of views to represent display pages in applications is suitable for applications that do not publish views for use by external application, or which can handle interruption by external applications. Applications that use view architecture each have one active view. When another view is activated in the application, the current view is deactivated. When a view is deactivated, any menus, dialogs or embedded apps active in that view are closed. If an application is entirely in control of its view switches, it can offer the user a choice of how to handle the view switch (e.g. save or discard data). However if an external view switch can interrupt an application, it must be

able to save data state so that the user can later recover it. Note that switching away from an application and then back to it (without using view activation) does not cause view activation and deactivation in the application.

Views can also be used as a message passing system, where the application is written using traditional architecture and the UI display pages are not directly controlled by views. In this case the activation status of a view is not relevant, other than that it brings the active view owning application to the foreground. Instead, view activation can be seen as a request to receiving application to carry out some task. The activated application may have something else it wants to complete first. This rules out the use of display page owning views for applications that need to provide more than one display page in existence at once.

The view management system only allows one view to be active in each application. If you switch to a new view in an application, the current view is immediately deactivated. So it's use is not appropriate in the following cases:

Applications with any view that can not cleanly handle unexpected activation of another view in that application.

Applications that provide views that can be nested over other applications, except where embedding is used.

Applications that provide controls that can be used inside other applications (e.g. using a web control inside an email viewer to show an email with HTML content).

Applications that might want multiple instances running at one time, except where all but one of the instances are embedded applications

Applications that already provide a client/server or other control interface for external access.

The Application UI creates and registers each of the views that it provides, which allows external applications to switch to these views, using a view UID. Views are registered with a server, which knows about all the views in the system.

Views are activated and deactivated by the app UI in response to events from the view server. The app UI arranges for only one view to be active at any time in the application. When a view is deactivated, it may be able to free up memory by deleting memory that is only required while the view is active.

Views also receive events to tell them when they are moved to the foreground or background. The current view for each app will receive this when the app moves between foreground and background. Deactivating views will receive a background event, and activating views will receive a foreground event.

Navipane interaction with the views must be handled manually by the application.

Application views all exist at the same level. They are not intrinsically hierarchical, although you can arrange the inter-view navigation to be hierarchical.

2.3.1 Deriving View Classes

Each view acts like a small application UI. It must provide an Id() function so that it can be identified by the system. It should also override the DoActivateL(), DoDeactivate(), HandleForegroundEventL(), HandleCommandL() and HandleStatusPaneSizeChange() functions to handle various events.

DoActivateL()

This is called when a client requests that your view is activated. The client may or may not pass message parameters to your view in this call. While your view is active, this function will be called again only if a client requests re-activation of your view with some message parameters. Your DoActivateL() call must be prepared to handle this case where it is called while your view is active. If you do not publish your view for external use, or your view does not handle message parameters, a simple check and return if the view is already active will suffice.

DoDeactivate()

This is called when your view is to be deactivated. Your view will only be deactivated when your application exits, or another view in the same application is activated. This function must not leave.

HandleForegroundEventL()

This function will only be called while your view is active (i.e. in-between calls to DoActivateL() and DoDeactivate()). When your view comes to the foreground, it will receive HandleForegroundEventL(ETrue). When you view is removed from the foreground, it will receive HandleForegroundEventL(EFalse). This function will only be called when the foreground state actually changes. Note it may be called a number of times during your views active period, as the owning application comes and goes from the foreground. You may want to use this view for setting focus or controlling screen updates.

HandleCommandL()

This will be called when the views menu generates a command.

HandleStatusPaneSizeChange()

This will be called when the client rectangle size changes due to a change in the status pane.

This is the typical order of events that a view will receive over an active period.

DoActivateL()

HandleForegroundEventL(ETrue)

HandleForegroundEventL(EFalse)

DoDeactivate()

The pair of HandleForegroundEventL() calls may happen a number of times during a views activation.

The DoActivateL() call may occur again before DoDeactivate() is called.

2.3.2 View Resources

To let a view have its own CBA and/or menu, create a AVKON_VIEW resource, then pass the resource id into the view's BaseConstructL() function. Note - this is only appropriate if you are using views to represent display pages.

```
RESOURCE AVKON_VIEW r_viewapp_view1
           {
           hotkeys=r_viewapp_hotkeys;
           menubar=r_viewapp_view1_menubar;
           cba=R_AVKON_SOFTKEYS_OPTIONS_BACK;
           }
```

If there is no menubar resource given, then the view is likely to want to use the application's menubar.

2.3.3 Construction

The instances conforming to CAknView would normally be constructed in the AppUI object's ConstructL. They are registered with AddView and finally, the initial view is activated by setting it as the default view:

```
void CMyViewArchAppUi::ConstructL()
           {
           BaseConstructL();

           CMyViewArchAppView1* view1 = new(ELeave) CMyViewArchAppView1;
           CleanupStack::PushL(view1);
           view1->ConstructL();
           AddViewL(view1);        // Add view1 to CAknAppUi; transfers ownership
           CleanupStack::Pop();    // view1

           CMyViewArchAppView2* view2 = new(ELeave) CMyViewArchAppView2;
           << equivalent Push/ConstructL for view 2>>
           AddViewL(view2);        // transfer ownership to CAknAppUi
           CleanupStack::Pop();    // view2

           CMyViewArchAppView3* view3 = new(ELeave) CMyViewArchAppView3;
           << equivalent Push/ConstructL for view 3>>
           AddViewL(view3);        // transfer ownership to CAknAppUi
           CleanupStack::Pop();    // view3

           SetDefaultViewL(*view1);
<< More code >>
```

For the views to be useful (a view has no drawing capabilities of its own) they will need to hold containers derived from CCoeControl:

```
class CMyViewArchAppView1Container : public CCoeControl,MCoeControlObserver
```

An instance of this will therefore be present in the application writer's CAknView-derived class:

```
class CMyViewArchAppView1 : public CAknView
            {
<< … >>
private:
            CMyViewArchAppView1Container * iView;
            }
```

2.3.4 Command Handling

The currently activated view is given commands in HandleCommandL(). This is where softkey-generated commands and commands generated by a popup menu are handled.

For example:

```
void CMyAppView1::HandleCommandL(TInt aCommand)
            {
            switch (aCommand)
                        {
                        case EMyAppCmdSwitchView:
                                    AppUi()->ActivateLocalViewL(KView2Id);
                                    break;
                        case EAknSoftkeyOk:
                                    {
                                    << Do something >>
                                    break;
                                    }
                        case EAknSoftkeyBack:
                                    {
                                    ((MEikCommandObserver*)AppUi())-
>ProcessCommandL(EEikCmdExit);
                                    break;
                                    }
                        default:
                                    AppUi()->HandleCommandL(aCommand);
                                    break;
                        }
            }
```

The handling of some commands in AppUI may be desirable to handle more globally-defined commands.

2.3.5 Local View Switching

Local switching is done by referring to the UID of the view that you want to switch to.

```
// Now switch the view to view 2
iAvkonViewAppUi->ActivateLocalViewL(TUid::Uid(2));
```

Each view has its own menu system, if it wants to use it, as set in the AVKON_VIEW resource structure. See section 2.3.2.

If, however, the menu system owned by the application is to be used, it's contents must be updated for the new view prior to switching.

```
// Switch to a new menu system for the new view
iEikonEnv->AppUiFactory()->MenuBar()->
            SetMenuTitleResourceId(R_MY_VIEW_ARCH_APP_VIEW2_MENU);
// Now switch the view to view 2
```

2.3.6 Remote View Switching

Call one of the CCoeAppUi::ActivateViewL() functions, giving a TVwsViewId containing the target applications UID and the target view UID.

2.3.7 Demo Application

AppWizard can be used to generate view architecture demo application.

2.3.8 Leave recovery

Avkon view architecture base classes have an automatic recovery mechanism in the case of DoActivateL() leaving. This system will call DoDeactivate() in the view that has left, reinstate the previous view of that application, and return the user to the view that they had just been in. If the application had no previous view, it will exit. If the applications previous view was the same as the activating view (i.e. the view is reactivated with a new message), the application will attempt to recover to the default view.

In most cases, this mechanism removes the need for view writers to put any recovery mechanism in their DoActivateL() methods. The one situation where a view writer may consider something more complex is when their view can be re-activated (i.e. activated with a new message while already active). In this case, the view may attempt a more complex strategy to preserve the existing state if a leave happens during the attempt to handle the new message.

2.4 How to decide which architecture to use.

The following questions may help you to decide whether to use the traditional, dialog or view architecture for you application UI.

2.4.1 Is there an existing application UI that has similar navigation and display characteristics?

If there is and it is well written, it may be easiest to modify it. This implies that you will be using the existing architecture, which is probably the traditional Symbian OS control architecture. You will have to spend time understanding how the existing application code works. But most of the engine interaction and error handling code will already be in place, making the overall application UI development much easier.

2.4.2 Do you have an acyclic graph shaped navigation structure?

If the navigation between the views of your application can be viewed as an acyclic graph, then it may be suited to dialog architecture. There is a slight relaxation of this condition, in that you can navigate directly between sibling views in the graph, if they can be coded as multiple pages in the same dialog.

2.4.3 Are all the application screens dialog like?

If the screen of your application can be written using dialog layout, then consider using a dialog for that screen. Note that a screen consisting of just a choice list can be written as a dialog containing only a choice list field.

2.4.4 Does the application have multiple views or modes, which deal with different sorts of data at the top level?

If so, this rules out the dialog architecture. So choose between traditional or view architecture.

2.4.5 Do external applications need to switch to different views of your application?

Applications are written with view architecture primarily so that they can support multiple different views that external applications can reach. If you are writing an application UI from scratch, consider using the view architecture (but see the next question). Otherwise check if the existing code is suitable for adaptation first. Note - this does not include the case where an external application uses a display page provided by your application as if it was running in the external

applications process. In this case you should implement the display page in a DLL, which the external application can link to.

2.4.6 Can all of the applications views be exited without loosing user data?

Applications that use view architecture to handle their views have to be able to handle unexpected deactivations of their views caused by external application. This is only a problem if external applications can use the views. If applications views can not easily be deactivated automatically, look at traditional or dialog architecture instead. Alternatively, use view architecture as a message passing system, and leave the app UI in charge of view management in a more traditional style.

2.4.7 Do external applications need complex interactions with the data in your app?

If external applications need to have complex interactions with your application's data, you will probably need a client/server system. This reduces the need to use view architecture, by providing alternative ways to switch views and pass data. Existing complex apps, such as messaging, web and contacts use a client/server mechanism, and may be good candidates for UI modification rather than rewriting them.

2.4.8 Is there only a single complex main view in the app?

Use the traditional approach, as dialogs will not be able to handle the layout and there is no need for view architecture. There may be other views that could be interpreted as dialogs hanging off the main view.

2.5 Some examples

This section gives some examples of hypothetical applications and the appropriate application architectures for them.

An assumption that is made for the following example is that in general, only one instance of any app will be running at once. However some parts of the UI may be able to have multiple instances running - e.g. settings screens and new emails.

2.5.1 App launcher

An application (known as the shell in the ER5) which can launch and switch to other applications. The application just has a single internal view. It does not accept external requests that can interrupt it.

This is best implemented as a traditional Symbian OS application. It has no need of message passing, or provision of externally usable views. Existing ER5 shell applications work this way.

2.5.2 Fast swap window

A popup window (known as the task list in ER5) which shows running apps, and allows the user to switch to any running app.

It is just a sleeping popup dialog, not an app. It has no state, which could be interrupted.

This is best implemented as a sleeping dialog-like object; the same way that ER5 task list is done.

2.5.3 Email app

The email app allows reading and creating of emails. This app has externally usable views (the email editor), internally switchable views (the message viewers), and can do interruptible operations (email editing, etc). Notifier messages can tell the app to show new messages to the user. This means that the app could be editing an email when a notifier asks it to display a message. So the app must have a mechanism to pick up such requests, then deal with the situation that it is doing something else.

The editors can be used as views inside other applications, so these are best implemented as dialog like components provided in a separate DLL which external apps can link to. The Symbian OS messaging application has a client server mechanism and interfaces available for launching editors from other applications. There will be some work required to investigate how to launch the editors inside the external applications process.

Since the email app could itself be creating a new email, this should not be terminated by a notifier providing the ability to view a new message. So new message viewing should perhaps be done as a nested dialog-like component inside the messaging app. Note - the best implementation does depend heavily on the exact inter-application interaction model.

2.5.4 Contacts app

The contacts application allows viewing, editing and selection of contacts. Selection of contacts will be available for other apps to use. Also known as "phonebook" in Nokia phones. The contacts app has an externally usable view (the contacts selector). It can also use the email editor from the email app, but it can't be interrupted by external view switch requests.

The contacts selector should be available as a dialog like control in a separate DLL, which external applications can link to. The contacts app itself has no need of views, so should use the traditional application architecture, with dialogs for entry editing.

2.5.5 Web browser

The web browser is for viewing of web page as a whole browser app, or embedded in another document. The web browser offers a UI control in a DLL that external apps can use in their controls, however it does not offer any externally usable full screen views. It does allow external requests to view other pages, so does have to be able to deal with potential interruptions to activities such as writing emails as a result of a mailto link.

The web control embeddable inside other apps should be implemented as a separate DLL (as it is in ER5). The main browser could be implemented as a traditional Symbian OS app, using document opens to open new web pages. It could also use view architecture to receive page open requests.

2.5.6 Settings

This concerns both global settings and application specific settings. The settings app and app local settings are different things, although they may share a common library. They will not interfere with each other.

The settings app does not need to handle external requests, or provide external views, so it can be quite simple - either a traditional or dialog based application. Applications that use local settings views may have to be careful about what they do if they can be interrupted by external requests. Local settings will probably be implemented as modal dialogs.

2.5.7 Telephony app

The telephony app provides no external views, but it does have to accept requests from external sources (e.g. contacts). It must be able to decide whether it can handle the external requests as they happen. It is probably most straight forward to implement this as a traditional Symbian OS app. A lot of the external interaction may go through the ETEL server.

3. Series 60 Application Framework

3.1 Application Startup

A class, CAknApplication is supplied that derives from CEikApplication. This class specialises CEikApplication by reimplementing

PreDocConstructL()

OpenIniFileLC(RFs& aFs).

PreDocConstructL is implemented to first check that an instance of the application being constructed is not already present. If it is present, then the application switches to the existing instance and then exits. This check is only carried out for non-embedded applications.

By default, ini files are not supported by Series 60 applications. OpenIniFileLC is over-ridden with a simple implementation that merely leaves if called. If an .ini file is to be used, the application is forced to implement this method in their application class to call CEikApplication::OpenIniFileLC

3.2 Base classes

3.2.1 CAknDocument

This class is provided as a base class for application documents. By using this class, access to an application document file is never initiated. This situation is acceptable to the majority of Avkon applications.

3.2.2 CAknAppUi

All Avkon applications (apart from Eiksrv) must derive from this class.

This class supports several Avkon-specific functionalities:

KeySound support.

Accessories for CBA and StatusPane

TextResolver – Avkon-specific error reporting from CAknAppUI::HandleError()

Avkon view architecture integration

Control dumping – Debug feature

3.2.3 CAknViewAppUi

All view architecture based applications must derive from this class.

4. Using OPTIONS Menus

4.1 OPTIONS MENUS

Avkon options menus are constructed from menu bar and menu pane resources specified in the .RSS file. The menu is opened in response to either the 'F1' key on the WINS emulator, or in response to the 'Options' softkey, when bound using the EAknSoftkeyOptions Id. Application views wishing to use the pre-existing 'Options' soft-key pane should use the **R_AVKON_SOFTKEYS_OPTIONS_BACK** CBA resource.

4.2 Defining menu sections

Each section of the menu pane is defined as a MENU_PANE resource structure. A menu pane should be defined for each unique menu section. A section is used for each of the menu pane areas - System, Application, View or Context.

A menu pane is defined as follows:

```
RESOURCE MENU_PANE r_system_menu
          {
          items=
                      {
                      MENU_ITEM {command = ECmdCut; txt="Cut";},
                      MENU_ITEM {command = ECmdCopy; txt="Copy";},
                      MENU_ITEM {command = ECmdPaste; txt="Paste";}
                      };
          }
```

Sub-menu panes may be specified by a 'cascade' parameter, with the resource name of the sub-menu pane, e.g.

```
MENU_ITEM {command = ESystemOptions; txt="System Options";
cascade=r_system_options_menu;}
```

4.3 Combining menu sections

Menu sections are combined by a MENU_BAR resource. This lists all of the sections that will be combined to form the menu. The menu panes are defined in order from the bottom section to the top sections.

A menu bar is defined as follows:

```
RESOURCE MENU_BAR r_menuapp_menu
    {
    titles=
        {
                      MENU_TITLE { txt="System"; menu_pane=r_system_menu;},
                      MENU_TITLE { txt="App"; menu_pane=r_app_menu;},
                      MENU_TITLE { txt="View"; menu_pane=r_view1_options_menu;},
                      MENU_TITLE { txt="Context"; menu_pane=r_context1_menu;}
                      };
    }
```

The optional 'txt' parameters are for clarification only, and are not visible on the screen at all. They will however still be stored in the resource file.

The default menu bar is referenced by the EIK_APP_INFO resource. This is the menu bar that will be available when the application starts up.

If view architecture is used, then there is the option to have a menu resource for each view, set in the view <get name> resource structure.

4.4 Changing menu sections

The menu sections can be changed at any time within the application by changing the menu bar resource that is used by the application. This can be achieved by calling:

```
iEikonEnv->AppUiFactory()->MenuBar()->
          SetMenuTitleResourceId(MENU_BAR_RESOURCE_ID)
```

This should be done every time one of the sections should change their contents. Therefore, there should be as many MENU_BAR resources defined, as there are possible combinations of menu sections. I.e. a menu bar should be defined for each combination of view and context options.

Note also, that if view architecture is being used, and the view's own menu system is in use, it is this menubar's contents that need switching:

```
iMyView->MenuBar()->
          SetMenuTitleResourceId(MENU_BAR_RESOURCE_ID)
```

4.5 Changing menu items

Individual menu items may be changed whenever the menu is displayed. This allows the application to remove or add menu items in response to the application state.

The application UI should override the following virtual function:

```
void DynInitMenuPaneL(TInt aResourceId,CEikMenuPane* aMenuPane)
```

This is called after any of the sections has been added to the menu, and is called with the resource id of the section that has just been added, and the menu pane object that is being built.

The application UI can also override the virtual function:

```
void DynInitMenuBarL(TInt aResourceId,CEikMenuBar* aMenuBar)
```

This gets called before any sections are added to the menu, and can be used to dynamically change the sections that will be added to the menu. I.e. this could be used to change the resource id of the context menu section for a certain application state.

5. Scroller Service

The Avkon scroller indicator is in the middle of the softkey pane. It is therefore remote from the control being scrolled. Furthermore, softkey panes are created and destroyed by the appearance and disappearance of certain “popped-up” controls. The correspondences between controls that use scrolling and the currently visible scrolling indicator is managed by a system built into CAknEnv, and used by the softkey pane and CEikScrollbarFrame. Simply put, the system provides ScrollbarFrames with a scroller indicator to use *at the point of* the ScrollbarFrame’s construction. Once set, this correspondence is fixed for the life of the scrollbar frame. Thus, the Avkon scroller system depends upon the correct softkey pane to be fully created before the object using the scroller is constructed. (Otherwise, the more recently created softkey pane will sit on top of the scroller indicator that is being driven.)

This mechanism is hidden from the application writer. Indeed, in the majority of cases, the application writer need not worry about scrollers. This section will deal with the points that the application writer does need to know about the scrollers.

5.1 Ensuring Correct Scrolling Control - Scroller Indicator Correspondence

Problems can occur in certain situations, e.g. where softkey panes are created by popped-up controls.

5.1.1 Using Listboxes with CAknPopupList

The following code illustrates the correct ordering of listbox and popup list construction:

```
void CSimpleAppUi::CreatePopupSelectionL(TInt aItems, TBool aTitle)
            {
            CEikTextListBox* list = new(ELeave)CEikTextListBox;
            CleanupStack::PushL(list);
            CAknPopupList* popupList = CAknPopupList::NewL(list,
                        R_AVKON_SOFTKEYS_OK_BACK);          // sofkey pane created
here.
            CleanupStack::PushL(popupList);
            list->ConstructL(popupList,CEikListBox::ELeftDownInViewRect);
            // scroller reference fetched from (now existing) softkey pane here:
            list->CreateScrollBarFrameL(ETrue);
            // Set the visibility of the scrollbars (This code may become
redundant)
            list->ScrollBarFrame()->
                        SetScrollBarVisibilityL(CEikScrollBarFrame::EOff,

            CEikScrollBarFrame::EAuto);
            << Intervening code >>
            popupList->ExecuteLD(); // Popup selection list is put up
            << Intervening code >>
            CleanupStack::Pop();                            //popupList
            CleanupStack::PopAndDestroy();      // list
}
```

6. Application Writing Miscellaneous

6.1 Application Exit and EEikCmdExit

In Avkon applications, there are two ways of leaving an application: the “Exit” menu option and the “Back” softkey. These different exit methods have subtly different behaviours. This section explains how to achieve these behaviours.

There is a general requirement that any application that can be closed, should respond to EEikCmdExit by calling Exit(). There is also a similar requirement that any control which has modal behaviour, e.g. dialogs, menus, popup lists, etc, must respond to an escape key press by dismissing themselves. These two requirements are necessary for the standard implementation of the options menu exit command to work. They are also required by the low memory background application shutter. This holds for all Avkon products.

6.1.1 What to do

Applications and views handle commands from menus and softkeys in their HandleCommandL() functions. HandleCommandL() must be able to handle the command Id EEikCmdExit, which is the signal to exit the current application. In response to this command, the typical action of an app UI would be to call SaveAnyChangesL() and ExitL() on the app UI. Note: If you are writing a view based application, either the views or the app UI should handle this command. They should never both handle it, as this would cause a crash.

The HandleCommandL() function may also receive the command id EAknSoftkeyBack when the “Back” softkey is pressed. The behaviour in response to a “Back” softkey will depend on context. But when exit behaviour is required, it should be implemented in the same way as for EEikCmdExit.

The menus and softkeys are defined in application and view resources. Softkey “Back” should always generate command Id EAknSoftkeyBack. Menu option “Exit” should always generate command Id EAknCmdExit. Note that this is a different command Id from EEikCmdExit, which should not be used directly.

6.1.2 How it works.

The app UI function Exit() will exit the current application. If the application is embedded, control will return to its parent application. In Avkon, we want this behaviour from the “Back” softkey, but “Exit” menu option should also close all of the parent applications too. The machine shutdown process also requires this process of closing the “chain of embedded applications”. The framework provides support for this, which works by sending each of the applications in the chain the EEikCmdExit command Id. Therefore, applications should respond to the EEikCmdExit and EAknSoftkeyBack command by calling Exit(). The framework will trigger this process when it detects the EAknCmdExit command Id. This is why you should use EAknCmdExit command id in your resources, but should respond to the EEikCmdExit command id in HandleCommandL().

6.1.3 Dialog Shutter

Applications' dialogs and popup windows can be shut using a dialog shutter. As dialogs should respond to an escape key and dismiss themselves, the dialog shutter sends up to 50 Escape keys to the application or view as long as a menu or dialog is displayed.

6.2 Embedded applications

6.2.1 Exit and Back

As described in the previous section, “Exit” and “Back” have subtly different behaviours, which show up when using embedded applications. You must apply the rules of the previous section when using embedded applications.

6.2.2 App UI pointers

When you embed an application, all of the applications share the same CEikonEnv instance, but the member function CEikonEnv::EikAppUi always returns the most embedded application UI pointer. So, if you are writing a component (e.g. a control) that needs to access the app UI, you must decide whether you want to access the most embedded app UI, the app UI which was on top at construction time, or the root app UI.

If you always want to access the most embedded app UI, use CEikonEnv::EikAppUi().

If you always want to access the app UI that was the most embedded when your object was constructed, use the CEikonEnv::EikAppUi() function to get the app UI, but then store it in member data and use it from there.

If you want to access the root app UI, you can access it through any app UI pointer by iterating up the list of container app UIs. Here is a code fragment, which does this:

```
CEikAppUi* root = this;
while (root->ContainerAppUi())
           root = root->ContainerAppUi();
```

In any other situation where you require access to a fixed app UI, it may be best to store that app UI as member data.

6.3 INI files

Avkon disables INI file creation by default. This causes applications that try to open the INI file to fail with a “not supported” error. If you want your application to be able to use the INI file, you must override the application classes `OpenIniFileLC()` function to base call `CEikApplication::OpenIniFileLC()`. Here is an example from ClockApp:

```
CDictionaryStore* CClkApplication::OpenIniFileLC(RFs& aFs) const
           {
           return CEikApplication::OpenIniFileLC(aFs);
           }
```

6.4 Document files

Avkon disables document file creation by default when CaknDocument class is uses as base class for application document.. If you need to have document file then you must override document classes OpenFileL(TBool aDoOpen,const TDesC& aFilename,RFs& aFs) function following way:

```
CFileStore* CTestDocument::OpenFileL(TBool aDoOpen,const TDesC& aFilename,RFs&
aFs)
           {
           return CEikDocument::OpenFileL(aDoOpen, aFilename, aFs);
           }
```

6.5 AIF FILES

Each application should own an application information file (or aif file), which is used to contain a bitmap and captions associated with that application. If an application requires different bitmaps for different languages, this is achieved by producing multiple copies of the aif file, each containing the correct bitmap, using the aiftool. Each aif file produced must be localised by saving with the extension aXX where XX is the 2 digit language code associated with the appropriate language. The apparc software has been modified to attempt to load the aif file associated with the current language, selected by the user.

Avkon provides a possibility to associate a short caption with each application. By default this will be identical to the caption found in the aif. However it is possible for application authors to generate a separate caption file for each language, containing a caption and shortcaption. The shortcaption is used in application grid, whereas the long caption is used in application list. The caption file is generated in the same way as a normal resource file. The structure used to create this caption file is defined in apcaptionfile.rh.

For example:

```
#include "tstappcaption.loc"
#include <apcaptionfile.rh>

RESOURCE CAPTION_DATA
          {
          caption=tst_app_caption_string;
          shortcaption=tst_app_short_caption_string;
          }
```

The caption resource file should be named <NAME OF APP>_caption.rss, where <NAME OF APP> is the name of the app. The resource can be built as part of the normal build process by adding the additional line to an application's mmp file:

```
RESOURCE   <NAME OF APP>_caption.rss
```

The compiled resource file will then appear in \system\apps\%appname%\appname_caption.rXX where XX is the 2 digit language code specified in the mmp file.

6.6 Color palettes

Bmconv, the bmp to mbm compiler, has been changed so that the caller can specify a target palette for the conversion. Bitmaps with a target color mode other than color256 are not affected. Bitmaps with a target color mode of color256 are converted using the nokia palette. I.e. the 216 colors in the color cube, and 10 gray shades. The palette file used for this can be found from \epoc32\include\ThirdPartyBitmap.pal.

The target color mode can be set in the application's mmp file:

```
START BITMAP myapp.mbm
HEADER
TARGETPATH \System\Apps\MyApp
SOURCEPATH ..\Data
SOURCE c12 MyPict.bmp // color4k
SOURCE c8 MyOtherPict.bmp // color256
SOURCE 1 MyOtherPict_Mask.bmp // Black&White mask
END
```