Legal Notice

Copyright © 2006 Nokia Corporation. All rights reserved.

Nokia and Nokia Connecting People are registered trademarks of Nokia Corporation. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

The phone UI images shown in this document are for illustrative purposes and do not represent any real device.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.
Contents

1. Introduction ............................................................................................................. 7
   1.1 Purpose and scope ............................................................................................... 7
   1.2 Overview .............................................................................................................. 7
2. S60 application architectures ............................................................................ 8
   2.1 General S60 application structure .................................................................... 8
      2.1.1 Model – view – controller (MVC) pattern ...................................................... 8
      2.1.2 S60 application structure and MVC ............................................................. 9
   2.2 Traditional Symbian OS application architecture .......................................... 11
      2.2.1 Construction .................................................................................................. 13
      2.2.2 Switching views ........................................................................................... 13
      2.2.3 Advanced view handling .............................................................................. 14
   2.3 Dialog architecture ............................................................................................ 14
      2.3.1 Using dialogs for the main view .................................................................... 14
      2.3.2 Mixing dialog and other application architectures ........................................ 15
   2.4 View architecture .............................................................................................. 15
      2.4.1 When to use – application behavior ............................................................. 16
      2.4.2 Construction .................................................................................................. 17
      2.4.3 Deriving view classes ................................................................................... 18
      2.4.4 View resources ............................................................................................. 19
      2.4.5 Switching views ........................................................................................... 19
   2.5 Run-time environment – application startup ................................................. 20
3. Common application framework services ...................................................... 22
   3.1 Menus and command handling ......................................................................... 22
      3.1.1 Menu system in Avkon ............................................................................... 22
      3.1.2 Defining menu commands and sections ....................................................... 22
      3.1.3 Combining menu sections ............................................................................ 23
      3.1.4 Changing menu sections ............................................................................. 23
      3.1.5 Changing menu items .................................................................................. 24
      3.1.6 Handling commands .................................................................................... 24
   3.2 Key events and control stack ............................................................................ 26
4. Overview of S60 UI components ..................................................................... 28
   4.1 Constructing controls from resource files ....................................................... 28
   4.2 Constructing controls dynamically .................................................................... 28
   4.3 Avkon UI components ...................................................................................... 28
      4.3.1 Dialog ............................................................................................................ 29
      4.3.2 Query ............................................................................................................ 30
      4.3.3 Forms ........................................................................................................... 31
## Change History

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. Introduction

1.1 Purpose and scope

This document discusses issues involved in the design and implementation of an S60 C++ application. The reader is expected to have read *Symbian OS: Getting Started with C++ Application Development*, have good knowledge of C++, and have basic knowledge of Unified Modeling Language (UML).

1.2 Overview

This document provides an introduction to the design of S60 C++ applications. Some of the elements of design are beyond the scope of this document; where this is the case, references to other resources are given.

The document begins with a general introduction to the S60 application structure, describes specific S60 controls, and then the remainder of the document considers the general design issues of C++ applications.
2. **S60 application architectures**

The S60 platform adds a User Interface Layer (Avkon) onto the underlying Uikon application framework. Avkon provides a set of UI components and an application framework designed specifically for S60 devices.

This chapter describes the different application UI architectural approaches available in the Avkon. The choice of application architecture will depend on the application complexity, the view navigation and communications requirements, and the screen layout requirements.

Appendix A, “Example architectures in the S60 platform,” represents various applications and recommended architectures.

### 2.1 General S60 application structure

#### 2.1.1 Model – view – controller (MVC) pattern

![Diagram of MVC pattern]

**Figure 1: Logical dependencies in MVC pattern**

MVC is a common design pattern in S60 UI applications.

The application is split into separate logical parts; these encapsulate the different aspects of the whole application. Each part in the application has a specific role.

- **The Model:**
  - Encapsulates the application state.
  - Exposes the application functionality.
  - Notifies the View of changes.
  - Responds to state queries (from the View).
- **The View:**
  - Renders the Model.
  - Receives update notifications from the Model.
  - Sends user input to the Controller.
• The Controller:
  o Defines application behavior.
  o Maps user actions to Model updates.
  o Selects Views for response.

The MVC pattern forces a separation in the application design, allowing the model code to be reused.

2.1.2 S60 application structure and MVC

S60 applications are normally split into two parts, the engine and the UI. This is to aid maintainability, flexibility, and portability.

The application engine, which is also known as the application model, deals with the algorithms and data structures needed to represent the application’s data.

The application UI, also called the app, deals with the on-screen presentation of the application data and the overall behavior of the application.

This section describes the general structure of an S60 application. An application based on the S60 application framework, the Avkon, may be implemented in three typical ways. These architectural adaptations are presented in more detail in Section 2.2, "Traditional Symbian OS application architecture," Section 2.3, "Dialog architecture," and Section 2.4, "View architecture."
Figure 2: Relationship between the classes that make up a typical S60 application

An S60 application consists of application UI and application engine:

**Application UI (app or exe):**

- **CAnkApplication-derived class**
  - Acts as a startup object for the application framework.
  - Is initiated by the application framework (Avkon).
  - Defines application properties.
  - Creates the document class if the application is not yet running.
  - If an instance of the application is already running, switches to that application and exits.

- **CAnkDocument**
  - CAnkDocument-derived class creates the AppUi controller class.
CAknDocument is the base class for application documents. By default, it prevents document file creation and access. This is typical to the majority of Avkon applications.

- CAknAppUi or CAknViewAppUi-derived class acts as the (main) controller.
  - The choice of the base class depends on the application architecture (see Sections 2.2, 2.3, and 2.4).
  - Handles application-wide events such as menu commands, application focus change, and environment changes.
  - Commands the model.
  - Provides accessories for the buttons (command button area, CBA) and status pane.
  - Is typically responsible for switching between views.
  - In a very simple application, this class may contain the application model implementation, but typically the model is a separate engine owned and controlled by this class.
  - In view architecture, part of the controller logic is handled in CAknView-derived classes.

- View is a UI control
  - Displays the application state known by the model.
  - Receives user input.
  - Notifies the controller of relevant events, such as listbox item selection.
  - Typically observes model changes and updates the screen accordingly.

**Application engine**

- Represents the application’s data and state.
- Encapsulates non-UI-dependant functionality, which might be taken and used as is in other Symbian-based UI frameworks than the S60 platform.
- Is often implemented in its own class library.
- Typically uses non-UI-dependant APIs such as networking, databases, and so on.
- Is owned by the document or directly by the AppUi.

### 2.2 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.

This is the most flexible approach to application UI construction. Its main disadvantage over view architecture is that it does not use a system-provided view management system. This is also its main advantage, because the view architecture view management system is quite restrictive.
Figure 3: Traditional S60 application structure and base classes

Typical responsibilities in this architecture are:

- **CAknAppUi-derived class acts as a controller.**
  - It creates **CCoeControl-derived views** (see Chapter 5, “UI controls and views”).
  - To enable key event handling, it places view(s) on the control stack.
  - To switch between views, it creates and destroys or shows and hides the views.
  - Handles menu commands.
  - Receives environment change notifications from the run-time environment.

- **CCoeControl-derived view**
  - Shows application data and state on the screen.
  - Receives user input.
  - Notifies the **CAknAppUi-derived class** of relevant events.
  - Often observes model changes (directly or via AppUi) and updates the screen accordingly.

- **CAknApplication and CAknDocument-derived classes and the engine have the responsibilities described in Section 2.1.2, “S60 application structure and MVC.”**
2.2.1  Construction

The initial view is created on application startup in ConstructL of CAknAppUi-derived class. Firstly, the window-owning view (see Section 5.1.1, “Window-owning controls”) is created, and to enable key event redirection to the view, the view is then placed on the control stack:

class CMyAppUi : public CAknAppUi
{
  ...
  private: // data
  ...
  CCoeControl* iCurrentView;
};

#ifndef KEnableSkinFlag
#define KEnableSkinFlag 0x1000
#endif
#ifndef KLayoutAwareFlag
#define KLayoutAwareFlag 0x08
#endif

void CMyAppUi::ConstructL()
{
  #ifdef __SERIES60_30__
    BaseConstructL(EAKnEnableSkin);
  #else
    BaseConstructL(KEnableSkinFlag | KLayoutAwareFlag);
  #endif
  iCurrentView = CMyAppMainView::NewL(ClientRect());
  AddToStackL(iCurrentView); // to enable key events
}

2.2.2  Switching views

If the application contains multiple views and they need to be switched, it can be done in various ways. One of the common ways is to delete the current view, create a new view, and add the new one to the control stack. Here CMyAppUi has a custom method to change a view. Often a more generic approach should be favored, but for simplicity the basic idea could be as follows:

void CMyAppUi::ChangeToView2L()
{
  CCoeControl* newView = CMyAppView2::NewL(ClientRect());
  if(iCurrentView)
  {
    RemoveFromStack(iCurrentView);
    delete iCurrentView;
  }
  iCurrentView = newView;
  AddToStackL(iCurrentView);
}

Even if this architecture provides the most flexible way of manipulating different views, view architecture should still be considered. Among other things, it provides a ready framework for doing the essentials in view switching.
2.2.3 Advanced view handling

More complex view handling can be achieved using this architecture — for example, prior to S60 3rd Edition Web and Messaging applications have been 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 multiclent access to the application’s data, which is a common requirement when view switching is needed. The client/server approach is also the recommendation for inter-application communication in S60 3rd Edition.

2.3 Dialog architecture

Dialog architecture refers to a case where the dominant model for views in the application is that of dialogs. One particular case example of this is when the main view is run as a dialog. Multipage dialogs can be used to give a set of views in conformance with the S60 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 into its multipage dialogs. This contrasts with the traditional and view architectures, for which the application must drive the tabs, that is, implement code for switching tabs and updating the navigation pane.

2.3.1 Using dialogs for the main view

When the dialog architecture is used for the main view, it is recommended that the dialog is run as a modeless dialog. This ensures the framework works as it is supposed to.

2.3.1.1 Resources

Using a dialog as the main view will usually require use of the entire available client rectangle. It will also often be a multipage dialog.

The following example 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 modality of the dialog is set to modeless with the EEikDialogFlagModeless flag. Further flags are set to fill the client rectangle, suppress the title bar, and use the CBA buttons (softkeys).
2.3.1.2 **Constructing and running**

Constructing and running a dialog is quite straightforward:

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

Because the dialog is modeless, `ExecuteLD()` returns immediately after being called. The dialog can (and must) be added to the control stack with the `AddToStackL()` method, because modeless dialogs do 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.

2.3.2 **Mixing dialog and other application architectures**

The dialog-based architecture can be combined successfully with the other architecture types as long as dialog-based views are only used at the end points 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 another application view is switched to (presumably by another application), then the data associated with the open dialog may have to be disposed of.

2.4 **View architecture**

The view architecture 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.

Symbian OS provides a base for view architecture support, but Avkon has modified it from v6.0 to be more suitable for use in S60 applications. Figure 4 illustrates the base classes in a typical application supporting view architecture.
Similar to the traditional approach (Section 2.2), the CAknApplication-derived class acts as a starting point for an application. It creates a document, which in turn creates the AppUi controller class. The base class for AppUi is now CAknViewAppUi. It adds view support to CAknAppUi, which is the base class in the traditional approach. The AppUi owns one or more CAknView-derived views, which themselves do not draw the screen, but contain CCoeControl-derived classes to provide visual representation to the user. The view classes typically act as view-specific controllers — the AppUi still being the main controller.

The App 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 application UI in response to events from the view server. The application 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 resources that are only required while the view is active.

The current view for each application can receive events to tell them when they are moved to the foreground or background. Deactivating views will receive a background event, and activating views will receive a foreground event.

Navigation pane 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 the inter-view navigation can be arranged to be hierarchical.

### 2.4.1 When to use – application behavior

The use of views to represent display pages in applications is suitable for applications that do not publish views for use by external applications, or that can handle interruptions by external applications. Applications that use the view architecture 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 applications 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 (for example, save or discard data). However, if
an external view switch can interrupt an application, it must be able to save the 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 the 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 a receiving application to carry out some task.

The view management system only allows one view to be active in each application. If a new view is switched to within an application, the current view is immediately deactivated, so its use is not appropriate in the following cases:

- Applications with any view that cannot 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 (for example, using a Web control inside an e-mail viewer to show an e-mail with HTML content).
- Applications that might want multiple instances running at the same 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.

### 2.4.2 Construction

The instances conforming to CAknView are normally 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:

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

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

    CMyViewArchAppView2* view2 =
        new(ELeave) CMyViewArchAppView2;
    CleanupStack::PushL(view2);
    view2->ConstructL();
    AddViewL(view2); // Transfer ownership to CAknAppUi.
    CleanupStack::Pop(); // view2.

    SetDefaultViewL(*view1);
    // More code
}
```
For the views to be useful and interactive (a view has no drawing capabilities of its own) they will need to hold containers derived from MCoeControlObserver and 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;
}

2.4.3 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.

2.4.3.1 DoActivateL

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

2.4.3.2 DoDeactivate

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

2.4.3.3 HandleForegroundEventL

This function will only be called while the view is active (that is, in between calls to DoActivateL() and DoDeactivate()). When the view comes to the foreground, it will receive HandleForegroundEventL(ETrue). When the view is removed from the foreground, it will receive HandleForegroundEventL(EFalse). This function will only be called when the foreground state actually changes. Note that it may be called a number of times during the view’s active period, as the owning application comes to and goes from the foreground. This may be used for setting focus or controlling screen updates.

2.4.3.4 HandleCommandL

This will be called when the view’s menu generates a command. Command handling is explained in Section 3.1, “Menus and command handling.”
2.4.3.5  HandleStatusPaneSizeChange

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

Here is the typical order of events that a view will receive over an active period:

1. DoActivate()
2. HandleForegroundEvent(ETrue)
3. HandleForegroundEvent(ETrue)
4. DoDeactivate()

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

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

2.4.4  View resources

To let a view have its own CBA or menu, create an AVKON _ VIEW resource, and then pass the resource ID into the view’s BaseConstructL() function. Note that this is only appropriate if views are used 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 menu bar resource given, the view is likely to want to use the application’s menu bar.

2.4.5  Switching views

2.4.5.1  Local switching

Local switching, that is, switching within an application, is achieved by specifying the UID of the view to switch to.

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

Each view may have its own menu system as defined in the AVKON _ VIEW resource structure. However, if the menu system owned by the application is to be used, its 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
iAvkonViewAppUi->ActivateLocalViewL(TUid::Uid(2));
```
2.4.5.2 Remote switching

In remote view switching, an application initiates the remote target application to change its view. If the application is not running, it is started.

In the application, call the CCoeAppUi::ActivateViewL() function, giving TVwsViewId containing the target application’s UID and the target view UID.

Optionally, if the remote application view requires, give a message ID and the descriptor data, which the view needs for its initialization. The remote application view documentation publishes the required parameters.

Note that the calling and target application are independent of each other. When the target application exits, it will not exit the calling application. Also, when the calling client exits after switching the remote view of the target application, it will not exit the target application.

Remote view switching is a one-way request through the application framework from a calling application to a target application. The caller can send data, but the target application cannot pass data back through the framework. Of course, the calling application may pass a file path, which could be used for data exchange.

2.4.5.3 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 he/she was just in. If the application had no previous view, it will exit. If the application’s previous view was the same as the activating view (that is, 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 the view can be reactivated (that is, 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.

2.5 Run-time environment – application startup

In S60 3rd Edition, every application is an executable (.exe). They have their own exe-specific properties such as stack size and SID (secure identifier of an executable; see Section 6.10, “Platform security”).

The application is started in E32Main() by providing the application’s factory function to RunApplication() of EikStart (from eikstart.h):

GLDEF_C TInt E32Main() // exe entry point
{
    return EikStart::RunApplication(NewApplication);
}

Internally, this builds up the run-time environment by creating the Eikon environment (CEikonEnv) and process (CEikProcess) object. The process creates an instance of CApaExe, which calls the factory function.
Prior to S60 3rd Edition, the applications were polymorphic dlls, using the naming convention `<appname>.app`. They were started by application code or typically from the application shell, which created a process, loaded the dll, and created an application instance using the exported `NewApplication()`. The `E32Main()` entry point seen in current EXEs was essentially `E32D11()`, which had no special use:

```cpp
EXPORT_C CApaApplication* NewApplication() // .app factory function
{
    return new CMyHelloApp;
}
```

```cpp
GLDEF_C TInt E32D11(TD11Reason) // Dll entry point
{
    return KErrNone;
}
```

The factory function, named `NewApplication` in the examples, creates a `CAknApplication`-derived application class instance. The instance implements `CreateDocumentL()`, which is called by the process. `CreateDocumentL` constructs the application document.

Finally, the Eikon environment creates the `AppUi`, which in turn creates the views. The final structure of the application was presented in Section 2.1.2, “S60 application structure and MVC.”

Figure 5 gives an overview of how the S60 application code is part of its run-time environment.

**Figure 5: Application run-time environment**

Application code (application, document, AppUi, views) runs in conjunction with the Eikon environment and Avkon framework. Eikon (or the underlying control environment, CONE) handles interactions with the window server at a lower level, while Avkon provides a simplified UI framework for the application.
3. **Common application framework services**

3.1 **Menus and command handling**

3.1.1 **Menu system in Avkon**

An S60 menu is typically shown when the user presses the Options softkey (CBA button). Options menus are constructed from menu bar and menu pane resources specified in the resource file (.rss). The menu bar contains one or more menu panes, each of which contains menu items, that is, menu commands.

![Figure 6: Messaging application with Options and Exit softkeys and its Options menu opened. The Create message menu item has a cascading menu pane.](image)

The framework opens the menu in response to the Options softkey (or the F1 key on the emulator), when bound using the EAknSoftkeyOptions ID. Application views wishing to use the pre-existing Options softkey pane should use the R_AVKON_SOFTKEYS_OPTIONS_BACK CBA resource.

Once the user selects a menu item, the Avkon framework calls the applicable command handler depending on the application architecture being used. See Section 3.1.6, “Handling commands.”

3.1.2 **Defining menu commands and sections**

Each section of the menu bar 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, and Context.

A menu pane is defined as follows:

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

Version 2.0 | April 24, 2006
The command IDs should be defined in one common place, the .hrh file, which both the resource file and application code will include.

Submenu panes can be specified by a `cascade` parameter, with the resource name of the submenu pane, for example.

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

### 3.1.3 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:

```plaintext
RESOURCE MENU_BAR r_menuapp_menu_bar
{
    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:

```plaintext
RESOURCE EIK_APP_INFO
{
    menubar = r_aknexform_menu;
    cba = R_AVKON_SOFTKEYS_OPTIONS_BACK;
}
```

If view architecture is used, there is the option to have a menu resource for each view, set in the view resource structure:

```plaintext
RESOURCE AVKON_VIEW r_multiviews_view2
{
    menubar = r_multiviews_menuBar2;
    cba = R_AVKON_SOFTKEYS_OPTIONS_EXIT;
}
```

### 3.1.4 Changing menu sections

Changing the menu bar resource that is used by the application can change the menu sections at any time within the application:

```c
iEikonEnv->AppUiFactory()->MenuBar()->
    SetMenuTitleResourceId(MENU_BAR_RESOURCE_ID);
```
This sets the whole menu bar and should be performed every time one of the sections needs to change its contents. Therefore, there should be as many MENU_BAR resources defined as there are possible combinations of menu sections. That is, a menu bar should be defined for each combination of view and context options.

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

```c++
iMyView->MenuBar() ->
    SetMenuTitleResourceId(MENU_BARRESOURCE_ID);
```

### 3.1.5 Changing menu items

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

The application UI (in view architecture the view) should override the DynInitMenuPaneL() function:

```c++
void CMyAppUi::DynInitMenuPaneL(TInt aResourceId, 
    CEikMenuPane* aMenuPane) 
{
    if(aResourceId != R_SYSTEM_MENU_PANE) 
        return; // in this example only interest in system pane

    // First hide all
    aMenuPane->SetItemDimmed(ECmdCut, ETrue);
    aMenuPane->SetItemDimmed(ECmdCopy, ETrue);
    aMenuPane->SetItemDimmed(ECmdPaste, ETrue);

    // Show according to application state
    if(...) 
        { 
            aMenuPane->SetItemDimmed(ECmdCut, EFalse);
            aMenuPane->SetItemDimmed(ECmdCopy, EFalse);
        }
    else 
        { 
            aMenuPane->SetItemDimmed(ECmdPaste, EFalse);
        }
}
```

This method is called by the framework after each 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:

```c++
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; that is, it could be used to change the resource ID of the context menu section for a certain application state.

### 3.1.6 Handling commands

Commands are initiated by the Avkon framework in response to the user’s Option menu selection and to specific system-wide events. The AppUi or the object with which the menu is associated will be informed of the command via the
The `HandleCommandL()` method. The handling control is typically `AppUi` or the currently active view, depending on the application architecture being used.

### 3.1.6.1 AppUi

In the traditional application architecture, the `AppUi`'s `HandleCommandL()` method gets called:

```cpp
void CContainerAppUi::HandleCommandL(TInt aCommand)
{
    switch (aCommand)
    {
    case EEikCmdExit:
        {
            Exit();
            break;
        }
    case ECmdCut:
        {
            // implementation of cut operation
            break;
        }
    ...
    default:
        break;
    }
}
```

The custom command IDs (here `ECmdCut`) are shared with resource files and therefore defined in `.hrh` files. Avkon command IDs like `EAknSoftKeyBack` are defined in `avkon.hrh`. It is standard behavior that the application exits in response to the system exit command, `EEikCmdExit` (see Section 6.3, “Application exit and `EEikCmdExit`”).

In normal S60 applications it is standard behavior that the `Exit` softkey exits the application from the main screen and `Back` returns from the subviews, if the application has such.

### 3.1.6.2 View

In the view architecture it is the view's `HandleCommandL()` method that gets called. In a preferred design, the view should handle all view-specific commands, but forward application-wide command handling to the application UI:

```cpp
void CMyAppView1::HandleCommandL(TInt aCommand)
{
    switch (aCommand)
    {
    case EMyAppCmdSwitchToView2:
        AppUi() -> ActivateLocalViewL(KView2Id);
        break;
    case ECmdCut:
        {
            // Implement cut operation
            break;
        }
    // ... Other view-specific command handling here
    case EAknSoftkeyBack:
        {
            ((MEikCommandObserver*)AppUi()) -> ProcessCommandL(EEikCmdExit);
            break;
        }
    default:
        //...
In the example, CMYAppView1 represents the main view of the application. As standard behavior, in the application’s main screen the application should react to the back button (here tied to EAknSoftkeyBack) by exiting the application (useful when the application is embedded). This is done by generating the EEikCmdExit command to AppUi. Additionally, any application-wide commands are forwarded to AppUi.

3.2 Key events and control stack

Key events are typically initiated from the keyboard by user interaction. Key presses are interpreted by the FEP (front-end processor), which ties them to key events. For example, pressing button ‘2’ twice in certain devices and with a certain active FEP could generate a key event holding the key code for letter ‘b’. Special key events are captured by the application framework; for example, the call termination key will exit the application (prior to S60 2nd Edition, FP3 it moved the application to the background). Finally, key events are delivered to the application through the application framework.

The application framework delivers key events to objects placed in the control stack. The AppUi is the default handler for key events, but typically CCoeControl-derived UI controls (see Chapter 5, “UI controls and views”) should receive the events in the first place, and thus be placed on the control stack. The controls are typically added to the control stack within the AppUi, with the AddToStackL() method. In the traditional and dialog architecture this is done in the AppUi construction phase (see also Section 2.2.1, “Construction”):

```cpp
void CMyAppUi::ConstructL()
{
    BaseConstructL();
    iCurrentView = CMyAppMainView::NewL(ClientRect());
    AddToStackL(iCurrentView); // to enable key events
}
```

After this, iCurrentView is on the top and the AppUi next in the control stack. When a key event occurs, the framework first requests iCurrentView to handle it. If it does not consume the event, the framework calls AppUi’s HandleKeyEventL().

In the view architecture the stack push code is typically implemented in DoActivateL() when UI controls are constructed:

```cpp
void CMyView::DoActivateL(const TVwsViewId&, TUid, const TDesC8&)
{
    if(!iUiControl) // iUiControl is CCoeControl-derived UI ctrl
    {
        iUiControl = CMyUiControl::NewL(this, ClientRect());
        AppUi()->AddToStackL(*this, iUiControl); // to ctrl stack.
    }
}
```

When the controls do not want to receive key events anymore, they are removed by a call to the AppUi’s RemoveFromStack() method (see the example in Section 2.2.2, “Switching views”). In the view architecture, the UI control is often removed from the stack in DoDeactivate():

```cpp
void CMyView::DoDeactivate()
{
    if(iUiControl)
    {
        AppUi()->RemoveFromStack(iUiControl);
    }
}
```
UI Controls on the control stack are notified of key press events by the control framework calling their OfferKeyEventL(). This happens in the order they are in the control stack, lastly added being the first being called.

The default implementation of OfferKeyEventL() does not consume the event. To enable key event handling, it must be overridden:

```cpp
TKeyResponse CMyUiControl::OfferKeyEventL(const TKeyEvent& aKeyEvent, TEventCode aType)
{
    TKeyResponse response = EKeyWasNotConsumed;
    // pass key press events to the listbox. It will typically
    // consume Up, Down, and Selection keys.
    if (aType == EEventKey && iListBox)
    {
        response = iListBox->OfferKeyEventL(aKeyEvent, aType);
    }
    return response;
}
```

In the example code, CMyUiControl is a window-owning CCoeControl-derived UI control, which owns one listbox control. CMyUiControl passes the key event to the iListBox and lets it handle it. The listbox is interested in key events as it consumes Up, Down, and OK keys.

In general, if a control is interested in a key event, it returns EKeyWasConsumed; if not, it will return EKeyWasNotConsumed. If the container or one of its components does not consume the event, the framework will pass it on to the next control in the control stack. Finally, if none of the controls in the stack handle the key event, AppUi will handle it in HandleKeyEventL.
4. Overview of S60 UI components

This chapter considers some commonly used Avkon controls and provides examples of their construction. The components considered are:

- Dialog customizable containers
- Query for simple data (text and numeric) entries
- Form for containing controls
- Pop-up Notes for flashing up important information
- Listbox and Grid for displaying data

4.1 Constructing controls from resource files

Resource files are used for defining user interface components and text strings. The advantage of using resource files is that it makes source code shorter and simpler. Resources should also be used for all strings that are displayed, enabling the text to be loaded only when needed, thus saving memory; this also makes it easier to implement localization (see also Section 6.8, “Internationalization and localization”).

Each resource statement has the form shown below:

```
RESOURCE struct-name [resource-name]
{
  resource-initializer-list
}
```

`struct-name` is the name of a struct defined in a `STRUCT` statement.

`STRUCT` definitions occur in `.rh` files (for example, Avkon resource structures are defined in \epoc32\include\avkon.rh).

The examples in the following sections demonstrate the use of resource files in defining controls and the construction of the controls within code.

4.2 Constructing controls dynamically

Controls can be created and/or modified within code; consult the S60 SDK documentation for details concerning a control’s APIs.

4.3 Avkon UI components

All of the Avkon controls considered below, except dialogs, are non-window-owning controls, which means that they cannot exist on the screen without being inside a container window (see Section 5.1, “Window-owning controls and non-window-owning controls,” for a more detailed explanation).

In the examples below, a dialog is used to hold the components; this is typically defined within a resource file. The application developer generally uses a dialog or creates a blank view /container control (derived from `CCoeControl`), which will hold the Avkon controls. Further information on the usage of the Avkon controls is provided in S60 Platform: Avkon UI Resources [1].
Avkon or Avkon-derived components should be favored over custom CCoeControl-derived controls. Avkon UI components scale automatically according to screen resolution and orientation.

### 4.3.1 Dialog

A dialog is a modal control typically used for collecting user data. The dialog shown consists of a label and an editor. A dialog will generally have an OK button and a Cancel button.

![Time dialog in the Clock application](image)

**Figure 7: Time dialog in the Clock application**

Dialogs are objects of the CAknDialog (AknDialog.h) class; many other controls use dialog base classes and therefore are used in a similar manner.

A dialog will generally be defined in resources and can then be used as required within application code as illustrated in the code snippets below.

**Defining a dialog in resources:**

```c
RESOURCE DIALOG r_aknexgrid_dialog_fillingorder
{
    title = qtn_aknexgrid_title_dialog_fillingorder;
    buttons = R_AVKON_SOFTKEYS_OK_CANCEL;
    flags = EEikDialogFlagWait;
    items =
    {
        DLG_LINE
        {
            type = EEikCtLabel;
            id = EAskExGridDialogFillingOrderLabel1;
            control = LABEL
            {
            },
        },
        DLG_LINE
        {
            type = EEikCtNumberEditor; //EAskExGridDialogFillingOrderPrimaryOrientation
            prompt = qtn_aknexgrid_text_dialog_fill_vert_horiz;
            id = EMyGridDialogFillingOrderPrimaryOrientation;
            control = NUMBER_EDITOR //AVKON_INTEGER_EDWIN
            {
                min = EAskExGridDialogFillingOrderVert;
                max = EAskExGridDialogFillingOrderHoriz;
            },
        }
    }
};
```
The dialog above contains a label and a numeric editor.

4.3.1.1 Constructing and displaying a dialog

CCAknExEditorDialog* dialog =
   new (ELeave) C knExEditorDialog();
dialog->ExecuteLD(R_AKNEXGRID_DIALOG_FILLINGORDER);

Dialogs are typically constructed and executed within the same block of code. The dialog should not be placed on the cleanup stack when ExecuteLD() is called; ExecuteLD() guarantees that if it leaves, the dialog will be deleted. ExecuteLD() returns once the user has exited the dialog with the reason why the dialog was dismissed, typically ‘OK’ or ‘Cancel’.

4.3.2 Query

A query is a simplified dialog containing one or two editors. Queries provide a convenient way of getting data from the user. They have an OK and a Cancel button.

There are two types of queries: data entry queries for names, telephone numbers, and so on; and code queries, in which the text is replaced with a star.

RESOURCE DIALOG r_aknexquery_code_query
{
   flags = EGeneralQueryFlags;
   buttons = R_AVKON_SOFTKEYS_OK_CANCEL;
   items =
   {
      DLG_LINE
      {
         type = EAknCtQuery;
         id = EGeneralQuery;
         control = AVKON_DATA_QUERY
         {
            layout = ECodeLayout;
            label = qtn_aknexquery_code_label_text;
            control = SECRETED
            {
               num_letters = 8;
            },
         },
      },
   },
In the above example, the Secret Editor Query, which contains a Secret Editor, resides inside the dialog.

4.3.3 Forms

Forms offer the ability to handle data in a more complex way.

![Figure 9: A form in edit mode](image)

A form is similar to a dialog. It can contain other user interface elements, for example editors, pop-up lists, sliders, and so on. A form has a view and an edit mode; it is useful for user data that requires persistence. For example, a Contacts application could use a form to allow the user to view contact details and easily edit them if required. Forms can also consist of multiple pages, which can be used to group relevant controls. The navigation key (left/right) can be used to switch between pages.

4.3.3.1 Using forms

Forms can be defined in resources. The form resource structure is defined in `epoc32\include\eikon.rh`.

Typically, a form will be derived from `CAlknForm` and the methods `SaveFormDataL()` and `DoNotSaveFormDataL()` will be implemented by the developer. These are called when users switch from edit mode to view mode and are prompted by the form as to whether they wish to save their modifications.

4.3.3.2 Defining a form from resources

```cpp
RESOURCE DIALOG r_aknexform_text_field_dialog
{
    flags = EEikDialogFlagNoDrag | EEikDialogFlagFillAppClientRect | EEikDialogFlagNoTitleBar | EEikDialogFlagNoBorder |
            EEikDialogFlagCbaButtons;
    buttons = R_AVKON_SOFTKEYS_OPTIONS_BACK;
    form = r_aknexform_text_field_form;
}
```

```cpp
RESOURCE FORM r_aknexform_text_field_form
{
```
Again the FORM structure is contained within the DIALOG structure. The form given in the example consists of two editors.

4.3.3.3 Constructing and displaying the form

CAknExFormAdd* form = new(ELeave) CAknExFormAdd;
CleanupStack::PushL(form);
form->ConstructL();
CleanupStack::Pop();
form->ExecuteLD(R_AKNEXFORM_TEXT_FIELD_DIALOG);

The form is pushed on the cleanup stack before ConstructL() (a potentially leaving method) is called, and popped off before ExecuteLD() is called. See Section 4.3.1, "Dialog," for further details.
4.3.4 Pop-up list

Pop-up lists are so named because they pop up at the bottom of the screen. Typically they have an OK and a Back button.

Figure 10: Pop-up list

4.3.4.1 Pop-up notes

A pop-up note is a screen that shows various kinds of messages to the user. Depending on the type of the pop-up note, it may appear for a fixed duration or be modal.

Figure 11: Pop-up note

Pop-up notes have a variety of types: confirmation note, information note, warning note, and error note.

The following is an example code for creating and showing an information note:

```cpp
CAknInformationNote* dialog = new(ELeave)CAknInformationNote();
TBuf<64> aDes;
iCoeEnv->ReadResource(aDes, R_AKNEXNOTE_INFORMATIONNOTE_TEXT);
dialog->ExecuteLD(aDes);
```
4.3.5 Listboxes and grids

There are a wide range of list types available: single line, double line, heading, numbered, and markable. A list can include graphics.

Figure 12: Listbox view in the Menu application

Figure 13: Grid view in the Menu application

Lists and grids are good for displaying a menu of choices; they are used by the S60 Menu application as shown in Figure 12 and Figure 13. For examples on how to define and construct lists and grids, see AknExList and AknExGrid included in the S60 2nd Edition FP3 SDK.

4.3.6 Scroll pane

With any list, grid, or other component that can be scrolled vertically, a scroll bar appears on the right side of the component (from S60 3rd Edition onwards). The scroll bar is also displayed with pop-up components.
Prior to S60 3rd Edition, the Avkon scroller indicator has been in the middle of the softkey pane. It is separate 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 are managed by a system built into CAknEnv, and used by the softkey pane and CEikScrollbarFrame. Simply put, the system provides scroll bar frames with a scroller indicator to use at the point of the scroll bar frame's construction. Once set, this correspondence is fixed for the life of the scroll bar frame. Thus, the Avkon scroller system depends on 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. The following section will address the points that the application writer needs to know about scrollers, prior to S60 3rd Edition.

4.3.6.1 Ensuring correct scrolling control - scroller indicator correspondence

Problems can occur in certain situations, for example, where softkey panes are created by popped-up controls. The following code illustrates the correct ordering of listbox and pop-up list construction:

```c++
void CSimpleAppUi::CreatePopupSelectionL(TInt aItems, TBool aTitle)
{
    CEikTextListBox* list = new(ELeave)CEikTextListBox;
    CleanupStack::PushL(list);
    // Softkey pane (CBA) is created here.
    CAknPopupList* popupList = CAknPopupList::NewL(list, R_AVKON_SOFTKEYS_OK_BACK);
    CleanupStack::PushL(popupList);
    list->ConstructL(CEikListBox::ELeftDownInViewRect);
    // Scroller reference fetched from (now existing)
    // softkey pane here:
    list->CreateScrollBarFrameL(ETrue);
    // Set the visibility of the scroll bars (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.
}

4.3.7 Status pane

Avkon applications have a status pane. In traditional orientation, the status pane is the area on the upper part of the screen. The visible content of the status pane depends on its layout. In normal layout (see Figure 15) the status pane consists of the following subpanes:

- Signal – signal strength
- Context – application icon
- Title – application title (“Tst” in Figure 15)
- Small indicator – new mail, messages, missed call icons, etc., appearing in the upper-right corner of the status pane (no icon in Figure 15)
- Navigation – tabs, here “View1” and “View2”

![Figure 15: Application having two tabs (View1 and View2)](image)

Applications can define and modify only the Context pane, Title pane, and Navigation pane. The application framework sets these automatically on application startup, but they can be changed dynamically in the code. However, if the default pane content is to be overridden, it is done in the application’s resource file, in EIK_APP_INFO:

```
RESOURCE EIK_APP_INFO
{
    status_pane = r_tst_status_pane;
}
```

```
RESOURCE STATUS_PANE_APP_MODEL r_tst_status_pane
{
    panes = 
```
This resource definition will produce a status pane that contains two tabs, seen in Figure 15. The application framework sets the Context pane and Title pane, because they were not given.
5. **UI controls and views**

A control is an area of the screen that may respond to user input events. Key event handling was discussed in Section 3.2, “Key events and control stack”; this chapter concentrates on the visual aspects and compound control hierarchy.

Chapter 4, “Overview of S60 UI components,” presented some of the concrete ready-to-use UI components in the S60 platform. Application developers can also implement their own concrete controls by deriving them from `CCoeControl`.

5.1 **Window-owning controls and non-window-owning controls**

![Control hierarchy for a simple example dialog](image)

**Figure 16: Control hierarchy for a simple example dialog**

5.1.1 **Window-owning controls**

Window-owning controls are typically top-level controls that contain other controls; normally they occupy the whole of the area available for drawing.

The following is an example of how to construct a window-owning control:

```cpp
CMyControl::ConstructL()
{
    CreateWindowL(); // This makes the control window-owning
    SetRectL(ClientRect()); // This sets the control’s size
    ActivateL(); // This must be called before the control can be drawn
}
```

`ClientRect()` returns the screen area available to the application for drawing. This does not include the space that is reserved for the status/control panes.
5.1.2 Non-window-owning control

A non-window-owning control must be contained within a window-owning control. It is faster and requires fewer resources than a window-owning control. When constructed, it needs to set its container window.

The following is an example of how to construct a non-window-owning control:

```cpp
CMyControl::ConstructL(CCoeControl* aParentControl)
{
    SetContainerWindowL(aParentControl);
}
```

5.1.3 Compound controls

Grouped controls can be used to position and reuse common combinations of controls within applications. A compound control must provide information about the controls that it contains via two virtual methods:
`CountComponentControls()`, which returns the number of component controls, and `ComponentControl(TInt aIndex)`, which returns each of the controls by index (zero-based).

5.2 Redraw events and view update

UI controls should override the `Draw` function of `CCoeControl` and implement their drawing code there. This is called by the framework whenever the control should be drawn. The following is an example of container control’s drawing code, which fills the invalidated rectangle area with solid gray:

```cpp
void CMyContainerControl::Draw(const TRect& aRect) const
{
    CWindowGc& gc = SystemGc();
    gc.SetPenStyle(CGraphicsContext::ENullPen);
    gc.SetBrushColor(KRgbGray);
    gc.SetBrushStyle(CGraphicsContext::ESolidBrush);
    gc.DrawRect(aRect);
}
```

The control area (or part of it) may have been invalidated for many reasons, such as overlapping windows. The window server knows when the window area should be (re)drawn and it then sends a redraw event. The `Draw` method of correct control(s) in the window gets called through the application framework.

Application developer can also draw the control area manually, for example, when data being displayed by the control has been changed and the control should be redrawn. The `Draw` function should not be called directly; instead, `CCoeControl` provides two functions to initiate drawing:

- `DrawNow`: This will tell the framework to draw the control immediately, synchronously.
- `DrawDeferred`: This will cause the window server to generate a redraw event, asynchronously.

For more information, refer to SDK documentation.
5.3 Scalable UI

The S60 platform has traditionally been assuming a fixed size resolution, 176 x 208 pixels. From 2nd Edition Feature Pack 3 onwards, S60 devices may also use 240 x 320 and 352 x 416 resolutions. In addition to the traditional portrait layout, the platform also supports the landscape mode in the latter new resolutions.

For an application to be compatible with the varying resolutions and orientations, it should support scalability. Essentially this means implementing the drawing code independently of resolution by using:

- UI metrics API: utilities for determining layout properties for UI controls (size, position, and so on)
- Orientation mode API: getting (and setting) the current orientation mode
- Scalable icons API: creating scalable icons (vector graphics)
- Logical font API: getting a suitable font regardless of the current screen resolution

Scalable UI is enabled by default from S60 2nd Edition, Feature Pack 3 onwards. Legacy applications do not work in S60 3rd Edition, but in S60 2nd Edition, Feature Pack 3 it is possible via compatibility mode (for more information, refer to the scalable UI documentation in the SDK).

For further information on developing scalable applications, refer to the Scalable UI Learning Path [6] that is a comprehensive set of documents and code examples introducing the concept of Scalable UI and demonstrating the development of a scalable application UI (for example, using Avkon UI components or various drawing methods).

5.4 Using resources to define custom controls

It is common when creating custom controls to create a resource structure, so that the control can be defined in a resource file. Resource structures are typically defined in a resource header file (.rh). For further details, see S60 SDK documentation and examples.

5.5 Example application using Listbox

The purpose of this example is to show how a control can be incorporated in an application; it will show controls and event handling within S60 applications.

The application presents users with a list from which they can select.

5.5.1 Design

Figure 17 illustrates the structure of the UI components within an application. The application uses the traditional architecture approach, discussed in Section 2.2, “Traditional Symbian OS application architecture.”
This section will consider the AppUi, Container, and Listbox classes. The View is a ListBoxObserver, and it is through this interface (MEikListBoxObserver) that the View is notified of Listbox events.

A list is typically defined by resources. The top-level window will be the container, which will contain the listbox.

The example will cover the following:

- Construction of the Container control and a Listbox.
- The Container control needs to be able to be queried about what it contains; to do this it must implement certain methods.
- The Container control is responsible for dispatching key press events to the controls it contains.
- The Container control is responsible for notifying controls when its size changes.
- Handling Listbox events (the user selecting an item).

### 5.5.2 AppUi construction

The AppUi is responsible for the construction of the View, which takes place in the `ConstructL()` method.

```c++
void CContainerAppUi::ConstructL()
{
    BaseConstructL(EAknEnableSkin); // recommended flag from // S60 2nd Ed FP3
    iAppContainer = new (ELeave) CContainerContainer;
    iAppContainer->SetMopParent(this);
    iAppContainer->ConstructL(ClientRect());
    AddToStackL(iAppContainer); // iAppContainer is window // owning control.
}
```

The AppUi owns the View (container) and constructs it in its `ConstructL()`. The View is also added to the control stack (`AddToStackL()`) — this means the View will receive key events (via `OfferKeyEventL()`). Typically only Views (top-level container controls) are added to the control stack; other controls that require key press events have them dispatched to their `OfferKeyEventL()` by the View.
5.5.3 View and list construction

The View owns the list and is therefore responsible for its construction.

```cpp
void CContainerContainer::ConstructL(const TRect& aRect)
{
    CreateWindowL();
    // create listbox
    iListBox = new(ELeave) CAknSingleStyleListBox();
    TInt resourceId = R_AKNEXLIST_LOOP_SINGLE;
    iListBox->SetContainerWindowL(*this);
    // read resource that defines listbox
    TResourceReader reader;
    CEikonEnv::Static()->CreateResourceReaderLC(reader, resourceId);
    iListBox->ConstructFromResourceL(reader);
    CleanupStack::PopAndDestroy();
    // Creates scroll bar.
    iListBox->CreateScrollBarFrameL(ETrue);
    iListBox->ScrollBarFrame()->SetScrollBarVisibilityL(CEikScrollBarFrame::EOff, CEikScrollBarFrame::EAuto);
    // set container class as an observer of the listbox
    iListBox->SetListBoxObserver(this);
    iListBox->SetObserver(this);
    SetRect(aRect);
    ActivateL();
}
```

CreateWindowL() sets this control to be a top-level control (window-owning control). In this example, the control is also a compound control, that is, it owns and contains other controls (the listbox). SetRect() sets the size of the control, which will be the size of the client rectangle, given in AppUi construction. ActivateL() is called to inform the framework that the control is ready to be drawn.

Avkon lists are defined in the header file aknlists.h.

When the list has been constructed, it needs to be added to the container control (iListBox->SetContainerWindowL(*this)). The data defining the ListBox is read from resource and then a scroll bar is added to the listbox. The scroll bar is optional, but it does provide useful information to the user.

The View is then set as an observer of the ListBox, so it will be notified of ListBox events.

5.5.4 Container control methods

A container control must provide information about how many controls it contains and what those controls are. The framework uses this information to draw the container. The container provides this information by implementing two methods: CountComponentControls(), which returns the number of component controls, and ComponentControl(TInt aIndex), which returns the control at a given index.

For example, for the CContainerContainer example control containing the listbox this becomes:

```cpp
TInt CContainerContainer::CountComponentControls() const
{
    return 1; // return number of controls inside
```
The methods work in conjunction with each other. The framework can query the control as to how many components it has and then request a pointer to each of the component's controls. Controls can be added or removed at run time using these two methods.

### 5.5.5 SizeChanged()

`SizeChanged()` is called by the framework to inform the control that it has changed size. A container is required to call `SizeChanged()` on all its owned component controls.

```cpp
void CContainerContainer::SizeChanged()
{
    // container control resize code.
    if (iListBox)
    {
        iListBox->SetRect(Rect()); // Set rectangle of
            // listbox.
    }
}
```

### 5.5.6 Key events

Section 3.2, “Key events and control stack,” discussed how controls can receive and consume key events. Components register for key press events by being added to the control stack; this was done during AppUi construction. Controls on the control stack are notified of key press events by the control framework calling their `OfferKeyEventL()`.

```cpp
TKeyResponse CContainerContainer::OfferKeyEventL(
    const TKeyEvent& aKeyEvent,
    TEventCode aType)
{
    if (aType != EEventKey)
    {
        return EKeyWasNotConsumed;
    }

    // pass key press events to the listbox. It will typically
    // consume Up, Down and Selection keys.
    if (iListBox)
    {
        return iListBox->OfferKeyEventL(aKeyEvent, aType);
    }
}
In the example code, the key event is passed to the Listbox. If a control is interested in a key event, it returns EKeyWasConsumed; if not, it will return EKeyWasNotConsumed. If the container or one of its components does not consume the event, it will be passed on to the next item in the control stack.

The Listbox is interested in key events as it consumes Up, Down, and OK keys.

### 5.5.7 Listbox events

The specific listbox event of interest is that of an item being selected (EEventEnterKeyPressed).

```cpp
class CContainerContainer : public MEikListBoxObserver

Here the container declares itself as a listbox observer (within the header file); thus it implements HandleListBoxEventL(), as shown below:

```cpp
void CContainerContainer::HandleListBoxEventL(CEikListBox* aListBox, TListBoxEvent aEventType)
{
    if (aEventType == EEventEnterKeyPressed)
    {
        // Do something, e.g. open selected item
    }
}
```  

The container has set itself to be the Listbox's observer (iListBox->SetListBoxObserver(this)) within the Container's ConstructL().

The `HandleListBoxEventL()` method itself could, for example, open the item, when it has been selected.

It should be noted that the observer design pattern is commonly used in S60 applications. It enables an object to notify an object type as opposed to an actual object. This allows greater flexibility, separation of modules, and code reuse when designing applications.
6. Common application issues

6.1 Application registration files

From S60 2nd Edition, Feature Pack 3 onwards applications are presented in the system by registration. Although not necessary, application registration brings the application visible in the application launcher application, and provides capabilities, supported data types, and embeddability information to the system.

The application should provide:

- Non-localized application registration information resource: hiding and embeddability
- Localized part of the application information resource: captions and icons

The name of the non-localized resource file must be `<application_name>_reg.rss`. The following is an example or tstapp's registration resource file, `tstapp_reg.rss`:

```c
#include <appinfo.rh>
UID2 KUidAppRegistrationResourceFile
UID3 0x10005C30

RESOURCE APP_REGISTRATION_INFO
{
    app_file="tstapp";
    localisable_resource_file="\\resource\\apps\\tstapp_loc";
    hidden = KAppNotHidden;
    embeddability = KAppNotEmbeddable;
    newfile = KAppDoesNotSupportNewFile;
}
```

In essence, the registration file defines the application executable and the localization file name base. The corresponding resource file is defined in `tstapp_loc.rss`. The following is an example supporting only the default language:

```c
#include <appinfo.rh>
#include "tstapp_loc.rls" // localized strings for default language

RESOURCE LOCALISABLE_APP_INFO
{
    short_caption = STRING_tstapp_short_caption;
    caption_and_icon =
    {
        CAPTION_AND_ICON_INFO
        {
            Caption = STRING_tstapp_caption;
            //number_of_icons=3;
            //icon_file=STRING_r_tstapp_icon_path;
        }
    }
}
```

If the application does not support localization, the values for the strings and icon paths could be hard coded in the resource file. If localization is to be supported, separate localization `.rls` file for each language should be created and included in the resource file (conditionally for each language).
The registration resource files are taken into the project .mmp file as follows:

```plaintext
sourcepath .
START RESOURCE tstapp_reg.rss
targetpath \private\10003a3f\apps
END

START RESOURCE tstapp_loc.RSS
targetpath \resource\apps
lang sc
END
```

Note that even if the registration file, `tstapp_reg.rss`, is built in the development environment to `\private\10003a3f\apps`, for platform security reasons it must be placed on hardware to `\private\10003a3f\import\apps`.

Prior to S60 3rd Edition, applications have been presented in the system by application information files, `.aif` (S60 2nd Edition, Feature Pack 3 also supports application registration). Essentially, the `.aif` file contains bitmaps and captions associated with the application. Similar to application registration, the application information file is not required, but it provides additional information to the system.

The `.aif` file supports localization for captions. If separate icons are needed for each language, the appropriate language variant must be created and saved as an `.aXX` file, where XX is the two-digit language code associated with the language. When the `.aif` file is accessed, the apparc software loads the correct `.aif` file associated with the language currently selected by the user. The following is an example of a resource file needed in application information file build process:

```plaintext
#include <aiftool.rh>
RESOURCE AIF_DATA
{
    app_uid=0x10005C30;
    caption_list=
    {
        CAPTION
        {
            code = ELangEnglish;
            caption = "MyApp";
        }
    },
    num_icons=2;
    embeddability=KAppNotEmbeddable;
    newfile=KAppDoesNotSupportNewFile;
}
```

Avkon provides a short caption for each application. By default it is identical to the one found in the `.aif` file. It is possible, and common, to provide separate localized short and long captions for the application. These are used, for example, by the “Applications menu” application, when it builds the list (or grid) of applications in the system. This is achieved by introducing a `CAPTION_DATA` resource, placed in `<application_name>_caption.rss`. For example, for the `tstapp` application the resource file is `tstapp_caption.rss`:

```plaintext
#include "tstapp_caption.loc" // application-specific .loc file
#include <apcaptionfile.rh>  // CAPTION_DATA resource definition
RESOURCE CAPTION_DATA
{
    caption = tstapp_caption_string;  // defined in tstapp_caption.loc
    shortcaption = tstapp_short_caption_string;
}
```
The caption data resource and the .aif file can be compiled as part of the application build process by appropriate project .mmp file definitions:

```
RESOURCE tstapp_caption.rss
   AIF tstapp.aif ..\Aif tstappAif.rss
   c12 tstIcon.bmp tstIcon_mask.bmp
   tstIcon_42x29.bmp tstIcon_42x29_mask.bmp
```

Much of the work in creating the required registration files and application information files can be eased by using application wizards. For more detailed information on application registration and localization, refer to SDK and IDE documentation.

### 6.2 Deployment to device

From S60 3rd Edition onwards, while targeting to device, the code is built using either GCCE or ARMV5 target. GNU Compiler Collection tools, which produce the GCCE target, are freely available in the 3rd Edition SDK. The ARMV5 target is produced by ARM’s RealView Compiler Tools. Due to data caging and other platform security issues (see Section 6.10, “Platform security”), the resulting files and their locations in the target hardware are as follows (sample from a package definition file, .pkg):

```
"<SDK>\epoc32\release\gcce\udeb\Tst.exe" -
"!:\sys\bin\Tst.exe"

"<SDK>\epoc32\data\z\private\10003a3f\apps\Tst_reg.rsc" -
"!:\private\10003a3f\import\apps\Tst_reg.rsc"

"<SDK>\epoc32\data\z\resource\apps\Tst_loc.Rsc" -
"!:\resource\apps\Tst_loc.Rsc"

"<SDK>\epoc32\data\z\resource\apps\Tst.Rsc" -
"!:\resource\apps\Tst.Rsc"
```

In the example “!” means a user-selectable drive during installation. The application built result is an executable (.exe), and it is placed in the \sys\bin directory. As discussed in Section 6.1, “Application registration files,” the application is registered by placing the registration file to the import\apps folder of the application launcher.

Prior to S60 3rd Edition, the application binaries have been application dlls (.app). They are compiled using ARM or THUMB target. The resulting files are typically the application dll, application resources, and the application information file. These are packaged into a .sis file from the following locations in the development environment (a .pkg file sample):

```
"<SDK>\Epoc32\release\thumb\urel\Tst.app" -
!:\system\apps\Tst\Tst.app"

"<SDK>\Epoc32\data\z\system\apps\Tst\Tst.rsc" -
"!:\system\apps\Tst\Tst.rsc"

"<SDK>\Epoc32\data\z\system\apps\Tst\Tst_caption.rsc" -
"!:\system\apps\Tst\Tst_caption.rsc"

"<SDK>\Epoc32\data\z\system\apps\Tst\Tst.aif" -
"!:\system\apps\Tst\Tst.aif"
```

Prior to S60 3rd Edition, the .sis files have been generated with the makesis tool (or similar). S60 3rd Edition has the same packaging phase, but the resulting .sis file must also be signed. In a simple test application self-signing is enough; advanced signing has to be done mostly due to specific capabilities (for details, see SDK documentation and document [Testing And Signing With Symbian Platform Security [7]]).
6.3  Application exit and EEikCmdExit

In Avkon applications, having been written according to S60 UI Style Guide [5], there are two ways of leaving an application:

- The ‘Exit’ menu option
- The ‘Back’ softkey

These different exit methods have slightly different behaviors. The following section explains how to achieve these behaviors.

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 that has modal behavior (for example, dialogs, menus, and pop-up lists) must respond to an "escape key" press by dismissing itself. These two requirements are necessary for the standard implementation of the Options menu ‘Exit’ command and low memory background application shutter to work. This holds true for all S60 applications.

6.3.1  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 application UI would be to call SaveAnyChangesL() and Exit() on the application UI. Note that when writing a view-based application, either the views or the application UI should handle this command; they should never both handle it, since this will cause a crash.

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

Menus and softkeys are defined in application and view resources. The Back softkey should always generate command ID EAknSoftkeyBack. The Exit menu option should always generate command ID EAknCmdExit. Note that this is a different command ID from EEikCmdExit, which should not be used directly.

6.3.2  How it works

The application UI function Exit() will exit the current application. If the application is embedded, control will return to its parent application. In Avkon, this behavior is needed from the Back softkey, but the Exit menu option should also close all of the parent applications. The device shutdown process also requires this process of closing the chain of embedded applications. The framework provides support for this closure, which works by sending the EEikCmdExit command ID to each of the applications in the chain. 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 the EAknCmdExit command ID should be used in the resources, but the framework should respond to the EEikCmdExit command ID in HandleCommandL().

Note that Exit() is a leaving function and it must not be trapped (using TRAPD macro). It must leave to the application framework, which happens typically in HandleCommandL(). The application framework will then exit the application.
6.3.3 Dialog shutter

Application’s dialogs and pop-up 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.

6.4 Embedded applications

From the end user’s perspective, application embedding means including an application UI within a host application’s environment.

In S60 2nd Edition, embedded applications are run within a host application’s process.

In S60 3rd Edition, traditional application embedding does not work because applications are run in separate processes. The embeddable applications are known as server applications, which can act as if they were embedded within a host application. In this approach, the application acts in both the application role and the server role. A host is a client for the server, requesting it for services. The desired functionality could be implemented manually by other means, but the server application approach is recommended in S60 3rd Edition.

6.4.1 Server application

A simple server application can be implemented by overriding certain parts in base classes and implementing server functionality. In the application class, inherited from C AkronApplication, override the NewAppServerL() method:

```cpp
void CMyServerAppApp::NewAppServerL(CApaAppServer*& aAppServer)
{
    aAppServer = new(ELeave)CMySimpleServer;
}
```

This creates a custom server instance, which has a similar role as the server in conventional client/server programming. The server creates a session with a calling client (here host application). The session then communicates with the client.

In S60 applications, the server, here CMySimpleServer, should be inherited from C AkronAppServer and override desired functionality, typically CreateServiceL():

```cpp
class CMySimpleServer : public C AkronAppServer
{
public:
    CApaAppServiceBase* CreateServiceL(TUid aServiceType) const
    {
        if(aServiceType.iUid == 0x06C22C0C)
            return new(ELeave) CMySimpleService();
        else
            return CAknAppServer::CreateServiceL(aServiceType);
    }
};
```

CreateServiceL() creates a server-side session, which communicates with the client. When the client starts the server application, it specifies a UID, which is forwarded through the application framework to the CreateServiceL() method.
The server can provide several services, depending on the given UID. In the example code, the server passes all unknown service UIDs to the CAknAppServer base class. It creates a custom CMySimpleService service when the client requests a service identified by UID=0x06C22C0C. This is the main service of the example server application.

In the simplest form, if the calling application and embeddable application do not need to communicate during embedding, the session class implementation may be empty:

```cpp
class CMySimpleService : public CAknAppServiceBase
{
public:
    CMySimpleService () {} 
    virtual ~CMySimpleService() {}  
};
```

However, typically the service overrides the ServiceL() and ServiceError() IPC methods of the base class to communicate with the client. The service is inherited from CAknAppServiceBase.

Note that valid command IDs (sometimes called opcodes) used in the client/server communication must start from RApaAppServiceBase::KServiceCmdBase = 0x200. Lower values are used by the application framework for GUI interaction. This range rule means that if the service overrides ServiceL(), it must forward all unknown command IDs to the base class, that is, call CAknAppServiceBase::ServiceL(...).

### 6.4.2 Client of the server application

The client launches the server application with a RAknAppServiceBase-derived class. If the client does not need to communicate through IPC with the server application, at minimum it must only implement the ServiceUid() method of RAknAppServiceBase:

```cpp
const TUid KUidMyServerAppServiceUid = { 0x06C22C0C }; 
class REmbedderClient : public RAknAppServiceBase 
{
public:
    TUid ServiceUid() const
    {
        return KUidMyServerAppServiceUid; 
    }
    virtual ~REmbedderClient() { Close(); }  
};
```

However, the client class typically uses IPC methods from the base class. The client class should be encapsulated into some class as a member variable. The owner could be CActive-derived, because the exit of an embedded application is tracked asynchronously. There is also a utility class, CApaServerAppExitMonitor, which could be used for the purpose as well. Here the client class and the monitor are members of AppUi, which implements the MAknServerAppExitObserver interface, required by the monitor:

```cpp
CEmbedderAppUi::OpenSimpleServiceEmbeddedL()
{
    iEmbedderClient = new(ELeave)REmbedderClient; 
    iEmbedderClient->ConnectChainedAppL(KUidMyServerApp); 
    iMonitor = CApaServerAppExitMonitor::NewL(*iEmbeddedClient, *this, CActive::EPriorityStandard); 
}
```
The host application calls `ConnectChainedAppL` with the application UID of the server application. The framework launches the application and initiates connection to the server. Then the monitor is attached to track the lifetime of the server application. When the server application exits, the monitor reports it to `MAknServerAppExitObserver`-derived observer:

```cpp
void CEmbedderAppUi::HandleServerAppExit(TInt aReason)
{
    // Custom processing
    MAknServerAppExitObserver::HandleServerAppExit(aReason);
}
```

To handle correct Exit/Back behavior from the embedded application, the observer should call the base class implementation of `HandleServerAppExit` as done in the example code.

### 6.4.3 ‘Exit’ and ‘Back’

As described in Section 6.3, “Application exit and EEikCmdExit,” **Exit** and **Back** have slightly different behaviors, which show up when using embedded applications. The rules of Section 6.3 must be applied when using embedded applications.

### 6.4.4 Application UI pointers

In S60 2nd Edition, when embedding 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. Thus, when writing a component (for example, a control) that needs to access the application UI, a decision must be made as to whether to access the most embedded application UI, the application UI that was on top at construction time, or the root application UI.

If the most embedded application UI should always be accessed, use `CEikonEnv::EikAppUi()`.

If the application UI to access is the one that was on top at the time of construction, use the `CEikonEnv::EikAppUi()` function to get the application UI and store it in member data while constructing the object.

To access the root application UI, access it through any application UI pointer by iterating up the list of container application UIs. The following code fragment does this:

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

In any other situation where access to a fixed application UI is required, it may be best to store that application UI as member data.

In S60 3rd Edition, `CEikonEnv` can be queried if the application was started as embedded (that is, as a server application) by a call to `StartedAsServerApp()`. Since the embedded application runs in a separate process, it cannot access the container UI.
6.5 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. To enable an application to use an .ini file, the application class’s OpenIniFileLC() method must be overridden to call CEikApplication::OpenIniFileLC(). For example:

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

6.6 Document files

Avkon disables document file creation by default when the CAknDocument class is used as a base class for the application document. If a document file is needed, the document class’s OpenFileL function must be overridden in the following way:

```cpp
#ifdef EKA2
void OpenFileL(CFileStore *&aFileStore, RFile &aFile)
{
    CEikDocument::OpenFileL( aFileStore, aFile );
}
#else
CFileStore* CTestDocument::OpenFileL(
    TBool aDoOpen, const TDesC& aFilename, RFs& aFs)
{
    return CEikDocument::OpenFileL(aDoOpen, aFilename, aFs);
}
#endif
```

6.7 Graphics

Scalable vector graphics are supported from S60 2nd Edition, Feature Pack 3 onwards. A separate icons.mk file containing a list of graphic files should be defined. Both vector graphic and windows bitmap files are supported. The makefile is added to the bld.inf file as follows:

```makefile
PRJ_MMPFILES
#ifdef EKA2 // S60 3rd Ed
gnumakefile MyApp_icons.mk
MyApp_3rd.mmp
#else // S60 1st and 2nd Ed
MyApp_2nd.mmp
#endif
```

See [S60 Platform: Scalable UI Support](#) for further information on the icons.mk file structure.

In the S60 platform prior to 2nd Edition Feature Pack 3, the preferred way of using bitmaps is to create multi-bitmap files (.mbm) and access them at run time. A single .mbm file contains one or more bitmaps, each of which is accessed using indexes.

Multi-bitmaps can be created by defining a BITMAP section in the project .mmp file. The following is an example:

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

When the application is built, the tool chain will generate the myapp.mbm multi-bitmap file, which contains three bitmaps. Additionally, because the HEADER keyword was specified, the tool will generate a bitmap header file, which contains the indexes (enumeration) to contained bitmaps. The header is used in the code to point to certain bitmaps within the multi-bitmap.

Bitmaps are often used in the code as a CPbsBitmap object. See SDK documentation for information on how files are loaded and used from the .mbm file.

6.8 Internationalization and localization

The need for internalization (that is, preparing an application to be potentially used virtually everywhere) should be considered from the beginning of a project. This will require a flexible attitude towards layout, because for example texts in different languages are displayed with different lengths, and date and time have different formats.

Localization (that is, adapting an application for a certain locale) should only require changing the language-specific resource file. Each language will require its own compiled resource file. The default compiled resource file has the extension .rsc. Specific languages have the extensions .r01, .r02, .r03, and so on.

The following is a simple example of how localization can be achieved in an S60 project.

The project file (.mmp) contains the language information in the LANG attribute.

The resource file (.rss) includes the localization file (.loc).

The localization file contains the strings for the different languages.

The localization file can include resource file statements; it typically only needs to specify that the encoding of Unicode characters is UTF8.

Example code from a .loc file:

//The Text in source files is encoded in UTF-8
CHARACTER_SET UTF8

// English is the default language(sC) and 01
#ifndef LANG_01 // or sC
#define str_myapp_exit "exit"
#define str_myapp_open "open"
#endif

// French language resources follow
#ifndef LANG_02
#define str_myapp_exit "sortie"
#define str_myapp_open "ouvrir"
#endif

Example code from a localized resource file:

#include "loc file"
...
RESOURCE CBA r_softkeys_open_exit
{
    buttons =
{
To build the French resources, change the .mmp file’s LANG attribute to LANG SC 01 02 03, and then build the resource files. This will build the resource files for default(SC), English (01), French (02), and German (03).

For information regarding the installation of multilingual applications, consult the S60 SDK documentation, section “How to create an installation file for a multilingual application.”

Note the following points concerning localization:

- Do not embed text in C++ source files.
- Do not hard code the size of text buffers.

See also Section 6.1, “Application registration files,” to get an overview of how application registration should take localization into account.

### 6.8.1 System locale

TLocale can be used to access the current locale settings in the system. TLocale and related classes are defined in E32STD.H; they allow Dates, Times, Currency, and many other locale-sensitive data types to be correctly formatted for the current locale.

### 6.9 Context-sensitive help

Application framework supports context-sensitive help. This means that whenever the user requests help (by pressing the help key or from the application menu), the appropriate help page is displayed.

Refer to SDK documentation for information on how the help files are written and compiled (look at “CS-Help Guide”). The process produces a header file, which provides named links to the help topics in the help file. These links are used when referred to from the application code.

The help may be launched manually in the application in response to the custom "Help" menu item. The help could be shown as follows, in the HandleCommandL of AppUi:

```cpp
CArrayFix<TCoeHelpContext>* buf = CCoeAppUi::AppHelpContextL();
HlpLauncher::LaunchHelpApplicationL(iEikonEnv->WsSession(), buf);
```

This will request the application framework to find the current help context. As a first try the application framework requests the top-most CCoeControl-derived UI control to provide the help context. If it does not provide one, the framework requests AppUi for it. Finally, the help application is launched with the correct help item.
To be fully context aware, the application UI controls should provide help context associated with them. They do it by overriding the GetHelpContext method of CCoeControl:

```cpp
void CMyControl::GetHelpContext(TCoeHelpContext& aContext) const
{
    aContext.iMajor = KUidMyApp; // UID of help file, often is
    // the application UID
    aContext.iContext = K_MY_CONTROL_HLP; // From the help header
}
```

Similarly, the application should override the AppUi’s method:

```cpp
CArrayFix<TCoeHelpContext>* CMyAppUi::HelpContextL() const
{
    ...
}
```

### 6.10 Platform security

Platform security is part of the enhanced security model from Symbian OS v9 (S60 3rd Edition) onwards. It ensures that an executable code trying to use an API has rights to use it, and uses it in the manner that it has been granted.

#### 6.10.1 Capabilities

The permissions for an application process are defined as capabilities, entities of protection. Simple GUI applications do not need any capabilities. More functional applications (for example, a Web browser using network, or file system access to specific folders) require specific capabilities.

For more information on capabilities, refer to the document *Testing And Signing With Symbian Platform Security* [7].

#### 6.10.2 Data caging

Platform security provides a way to protect private data. All other folders except system folders are available to any process with any capabilities. The system folders are restricted by the security model:

- `\sys\` – Only accessible to code with TCB capability. Executables, such as applications, are placed in and can only be run from `\sys\bin`.

- `\private\` – Private data for all programs is placed in `\private\<SID>` and is accessible only to the executable in question (SID is a security ID of an executable; often UID3. Refer to SDK documentation). The special subfolder `\private\<SID>\import` is accessible to any process; it is meant for sharing application-specific data, plug-ins, etc. Note that all applications should register to the import directory of application launcher (shell) by adding a registration resource file: `\private\10003a3f\import\apps\<app_name>_reg.rsc`

- `\resource\` – Any process can read data (one with TCB capability can modify; in practice this means software installer). Applications place their resources in the `\resource\apps` folder.

Note that prior to S60 3rd Edition, application data (bitmaps, etc.) has often been placed in the application directory, `\system\apps\<appname>`. It has been common to determine the path by parsing the path from the application’s full name:
TFileName myBitmap;
TParsePtrC parse(iEikonEnv->EikAppUi()->
Application()->AppFullName());
myBitmap = parse.DriveAndPath();
myBitmap.Append(KMyBitmapFilename);

From S60 3rd Edition onwards the application data should be placed in the
application's private folder \private\SID\.

6.11 Global notes and dialogs

Avkon provides various dialogs as noted in Section 4.3.1, “Dialog.” It is important to
notice that while applications typically show dialogs within the application process,
sometimes global dialogs are needed.

Global dialogs appear in the user interface even if the application (or server process)
launching them is not the top-most, visible application. Avkon provides ready-to-use
global dialog components; they start with CAknGlobal*.

Global dialogs and notes are based on the notifier framework. When a dialog is executed
in the code, the framework will request the notifier service to show a certain dialog. This
happens through the RNotifier client interface, to the notifier system server process.
Components in the notifier framework are plug-ins for the notifier service, and they can
be added by the application developer. However, CAknGlobal* classes provide the
most common tasks and should be used in the first place.
7. Further reading

- Forum Nokia Web site: http://www.forum.nokia.com
- S60 Tools
- S60 Platform: C++ Coding Conventions, available at www.forum.nokia.com

Books


Programming for the Series 60 Platform and Symbian OS. DIGIA inc. Wiley.


## 8. Terms and abbreviations

<table>
<thead>
<tr>
<th>Term or abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVKON</td>
<td>S60 extensions and modifications to the Symbian Uikon application framework</td>
</tr>
<tr>
<td>CBA</td>
<td>Command button area. A toolbar controlled by the softkeys.</td>
</tr>
<tr>
<td>FP(x)</td>
<td>Feature Pack (x)</td>
</tr>
<tr>
<td>MVC</td>
<td>Model – View – Controller. MVC is a common design pattern in S60 UI applications.</td>
</tr>
<tr>
<td>UI</td>
<td>User interface</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
</tbody>
</table>
9. References

The following resources are available on the Forum Nokia Web site at www.forum.nokia.com:

[1] S60 Platform: Avkon UI Resources
[3] S60 Platform: Scalable Screen-Drawing How-To
[5] S60 UI Style Guide
[6] Scalable UI Learning Path
Appendix A. Example architectures in the S60 platform

This chapter gives some examples of architecture usage in hypothetical applications. A general assumption is made that only one instance of any application will be running at the same time. However, some parts of each UI may be able to have multiple instances running (for example, settings screens and e-mail viewers).

A.1 App launcher

This is an application (formerly known as the shell) that can launch and switch to other applications. The application has only a single internal view. It does not accept external requests that can interrupt it.

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

A.2 Fast-swap window

This is a pop-up window (formerly known as the task list) that lists all currently running applications, and allows the user to switch to any running application. It is only a sleeping pop-up dialog, not a full application itself. It has no state that can be interrupted. It is best implemented as a dialog application. Existing task list applications work this way.

A.3 E-mail application

The e-mail application allows reading and creation of e-mails. This application has externally usable views (the e-mail editor), internally switchable views (the message viewers), and can do interruptible operations (e-mail editing, etc). Notifier messages can tell the application to show new messages to the user. This means that the application could be editing an e-mail when a notifier asks it to display a message; thus the application must have a mechanism to pick up such requests, and 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 to which external applications can link. The Symbian OS messaging application has a client/server mechanism and interfaces available for launching editors from other applications. Some work will be required to investigate how to launch the editors inside the external application’s process.

A notifier providing the ability to view a new message should not terminate the e-mail application during creation of a new e-mail, so viewing new messages should perhaps be performed as a nested dialog-like component inside the messaging application. Note that the best implementation depends heavily on the exact inter-application interaction model.
A.4 Contacts application

The contacts application allows the viewing, editing, and selection of contacts. Selection of contacts will be available for other applications to use. The contacts application has an externally usable view (the contacts selector). It can also use the e-mail editor from the e-mail application, but it cannot be interrupted by external view switch requests.

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

A.5 Web browser

The Web browser can view Web pages as a full-screen browser application, or can be embedded in another document. The Web browser offers a UI control in a DLL that external applications can use; however, it does not offer any externally usable full screen views. It does allow external requests to view other pages, so it has to be able to deal with potential interruptions to activities such as writing e-mails as a result of a mailto link.

The Web control that is embeddable inside other applications should be implemented as a separate DLL. The main browser could be implemented as a traditional Symbian OS application, using document open requests to open new Web pages. It could also use the view architecture to receive page open requests.

A.6 Settings

This example application handles both global settings and application-specific settings. The settings application and application local settings are separate entities. Although they may share a common library, they will not interfere with one another.

The settings application 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.

A.7 Telephony application

The telephony application provides no external views, but it does have to accept requests from external sources (for example, Contacts). It must be able to decide whether it can handle the external requests as they happen and is probably most straightforward to implement as a traditional Symbian OS application. Note that the application should use a client/server system. Much of the external interaction may go through the ETEL server.
Evaluate this resource

Please spare a moment to help us improve documentation quality and recognize the resources you find most valuable, by rating this resource.