

# Series 80 2nd Edition: Getting Started with C++ Application Development

Version 1.0; June 15, 2004

# Symbian C++

# NOKIA

Copyright © 2004 Nokia Corporation. All rights reserved.

Nokia and Nokia Connecting People are registered trademarks of Nokia Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. 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.

#### **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

<b>1</b>	<b>Overview .....</b>	<b>5</b>
<b>2</b>	<b>What You Should Know .....</b>	<b>6</b>
<b>3</b>	<b>What You Will Need .....</b>	<b>7</b>
<b>4</b>	<b>Background Information and Concepts.....</b>	<b>8</b>
4.1	Uikon .....	8
4.2	Ckon .....	8
4.3	CONE.....	8
4.4	Application Framework on Symbian OS.....	8
4.5	Differences Between Emulator and Target Device.....	9
4.6	Event Handling .....	9
4.7	Starting an Application.....	10
<b>5</b>	<b>Compiling a Sample Application.....</b>	<b>11</b>
<b>6</b>	<b>Application File Details.....</b>	<b>12</b>
6.1	Build Files.....	12
6.1.1	Project definition file .....	12
6.1.2	Component description file .....	13
6.2	Source Files.....	13
6.2.1	Common header files.....	13
6.2.2	Resource definition files.....	14
6.2.3	Global functions.....	15
6.2.4	Application .....	16
6.2.5	Document .....	17
6.2.6	Application UI.....	18
6.2.7	View.....	20
6.3	Installation Files.....	21
6.3.1	Package file.....	21
<b>7</b>	<b>Where to Go Next.....</b>	<b>23</b>
<b>8</b>	<b>Terms and Abbreviations .....</b>	<b>24</b>
<b>9</b>	<b>References .....</b>	<b>25</b>
<b>10</b>	<b>Evaluate This Resource.....</b>	<b>26</b>

## Change History

June 15, 2004	Version 1.0	Initial document release Revision on April 12, 2006: minor editorial changes including terminology update. Nokia 9300 Product ID added in Section 6.3.1.1

# 1 Overview

Series 80 2nd Edition runs on top of Symbian OS and provides application developers with a rich set of APIs to develop versatile and robust smartphone applications. This document shows how to develop a simple application in C++ for Series 80 2nd Edition. The UI layer within Series 80 2nd Edition is called `Clon`. For those familiar with the history of Symbian Device Family Reference Designs, `Clon` was developed from the Symbian Crystal reference design.

## 2 What You Should Know

You should be familiar with C++ programming and object-oriented design principles. It is also recommended that you read [Symbian OS: Getting Started with C++ Application Development](#) [1], located on the Forum Nokia Web site at <http://www.forum.nokia.com/symbian> under **Getting Started**.

### 3 What You Will Need

There are three integrated development environment (IDE) tools that can be used for Series 80 2nd Edition application development: Microsoft Visual Studio .NET, Borland C++ BuilderX Mobile Edition, and Metrowerks CodeWarrior. For all three, the default operating system is Windows 2000.

You will need to download the Series 80 SDK for Symbian OS or order a copy of the SDK on a CD from Forum Nokia at <http://www.forum.nokia.com/tools>. The SDK contains developer libraries, SDK Help, an emulator, and compilation tools for target devices. If you are using Metrowerks CodeWarrior, be sure to get the Metrowerks Edition of the SDK. The basic edition works for both .NET and BuilderX.

## 4 Background Information and Concepts

This section discusses background topics that are necessary to gain a basic understanding of how to write applications for Series 80 2nd Edition. It discusses Symbian OS and Series 80 concepts that, architecturally speaking, sit close to a Series 80 application.

### 4.1 Uikon

A key part of the application framework is `Uikon`, a standard framework common to all Symbian OS platforms. It not only provides the framework for launching applications, but it also provides a rich array of standard control components (e.g., dialog boxes, number editors, date editors, etc.) that applications can make use of at run time.

### 4.2 Ckon

The `Ckon` layer is an extension of the Symbian OS `Uikon` layer, and is a Series 80-specific GUI layer. `Ckon` determines the application look and feel and the detailed GUI behavior specific to Series 80 2nd Edition.

### 4.3 CONE

Once the application is running, "events" are channeled to it via another part of the Symbian OS framework – the Control Environment (CONE). This component notifies your application of events such as key presses, as well as more advanced events like the machine being turned off, the application coming to the foreground, etc.

### 4.4 Application Framework on Symbian OS

Symbian OS has a concept called the "application framework," which, unless you are already familiar with it, can take a while to understand. Basically, it refers to a certain set of classes that **MUST** be present in order to have a functioning application with a decent-looking UI. Of course, you can opt not to have the framework, and have an application *without* a decent-looking UI; for an example of this, see [Symbian OS: Getting Started with C++ Application Development](#) [1]. However, for a decent Symbian application, you will need to include the framework.

Here's an analogy that will help clarify the framework concept.

Let's say your application is a house. It will have an entry point, the front door, which is represented by the `application` class. When you enter a room, the kitchen, for example, you find utilities with certain functionality — a stove that handles three different heat levels, a sink that produces water, and a refrigerator that keeps food cool. These are comparable to the `application` UI. Then there is the separate factor of what items look like within the kitchen — this is the `view`. You may also need a filing cabinet to store information; this is represented by `documents`.

That's the basic idea. The key base classes that make up the framework are `CEikApplication`, `CEikDocument`, and `CEikAppUi`. All GUI applications are derived from these three base classes plus the `view`. The four parts of the application framework are:

- **Application:** This is the front door of the house, the entry point, and it derives from `CEikApplication`. It is the first object within an application to be instantiated. It has two purposes: it returns the application's unique identifier (like the street address of the house), and it creates the next object in the framework, the `document` object (see below). It passes the

application ID to the `document` object, so that it knows which data belongs to it, i.e., the files that relate to this house can be accepted by the file cabinet.

- **Document:** The `document` class derives from `CEikDocument`, and is always part of the framework, whether or not this application actually deals with "documents" in the sense that the user might think of them. For example, even in the case of the Telephone application, which does not offer the user the ability to create, open, or edit documents, there will still be a `document` class as part of the framework. For applications such as Telephone, the `document` class is little more than an empty class. The purpose of the `document` class is to start any possible engine that your application may have, handle any actual document/setting storage that needs to be done, and create the next object in the framework, the `application UI` object (see below). In the simplest case, all it does is create the `application UI` object.
- **Application UI:** The `application UI` class is derived from the class `CEikAppUi`. This is the class that specifies how to turn on the water in the kitchen sink, how to turn on the stove burners, what happens if there is a fire, etc. It provides major functionality such as event handling and command handling. The `CEikAppUi`-derived class is responsible for creating the final part of the application, the `application view`.
- **View:** The `application UI` has most of the actual functionality "behind the scenes," but the `view` is how the user sees it, for example there may be a chrome faucet with two separate taps for hot and cold, or a gold-plated single tap that swivels from hot to cold. This is the type of thing specified in the `view`. It can be used to simply display data or, in more complex applications, to collect input from the user. The `application UI` and the `view` are closely tied to each other.

#### 4.5 Differences Between Emulator and Target Device

The most important item a beginning developer needs to know is that he must compile the code for each target. Developers must compile one way to get code for the emulator, and a slightly different way to get code for an actual target device.

The differences that exist between an application running under WINS(CW) (Windows or Windows CodeWarrior emulator) and on target hardware such as the Nokia 9500 Communicator are quite subtle. Most application developers will not encounter problems when they move their code from WINS(CW) into a SIS file for installation on the device. However, it is important to note that differences do exist. The key difference lies in processes and threads. The WINS environment is a single process running on a PC. Each application running within this single process runs within its own thread. The target hardware is a multiprocess environment, where each application runs within its own separate process.

This leads to differences in the way applications are launched, which will be explained below in Section 4.7, "Starting an Application."

#### 4.6 Event Handling

Symbian OS and Series 80 2nd Edition applications are required to handle events, such as key presses, generated by the system. The CONE environment provides the event-handling framework to an application. Events can be key presses, menu commands, screen-redraw events, or events from other controls. UI Controls and Application Views need to handle events in a manner consistent with the [Series 80 UI Style Guide](#).

## 4.7 Starting an Application

A visual representation of the entire process from start to finish is as follows:

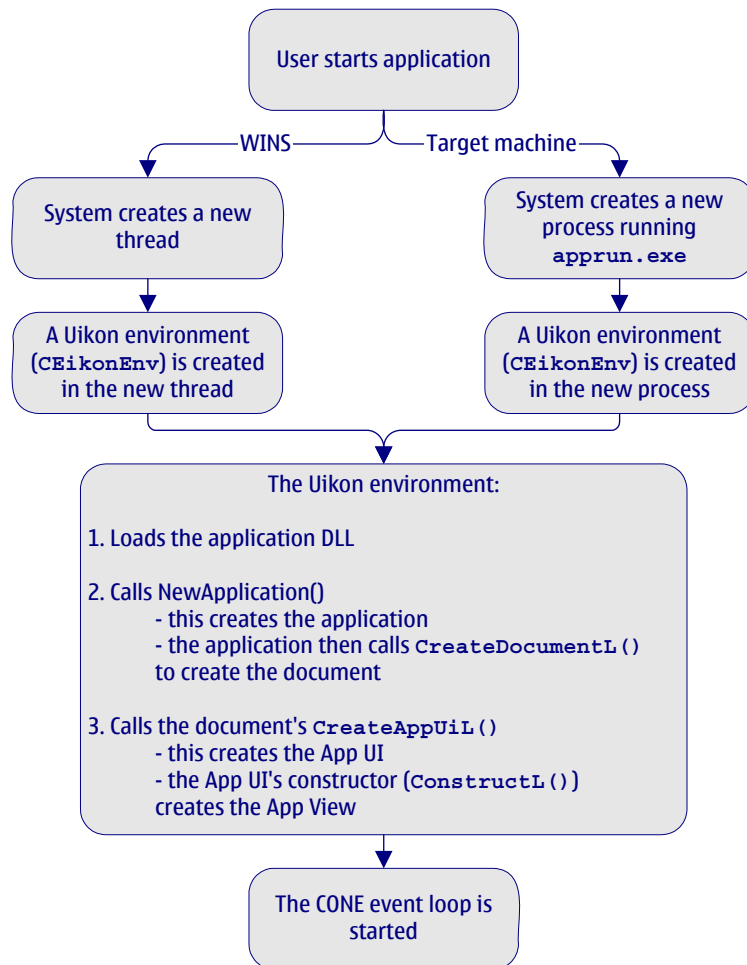


Figure 1: The application starting sequence appears from start to finish.

## 5 Compiling a Sample Application

For a step-by-step guide to compiling your first sample application, please consult the user's guide that accompanied your SDK. The user's guide describes how to compile the HelloWorldBasic example, which can be found at `<sdk installation dir>\series80ex\helloworldbasic`.

## 6 Application File Details

When compiling their first sample application, most developers want to know, “Why on earth do I need more than ten files just to get ‘helloworld?!’” Well, first you need two build files, then a .cpp file and a .h file for each of the four parts of the application framework. In addition, there is generally a .hrh file that lists enumerations, and a .rss, or “resource definition,” file. These files, and any others that are needed, will be explained in more detail in this section.

### 6.1 Build Files

Each Symbian OS project must include the two build files listed below. These are platform/compiler-independent files that can be used by a variety of Symbian OS tools to create tool-chain-specific project files when these are required.

- Project definition file (.mmp file): Specifies which files are needed for this project, where to get the include files, what libraries need to be linked, etc.
- Component description file (bld.inf): Lists .mmp files and may give additional build instructions.

See the *Series 80 Developer Library* (part of the SDK documents) for more information.

#### 6.1.1 Project definition file

A project definition file has the extension .mmp, for “makmake project.”

`makmake` is a tool that can be used to convert project definition files into makefiles for particular platforms. Another tool, called `abld`, wraps calls to `makmake`, and can be more convenient to use than `makmake` directly (more about that in Section 6.1.2, “Component description file”).

Some comments on the .mmp file format:

- Each statement occupies a single line.
- C++ style comment syntax is used for comments.
- A trailing backslash is used to indicate line continuation. Therefore, specify directories without their trailing backslash, for example: `SYSTEMINCLUDE \epoc32\include.`

##### 6.1.1.1 HelloWorldBasic.mmp

For a GUI application, `TARGETTYPE` should be set to *app*.

The `UID` line specifies two numbers. The first, `0x100039CE`, is an identifier used for every GUI application. Each application also requires a unique `UID` for the application itself. In this case, `0X10005B91` is used for the application `UID`. `UID` numbers can be obtained from Symbian. Visit [www.symbian.com](http://www.symbian.com) for details.

The `LIBRARY` statement lists the application framework and graphics libraries required for a GUI application.

```
TARGET           HelloWorldBasic.app
TARGETTYPE       app
UID              0x100039CE 0x10005B91
TARGETPATH       \system\apps\helloworldbasic
LANG             SC

SOURCEPATH       ..\src
```

```

SOURCE      HelloWorldBasic.cpp
SOURCE      HelloWorldBasicApplication.cpp
SOURCE      HelloWorldBasicAppView.cpp
SOURCE      HelloWorldBasicAppUi.cpp
SOURCE      HelloWorldBasicDocument.cpp

SOURCEPATH  ..\group
RESOURCE    HelloWorldBasic.rss

USERINCLUDE ..\inc

SYSTEMINCLUDE \epoc32\include

LIBRARY     euser.lib
LIBRARY     apparc.lib
LIBRARY     cone.lib
LIBRARY     eikcore.lib
LIBRARY     eikcoctl.lib
LIBRARY     bafl.lib
LIBRARY     egul.lib
LIBRARY     ckndlg.lib
LIBRARY     ckncore.lib

```

## 6.1.2 Component description file

The component description file relates to the `abld` tool mentioned earlier. This file, always named `bld.inf`, is used by the `bldmake` tool to create a batch file, `abld.bat`. This batch file is, in effect, the `abld` tool, and running the batch file creates the necessary makefiles.

A *bld.inf* file is made up of a number of sections, with headers `prj_platforms`, `prj_exports`, `prj_testexports`, and `prj_mmpfiles`.

- Each section header can appear any number of times in the file (including none).
- The section headers and their data are case-insensitive.
- C++ style comment syntax is used for comments.
- A trailing backslash is used to indicate line continuation.

The component description file for the HelloWorldBasic application appears below in Section 6.1.2.1, “Bld.inf.”

### 6.1.2.1 Bld.inf

```

PRJ_MMPFILES

HelloWorldBasic.mmp

```

## 6.2 Source Files

Now we’ll go through each of the source files and explain some of what is happening.

### 6.2.1 Common header files

When a user selects a menu item in an application, a command code is sent to the system. What that command code *is* must be defined somewhere, typically in a `.hrh` file. The “resource definition” file (see Section 6.2.2, “Resource definition files”) includes this `.hrh` file, so that it can map the menu item to the command code. The application UI source file includes this `.hrh` file so that it can map the command code to the proper action. Syntactically, a `.hrh` file is a C++ header file.

### 6.2.1.1 HelloWorldBasic.hrh

```
// HelloWorldBasic enumerate command codes
enum THelloWorldBasicIds //Note: T-prefix indicates
                        // "temporary" variable - allocated on stack
{
    EHelloWorldBasicCommand1 = 1 //start value must not be 0
                                //Note: E-prefix indicates
                                //enumeration
};
```

### 6.2.2 Resource definition files

The resource definition file is used to define the application "layout" on the screen. Everything from the status pane, the menu, and the hotkeys, through to individual dialog boxes can be defined in the resource definition file.

See the *Series 80 Developer Library* (part of the SDK documents) for more information.

#### HelloWorldBasic.rss

```
NAME HELL //only 4 letters allowed for this name - go figure

#include <eikon.rh>
#include "HelloWorldBasic.hrh"

// Define the resource file signature
// This resource should be empty.
// -----
RESOURCE RSS_SIGNATURE { }

// Default Document Name
// -----
RESOURCE TBUF r_default_document_name { buf="" ; }

// Define default menu, hotkeys and CBA keys.
// (CBA = Common Button Area, i.e. the 4 extra keys next
// to the screen on a Series 80 device)
// -----
RESOURCE EIK_APP_INFO
{
    menubar = r_helloworldbasic_menubar;
    cba = r_helloworldbasic_cba;
    hotkeys = r_helloworldbasic_hotkeys;
}

// Hotkeys
// -----
RESOURCE HOTKEYS r_helloworldbasic_hotkeys
{
    control =
    {
        HOTKEY
        {
            command = EEikCmdExit; //Note the E-prefix - this
                                //enumeration is defined in Eikon.rh
            key = 'e'; //So Ctrl+E will cause EEikCmdExit command
                    //to be sent to the system...
        }
    };
}

// CBA
// -----
RESOURCE CBA r_helloworldbasic_cba
```

```

    {
        buttons=                                //CBA is array of CBA_BUTTONS
        {
            CBA_BUTTON
            {
                id=EEikCmdExit; //Only defining 1 CBA button, which also
                txt="Exit";     //maps to EEikCmdExit command
            }
        };
    }

//  Menubar
//  -----
RESOURCE MENU_BAR r_helloworldbasic_menubar
{
    titles =                                    //Menubar is array of MENU_TITLES
    {
        MENU_TITLE {menu_pane = r_helloworldbasic_menu; txt = "File";}
    };
}

//  Menu for "File"
//  -----
RESOURCE MENU_PANE r_helloworldbasic_menu
{
    items =                                     //Menupane is array of MENU_ITEMS
    {
        MENU_ITEM {command = EHelloWorldBasicCommand1; txt = "Hello";},
        MENU_ITEM {command = EEikCmdExit; txt = "Exit";}
    };
    //This menu pane has two items - "Hello" sends the command
    //we defined earlier in the .hrh file.
    // "Exit" sends the by now very familiar EEikCmdExit command.
}

```

### 6.2.3 Global functions

Each GUI application, as with any DLL, must define the standard DLL entry point `E32Dll()`. A GUI application exports the `NewApplication()` function, which, when called, causes a new instance of the class derived from the base application class to be constructed. Application errors can be manipulated for easier-to-understand output with a global function — `Panic()`.

See the *Series 80 Developer Library* (part of the SDK documents) for more information.

#### 6.2.3.1 *HelloWorldBasic.cpp*

```

#include "HelloWorldBasicApplication.h"

// DLL entry point, return that everything is ok
GLDEF_C TInt E32Dll(TDllReason /*aReason*/)
{
    return KErrNone;
}

// Create an application, and return a pointer to it
EXPORT_C CAppApplication* NewApplication() //Note: C-prefix indicates
//variable allocated on
//the heap
{
    return (new CHelloWorldBasicApplication);
}

```

### 6.2.3.2 HelloWorldBasic.pan

```
/** HelloWorldBasic application panic codes */
enum THelloWorldBasicPanics
{
    EHelloWorldBasicUi = 1
    // add further panics here
};

inline void Panic(THelloWorldBasicPanics aReason)
{
    _LIT(applicationName, "HelloWorldBasic");
    User::Panic(applicationName, aReason);
}
```

### 6.2.4 Application

Each Uikon application class is derived from `CEikApplication`. Application writers must provide at least an implementation of the form of `CreateDocumentL()` that has no parameters. `AppDllUid()` returns the unique application UID.

#### 6.2.4.1 HelloWorldBasicApplication.h

```
#include <eikapp.h>

class CHelloWorldBasicApplication : public CEikApplication
{
public: // from CEikApplication
    TUid AppDllUid() const;

protected: // from CEikApplication
    CApaDocument* CreateDocumentL(); //Note: L-suffix indicates that
                                     //this function may LEAVE (stop
                                     //functioning due to out-of-memory
                                     //or some other error)
};
```

#### 6.2.4.2 HelloWorldBasicApplication.cpp

```
#include "HelloWorldBasicDocument.h"
#include "HelloWorldBasicApplication.h"

// UID for the application, this should
// correspond to the second uid defined in the mmp file
static const TUid KUidHelloWorldBasicApp = {0x10005B91};
//Note: K-prefix used to indicate a "Konstant"

CApaDocument* CHelloWorldBasicApplication::CreateDocumentL()
{
    // Create a HelloWorldBasic document, and return a pointer to it
    CApaDocument* document = CHelloWorldBasicDocument::NewL(*this);
    return document;
}

TUid CHelloWorldBasicApplication::AppDllUid() const
{
    return KUidHelloWorldBasicApp;
}
```

## 6.2.5 Document

If your application does not deal with files/documents, then this class simply creates the application UI.

Each Uikon application should use a class derived from `CEikDocument` to represent its document. The derived class must implement at least `CreateAppUiL()`.

### 6.2.5.1 *HelloWorldBasicDocument.h*

```
#include <eikdoc.h>

// Forward references
class CHelloWorldBasicAppUi;
class CEikApplication;

class CHelloWorldBasicDocument : public CEikDocument
{
public:
    static CHelloWorldBasicDocument* NewL(CEikApplication& aApp);
    static CHelloWorldBasicDocument* NewLC(CEikApplication& aApp);
    //Note: NewL and NewLC are used in two-phase construction. See
    // "Series 80 Developer Library" (part of the SDK documents) for
    // more information.

    //Note: C-suffix indicates that NewLC leaves variables on
    // the CleanupStack. (L-suffix also indicates that it may LEAVE.)

    ~CHelloWorldBasicDocument();

public: // from CEikDocument
    CEikAppUi* CreateAppUiL();

private:
    void ConstructL();
    CHelloWorldBasicDocument(CEikApplication& aApp);
};
```

### 6.2.5.2 *HelloWorldBasicDocument.cpp*

```
#include "HelloWorldBasicAppUi.h"
#include "HelloWorldBasicDocument.h"

// Standard Symbian OS construction sequence
CHelloWorldBasicDocument*
CHelloWorldBasicDocument::NewL(CEikApplication& aApp)
{
    CHelloWorldBasicDocument* self = NewLC(aApp);
    CleanupStack::Pop(self);
    return self;
}

CHelloWorldBasicDocument*
CHelloWorldBasicDocument::NewLC(CEikApplication& aApp)
{
    CHelloWorldBasicDocument* self =
        new (ELeave) CHelloWorldBasicDocument(aApp);
    CleanupStack::PushL(self);
    self->ConstructL();
    return self;
}
```

```

void CHelloWorldBasicDocument::ConstructL()
{
    // no implementation required
}

CHelloWorldBasicDocument::CHelloWorldBasicDocument
    (CEikApplication& aApp) : CEikDocument(aApp)
{
    // no implementation required
}

CHelloWorldBasicDocument::~CHelloWorldBasicDocument()
{
    // no implementation required
}

CEikAppUi* CHelloWorldBasicDocument::CreateAppUiL()
{
    // Create the application user interface,
    // and return a pointer to it,
    // the framework takes ownership of this object
    CEikAppUi* appUi = new (ELeave) CHelloWorldBasicAppUi;
    return appUi;
}

```

## 6.2.6 Application UI

An application UI is responsible for opening and closing the application document's files in response to user commands. It also owns and manages the core controls of the user interface, for example the tool bar, menu bar, and hotkeys, and handles commands from these. The commands that an application UI handles are specified in the resource definition file as explained earlier in Section 6.2.1, "Common header files."

Every Uikon application should use its own class derived from `CEikAppUi` to handle application-wide user interface details. Responses to various kinds of events can be handled at the level of the application UI by providing suitable implementations of the following virtual functions:

- `CCoeAppUi::HandleKeyEventL()`: **Key events**
- `CCoeAppUi::HandleForegroundEventL()`: **Application switched to foreground/background**
- `CCoeAppUi::HandleSwitchOnEventL()`: **Machine switched on**
- `CCoeAppUi::HandleSystemEventL()`: **System events**
- `CCoeAppUi::HandleMessageReadyL()`: **Message ready**
- `CCoeAppUi::HandleApplicationSpecificEventL()`: **Application-specific events**
- `CEikAppUi::HandleCommandL()`: **Handles commands defined in resource definition file**

### 6.2.6.1 *HelloWorldBasicAppUi.h*

```

#include <eikappui.h>

// Forward reference
class CHelloWorldBasicAppView;

class CHelloWorldBasicAppUi : public CEikAppUi
{
public:
    void ConstructL();
    CHelloWorldBasicAppUi();
}

```

```

    ~CHelloWorldBasicAppUi();

public: // from CEikAppUi
    void HandleCommandL(TInt aCommand);

private:
    CHelloWorldBasicAppView* iAppView;
};

```

### 6.2.6.2 HelloWorldBasicAppUi.cpp

```

#include <eikenv.h>
#include <ckninfo.h>

#include "HelloWorldBasic.pan"
#include "HelloWorldBasicAppUi.h"
#include "HelloWorldBasicAppView.h"
#include "HelloWorldBasic.hrh"

// ConstructL is called by the application framework
void CHelloWorldBasicAppUi::ConstructL()
{
    BaseConstructL();
    iAppView = CHelloWorldBasicAppView::NewL(ClientRect());
    AddToStackL(iAppView); //Adds view to the "Control Stack", so we
                           //can "listen" to events/commands from it
}

CHelloWorldBasicAppUi::CHelloWorldBasicAppUi()
{
    // no implementation required
}

CHelloWorldBasicAppUi::~CHelloWorldBasicAppUi()
{
    if (iAppView)
    {
        iEikonEnv->RemoveFromStack(iAppView);
        delete iAppView;
        iAppView = NULL;
    }
}

// handle any menu commands - here is where we finally deal with the
// commands we have defined earlier.
void CHelloWorldBasicAppUi::HandleCommandL(TInt aCommand)
{
    switch(aCommand)
    {
        case EEikCmdExit:
            CBaActiveScheduler::Exit();
            break;

        case EHelloWorldBasicCommand1:
            {
                _LIT(title, "Information");
                _LIT(message, "Hello World Basic");
                CKnInfoDialog::RunDlgLD(title, message);
                //Note: D-suffix indicates function will Delete the
                //dialog it creates after the function completes.
            }
            break;

        default:
            Panic(EHelloWorldBasicUi);
            break;
    }
}

```

```
    }  
}
```

## 6.2.7 View

A view is what the user actually sees on the screen. The base class for a view is typically `CEikBorderedControl`. The actual drawing functionality is in a function called `Draw`.

### 6.2.7.1 *HelloWorldBasicAppView.h*

```
#include <eikbctrl.h>  
  
class CHelloWorldBasicAppView : public CEikBorderedControl  
{  
public:  
    static CHelloWorldBasicAppView* NewL(const TRect& aRect);  
    static CHelloWorldBasicAppView* NewLC(const TRect& aRect);  
    ~CHelloWorldBasicAppView();  
  
public: // from CEikBorderedControl  
    void Draw(const TRect& aRect) const;  
  
private:  
    void ConstructL(const TRect& aRect);  
    CHelloWorldBasicAppView();  
};
```

### 6.2.7.2 *HelloWorldBasicAppView.cpp*

```
#include <cknenv.h>  
#include <HelloWorldBasic.rsg>  
#include "HelloWorldBasicAppView.h"  
  
// Standard construction sequence  
CHelloWorldBasicAppView* CHelloWorldBasicAppView::NewL(const TRect&  
aRect)  
{  
    CHelloWorldBasicAppView* self =  
        CHelloWorldBasicAppView::NewLC(aRect);  
    CleanupStack::Pop(self);  
    return self;  
}  
  
CHelloWorldBasicAppView* CHelloWorldBasicAppView::NewLC(const TRect&  
aRect)  
{  
    CHelloWorldBasicAppView* self =  
        new (ELeave) CHelloWorldBasicAppView;  
    CleanupStack::PushL(self);  
    self->ConstructL(aRect);  
    return self;  
}  
  
CHelloWorldBasicAppView::CHelloWorldBasicAppView()  
{  
    // no implementation required  
}  
  
CHelloWorldBasicAppView::~CHelloWorldBasicAppView()  
{  
    // no implementation required  
}  
  
void CHelloWorldBasicAppView::ConstructL(const TRect& aRect)
```

```

    {
    // Create a window for this application view
    CreateWindowL();

    // Set the window's size
    SetRect(aRect);

    // Activate the window, which makes it ready to be drawn
    ActivateL();
    }

// Draw this application's view to the screen
void CHelloWorldBasicAppView::Draw(const TRect& aRect) const
{
    // Draw the parent control
    CEikBorderedControl::Draw(aRect);

    // Get the standard graphics context
    CWindowGc& gc = SystemGc();
}

```

### 6.3 Installation Files

In order to run the application in the device, the application must be built for the target device and an installation file must be created.

- Go to the project's group directory and create the `abld.bat` batch file:

```
bldmake bldfiles
```

- Go to the project's group directory and build for the THUMB target platform:

```
abld build thumb urel
```

- Go to the project's sis directory and create the installation file:

```
makesis HelloWorldBasic.pkg
```

- Transfer the produced installation (.sis) file to the device using, for example, an infrared port.

#### 6.3.1 Package file

A Package (.pkg) file is a text file that contains installation information for applications or files. It consists of the following parts, some of which may be omitted:

- The languages supported
- The package header, including the name of the component to be installed, its build, and version information
- Product/platform version compatibility
- Package-signature details (optional)
- Package lines (optional) formed from the following:
  - Options line
  - Condition blocks
  - Language-independent files to install
  - Language-dependent files, of which only one will be installed
  - Capabilities line

- Requisite components
- Embedded SIS files
- Comments

The first item in the package file should detail the languages provided within the sis file. This is followed by the package header, which details the name of the component to be installed in all supported languages, the UID of the component, and version information. An optional digital signature line may be specified, which gives the name of the private key file and associated certificates to be used to digitally sign the sis file.

It is important that the product/platform version compatibility lines are done correctly. This is the way in which you specify what products/platforms your application is designed to work on. The installation program looks for a certain file on the target device, based on the Product ID and text string. If no files are found that match the IDs given, the installer presumes that the application is not meant to work on this device/platform, and may not allow installation of the application. The Series 80 Platform, the Nokia 9300 and the Nokia 9500 Communicator IDs are listed below; use these lines exactly as they appear here.

The remainder of the file lists the files and components to be included in the sis file or dependencies upon other installed components.

### 6.3.1.1 *HelloWorldBasic.pkg*

```
; HelloWorldBasic.pkg
;
;Language - standard language definitions
&EN

; standard SIS file header
; (here version is 1.0-0)
#{ "HelloWorldBasic" }, (0x10005B91), 1, 0, 0

;Supports Series 80 2nd Edition
(0x101F8ED2), 0, 0, 0, {"Series80ProductID"}
;Supports Nokia 9300
(0x101F8ED1), 0, 0, 0, {"Series80ProductID"}
;Supports Nokia 9500
(0x101F8DDB), 0, 0, 0, {"Series80ProductID"}

;Files in "Source"- "Destination" format
;i.e. first line says to take HelloWorldBasic.APP from the
;...thumb\urel directory, and copy it to c:\system\apps\HelloWorldBasic
;directory on the device.
"..\\..\\..\\epoc32\\release\\thumb\\urel\\HelloWorldBasic.APP"
-"C:\system\apps\HelloWorldBasic\HelloWorldBasic.app"
"..\\..\\..\\epoc32\data\z\system\apps\HelloWorldBasic\HelloWorldBasic.rsc"
-"C:\system\apps\HelloWorldBasic\HelloWorldBasic.rsc"
```

## 7 Where to Go Next

For further information:

- Read documents provided in the SDK.
- Browse the Symbian Web site at <http://www.symbian.com/>.
- Browse the Symbian section at the Forum Nokia Web site at <http://www.forum.nokia.com/symbian>.
- Read [\*Series 80 2nd Edition: Designing C++ Applications\*](#) at the Forum Nokia Web site at <http://www.forum.nokia.com/symbian> under **Documents**.

## 8 Terms and Abbreviations

Term or abbreviation	Meaning
ARM	ARM processor.
ARMI	ARM instruction set. The ARMI instruction set allows interworking between the ARM4 and THUMB instruction sets by providing an extra instruction that will allow swapping between sets.
ARM4	ARM 32-bit instruction set.
Ckon	GUI library for the Series 80 platform.
CONE	Control Environment.
GUI	Graphical user interface.
IDE	Integrated development environment. A system for supporting the process of writing software, including a syntax-directed editor, graphical tools for program entry, and integrated support for compiling and running the program and relating compilation errors back to the source, e.g., CodeWarrior.
SDK	Software development kit. Set of programming tools for creating applications and enhancing the use of certain software.
SIS	A file type for Symbian installation files.
THUMB	ARM 16-bit instruction set.
UID	Unique Identifier. A globally unique 32-bit number used in a compound identifier to uniquely identify an object, file type, etc.
Uikon	UI and control framework common to Symbian platforms.
WINS	The target platform for the Microsoft Windows-hosted emulator
WINSCW	The target platform for the Microsoft Windows-hosted emulator, Metrowerks CodeWarrior Edition

## 9 References

[1] *Symbian OS: Getting Started with C++ Application Development*,  
<http://www.forum.nokia.com/symbian> | Getting Started

## 10 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](#).