Symbian Mobile Programming

G.Rossetti & A.Schneider
HTI Biel
2005 - 2007
Acknowledgments

This course is based on our collective experiences over the last years, we have worked on Symbian mobile programming. We are indebted to all the people, that made our work fun and helped us reaching the insights that fill this course. We would also like to thank our employers for providing support and accommodation to teach this lecture.

These are Swisscom Innovations and SwissQual AG.
## Contents

1 Course Overview
   1.1 Lecturers ............................................. 1
   1.2 Examples ............................................. 1
   1.3 Motivation ........................................... 1
   1.4 Course contents ..................................... 2

2 Introduction to Symbian
   2.1 History ............................................. 3
      2.1.1 EPOC OS Releases 1-4 ........................... 3
      2.1.2 Symbian 5 ....................................... 3
      2.1.3 Symbian 6 ....................................... 3
      2.1.4 Symbian 7 ....................................... 3
      2.1.5 Symbian 8 ....................................... 4
      2.1.6 Symbian 9 ....................................... 5
   2.2 Symbian OS Architecture ............................. 6
   2.3 Instruction Sets ..................................... 6
   2.4 IDEs ................................................ 7
   2.5 SDKs ................................................ 7
   2.6 Useful links ........................................ 8
      2.6.1 Symbian OS manufacturer ...................... 8
      2.6.2 Symbian OS licensees ......................... 8
      2.6.3 3th party links ................................ 8
   2.7 Lecture Focus ....................................... 8

3 Framework ............................................. 9
   3.1 Classes ............................................. 9
   3.2 Launch sequence .................................... 9
   3.3 Basic Example ..................................... 10
      3.3.1 Project File ................................. 10
      3.3.2 Source- and Header-Files .................. 11
      3.3.3 Building Project ............................. 16
      3.3.4 Creating Installation File ................ 17

4 Symbian Types ....................................... 19
   4.1 Class Types ....................................... 19
      4.1.1 T Classes .................................. 19
      4.1.2 C Classes .................................. 19
      4.1.3 R Classes .................................. 20
      4.1.4 M Classes .................................. 20
   4.2 Numeric Types ..................................... 20
   4.3 Descriptors (Strings) ............................. 21
      4.3.1 Buffer descriptors ......................... 21
      4.3.2 Heap descriptor ........................... 22
      4.3.3 UTF16 $\longleftrightarrow$ UTF8 ............... 22
      4.3.4 Integer $\longleftrightarrow$ String ............ 22
   4.4 Arrays .............................................. 22
      4.4.1 Fixed Size Arrays ......................... 23
      4.4.2 Dynamic Arrays ........................... 23
5 Events & Menus .................................................. 26
  5.1 Menu based Events ........................................ 26
    5.1.1 Creating Menus ...................................... 26
    5.1.2 Handle Menu-Action Event .......................... 27
    5.1.3 Handle Menu-GUI Event ............................... 28
  5.2 Key Events .................................................. 28

6 Views & Drawing .............................................. 30
  6.1 Using a View .............................................. 30
  6.2 Drawing ..................................................... 30

7 GUI Components .............................................. 32
  7.1 Notes ....................................................... 32
  7.2 Dialogs ...................................................... 32

8 Multitasking .................................................. 34
  8.1 Threads, Multitasking and Preemption ................... 34
  8.2 Active Objects ............................................ 34

9 GSM API ....................................................... 38
  9.1 Overview .................................................. 38
  9.2 ETel Implementation ...................................... 38
    9.2.1 RPhone ............................................... 38
    9.2.2 RLine ................................................ 39
    9.2.3 RCall ................................................. 39
  9.3 Using ETel client API ..................................... 39
    9.3.1 Initialising basic objects ........................... 40
    9.3.2 Setting up a call ..................................... 40
    9.3.3 Listening for incoming call .......................... 41
    9.3.4 Answering incoming call ............................. 41
    9.3.5 Observing call state changes ........................ 41
    9.3.6 Hanging up a call .................................... 42

10 SMS & MMS API .............................................. 43
  10.1 Using Messaging Framework ............................... 43
  10.2 Example .................................................. 45

11 Bluetooth ..................................................... 46
  11.1 Overview ................................................ 46
  11.2 Symbian OS Bluetooth architecture ..................... 46
  11.3 Bluetooth Application development .................... 47
    11.3.1 Bluetooth Protocols ................................ 47
    11.3.2 Bluetooth Profiles ................................... 47
    11.3.3 Service Advertisement and Service Discovery .... 48
  11.4 Basic Example ............................................ 48

A Symbian Installation Guide .................................. 1
  A.1 Overview ................................................ 1
  A.2 Device Overview ......................................... 1
    A.2.1 Nokia Series 60 ..................................... 1
    A.2.2 Nokia Series 80 ..................................... 3
    A.2.3 Other Nokia Symbian Phones with special SDK’s ... 3
    A.2.4 SonyEricsson UIQ .................................... 3
  A.3 IDE adaptations ........................................... 4
    A.3.1 Codewarrior ......................................... 4
    A.3.2 Carbide .............................................. 4
  A.4 Common Errors ............................................ 4
A.4.1 Codewarrior ................................................. 4
A.4.2 Carbide ..................................................... 5

List of Figures

1 Symbian OS Architecture ........................................ 6
2 Framework UML .................................................. 9
3 ViewApp Example ................................................ 31
4 DrawingApp Example ............................................. 31
5 NotesApp Example ............................................... 32
6 QueryApp Example ............................................... 33
7 GSM Stack ......................................................... 38
8 ETel related classes ............................................. 39
9 Call state machine ............................................... 39
10 Bluetooth Architecture ......................................... 46
11 Bluetooth state machine ....................................... 1
1 Course Overview

1.1 Lecturers

- Gian Rossetti, Solothurn  
  gian.rossetti@bfh.ch, +41794671138  
  SW Engineer, Swisscom Innovations

- Adrian Schneider, Thun  
  adrian.schneider@bfh.ch  
  Symbian SW Engineer, formerly working at SwissQual AG

1.2 Examples

Example projects are available in a protected web directory:

```
url: http://www.eidelen.ch/SymbianMobile/load/
user: symbianer
passw: mobilehti
```

1.3 Motivation

Our motivation is to provide the necessary impulse, which awakes your own interests to become a Symbian expert. We hope, programming Symbian will inspire you, as it inspired us at that time.

Starting programming Symbian devices won’t always be easy. It’s inexcusably C++ code, the framework looks strange from the first point of view and more development steps are needed, compared to conventional programming on PC. Often there is not a lot of information available, the IDEs are bad looking and gray, your friends never heard from it and your mother always explains it absolutely wrong... in other words we talk about real embedded engineering. We try to make the beginning as easy as possible. It is very important to understand the basics as well as the additional development steps; This knowledge can be extended later by reading about particular APIs in the SDK-Help.

Mobile application development is a fast growing business and Symbian is the most used and most powerful operating system running on mobile devices. It will be an advantage having knowledge and experiences programming Symbian. Today, there is a lack of programmers having experience with embedded operating systems!

Further on you will use C++ language in a practical way. During the course you will get familiar with the framework and understand some useful, powerful constructs. Perhaps C++ will become the preferred language for some of you.
### 1.4 Course contents

<table>
<thead>
<tr>
<th>Lessons</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>Introduction to Symbian (History, SDKs, IDEs); Demo of some applications</td>
</tr>
<tr>
<td>3-4</td>
<td>Framework (Constructor, Heap); First application</td>
</tr>
<tr>
<td>5-6</td>
<td>Types (Strings, Numbers) and Events; practical training</td>
</tr>
<tr>
<td>7-8</td>
<td>Views, Drawing and Dialogs; practical training</td>
</tr>
<tr>
<td>9-10</td>
<td>Multitasking and Active Objects</td>
</tr>
<tr>
<td>11-12</td>
<td>SMS- and MMS-API; practical training</td>
</tr>
<tr>
<td>13-14</td>
<td>GSM-API (Voice-Calls, Layer3-Data); practical training</td>
</tr>
<tr>
<td>15-16</td>
<td>Bluetooth-API; practical training</td>
</tr>
<tr>
<td>17-18</td>
<td>GPRS-API (TCP/IP-Connection); practical training</td>
</tr>
<tr>
<td>19-20</td>
<td>An Introduction to Symbian 9.x</td>
</tr>
<tr>
<td>21-22</td>
<td>to be defined</td>
</tr>
<tr>
<td>23-24</td>
<td>Project</td>
</tr>
<tr>
<td>25-26</td>
<td>Project</td>
</tr>
<tr>
<td>27-28</td>
<td>Project</td>
</tr>
<tr>
<td>29-30</td>
<td>Short project presentations</td>
</tr>
</tbody>
</table>
2 Introduction to Symbian

2.1 History

2.1.1 EPOC OS Releases 1-4

1980, Psion Software was founded by David Potter. He was physics lecturer, no Wizard.

In between 1991 and 1998 Psion released several devices based on the EPOC16 architecture (also known as SIBO / EPOC Releases 1-3). 1997 EPOC Release 4 (ER4) was introduced based on a real 32-bit architecture (EPOC32). In 1998, Symbian Ltd. had been formed as a partnership between Ericsson, Nokia, Motorola and Psion, to explore the convergence between PDAs and mobile phones.

2.1.2 Symbian 5

EPOC Release 5 was released in 1999. A rich suite of application engines, including contacts, schedule, browsing, voice, office, utility and system control has been presented. It’s first devices were the Ericsson MC218 and Ericsson R380. These were not ‘open’ phones, software could not be installed. Later the netBook and the netPad were introduced, using ER5U (U = Unicode), which had been presented in 2000. Developers can make use of Symbian OS 5 SDKs and SDK extensions to target devices in C++ and OPL.

2.1.3 Symbian 6

The first ‘open’ Symbian OS phone, the famous Nokia 9210, was released on Symbian 6.0. A C++ SDK including development tools and an emulator based on the Crystal GTK was available for free download. Using MS Visual Studio 6.0 as IDE, development has been facilitated for Nokia 9210 communicator series. Symbian 6 was conform to mobile telephony protocols, such as 2G voice and circuit-switched data, 2.5G packet-switched data and SMS. Communication protocols such as TCP, IPv4, GPRS, WAP, Bluetooth, IrDA, and serial support were covered as well as a large set of features, like email (POP3, IMAP4, SMTP and MHTML), Fax, encryption, certificate management, secure communications protocols (HTTPS, WTLS and SSL), certificate-based application installation, OBEX, a multimedia server with support for several audio and image formats and the extensions for PersonalJava 3.0 have been introduced.

The Nokia 7650, 3650 imaging phones and the forthcoming N-Gage mobile game deck are based on Symbian 6.1 and introduced the Series60 generation. Symbian devices are now sold on the mass market. Borland and Metrowerks announce plans to deliver C++ tools for Series 60. Metrowerks are extending the capability of their existing CodeWarrior Development Tools for Symbian OS.

2.1.4 Symbian 7

The new Symbian OS 7 enabled devices, developed for 2.5G as well as 3G networks worldwide, opened a wide range of simple, attractive, revenue-generating applications and services. The Nokia 6600 is the most famous Series60 device, produced for Symbian 7.0. It was announced long before shipped effectively. Symbian 7.0 covers a comprehensive integrated messaging environment using OPL is an interpreted language targeted at Symbian OS phones. OPL is an entry-level development tool that enables rapid development of applications. OPL consists of phone-side code (runtime and translator) and PC-side code (translator and tools).

Author: Gian Rossetti
MMS, EMS and SMS, support for a full web browser, shared access to screen, keyboard, fonts and bitmaps, audio recording and playback, an API for graphics acceleration and streaming functionality, extended communication protocols such as IPv6 (with IPSec), USB support, over the air (OTA) data synchronization using SyncML and PIM data synchronization, the WIM framework, Bluetooth v1.1 and additional support for Java (J2ME MIDP 1.0 and PersonalJava 3.0 with JavaPhone 1.0 options). Symbian OS v7.0s is a fit-for-purpose platform for the 3G market, offering new functionality to enable 3GPP compliance and the delivery of 3G services. The major new features of Symbian OS v7.0s are: Lightweight multi-threaded multimedia framework, support for W-CDMA and Java MIDP 2.0. Java developers developing PersonalJava or J2ME applications ( MIDlets) can use Sun’s SDK, and the Wireless Toolkit (WTK), both of which are available for free download.

Nevertheless with the release of Symbian OS v7.0 in 2003, the OS was split into two big subcategories:

- **Series60**
- **UIQ**

**UIQ** Sony Ericsson P800, P900, P910, and P990 smartphones are based on this framework. Originally UIQ was an abbreviation of ‘User Interface Quartz’. The UI is designed and optimized for touchscreen displays. As developing for UIQ is just supported using the CodeWarrior IDE, (no SDK’s available for MS Visual Studio 6.0) C++ developers will need to obtain CodeWarrior Development Studio for Symbian OS.

**S60** The Series60 UI just supports conventional display and user interaction. Other to the UIQ SDK, the Series60 SDK contains per default additional multimedia APIs, which in turn is of great interest among developers. Both IDEs, MS Visual Studio 6.0 and Metrowerks CodeWarrior IDE can be used to target Series60 devices.

When developing Symbian applications, it’s best to separate the UI and application logic. This limits the amount of code that needs to be ported between platforms. In fact there is a third subcategory, the Nokia communicator series, which used from it’s beginning a separate UI framework (Crystal GTK). As this generation of devices did never target the mass market, it has been buried in oblivion for a while. At a later date it will become the Series80.

In 2004, Psion sold its stake in Symbian.

### 2.1.5 Symbian 8

Symbian OS v8.0 is ready for the 3G market with support for WCDMA (3GPP R4), GSM circuit switched voice and data (CSD, EDGE ECSD) and packet-based data (GPRS and EDGE EGPRS), CDMA (IS-95 and cdma2000), simply a UMTS enabled 3G phone. It provided also the SIM, R-UIM and UICC Toolkit. Enhanced features like OBEX for exchanging appointments (vCalendar) and business cards (vCard), the latest wireless Java standards MIDP 2.0, CLDC 1.1, additional JSR2 namely Mobile Media API (JSR 135), Bluetooth (JSR 082), Wireless Messaging (JSR 120), the SIM Application Toolkit class 3, QoS on GPRS and UMTS networks, SyncML DM 1.1.2 compliance, and secure protocols such as TLS have been added. Symbian OS v8.0, first shipped in 2004, offered the choice between two different kernels (EKA1 / EKA2). The

---

2 Java Specification Request: An J2ME runtime can additionally to its specification (f.ex. MIDP2.0) include further JSR implementations, which can be considered as API extensions

Author: Gian Rossetti
kernels behave very similar from user-side, but are internally quite different. EKA1 maintains compatibility with old device drivers, whereas EKA2 offered advantages such as single-chip phone support and a hard real-time capability. Unfortunately EKA2 kernel version did not ship until Symbian OS v8.1b. Symbian OS v8.1. was basically a cleaned-up version of 8.0, available as 8.1a and 8.1b (EKA1 / EKA2). The 8.1b version was popular among Japanese phone companies (DoCoMo OCD) desiring the realtime support but not allowing open application installation. Symbian 8.1 covered support for CDMA circuit switched voice, data and packet-based data (IS-95 and 1xRTT), Bluetooth v1.2, as well as additional JSRs such as the Mobile 3D Graphics API (JSR 184) and Personal Information Management and File GCF APIs (JSR 075). It provided support for USB client specification v2.0 (full speed), IPSec and VPN client support, as well as further Bluetooth profiles. In 2004, the first worm for Symbian phones was developed. 'Cabir' uses Bluetooth to spread itself to phones in its neighborhood.

2.1.6 Symbian 9

Symbian OS v9.0 was just used for internal Symbian purposes and discontinued in 2004. In 2005, Symbian OS v9.1 was announced. The new ARM EABI binary model conditions developers to rebuild their applications and possibly to re-code some parts of the application. A completely new security model has been introduced. The Nokia N91 was probably the first Symbian OS 9.1 device on the market. Symbian OS has generally maintained reasonable binary compatibility. The OS was compatible from ER1 to ER5, then from 6.0 to 8.1b. Substantial changes, related to the security model, were needed for 9.0, but this should be a unique event. The Symbian OS is now better protected than ever. It might even be an overkill in some way, as development for Symbian devices has been complicated. The new security model forces developers to submit their application to a TestHouse, where the application will be tested against a set of security and capability tests. If all tests are successfully approved, the application will be signed digitally by 'SymbianSigned'. The new Symbian OS contains on one hand an OS architecture that allows applications to have a protected data store, which is well separated among each other and on the other hand a proactive defense mechanism, based on granting and monitoring application capabilities through SymbianSigned certification. Symbian OS v9.1 provides additional support for WCDMA (3GPP R4 and R5 IMS), CDMA network roaming, third party OTA API, NAM programming mode, CDMA SMS stack, NAI handset identification, interfaces to enable Mobile IP, bridge and router gateway modes of operation, support for PPP and Mobile IP CDMA specifications, Connectionless WSP and WAP Push, OMA DM 1.1.2 compliance, OMA Client provisioning v1.1, email with support for POP3, IMAP4, MIME attachments, SMTP, SMTP authentication, communication protocols such MSCHAP v2 and RTP, HTTP 1.1, Pipelining, Multihoming, Cryptographic algorithms (DES, 3DES, RC2, ARC4, RC5 and AES), X509 Certificate management and the Bluetooth stereo headset support. Symbian OS V 9.2 has just been released, it contains support for RTCP, SIP, an additional JSRs namely JTWI (JSR 185), Content Handling API (JSR 211), Smart Card access APIs, OMA Device Management v1.2 and OMA Client provisioning v1.1 compliance and Bluetooth v2.0
2.2 Symbian OS Architecture

The Symbian OS is a sophisticated 32-bit operating with a modular, object-oriented C++ architecture. As the OS is especially designed for mobile devices, it consumes few resources. In contrast to MS PocketPC devices, a Symbian device can run for many months without being switched off. Reliability and stability were key design goals for the OS. The Symbian C++ API provides complete access to services, such as messaging and multimedia, as well as extended device and OS functionality. The Symbian OS API contains hundreds of C++ classes grouped in subsystems. Application engines give access to data from built-in applications (contacts, calendar objects...), which allows third-party developers to integrate their services with core applications. The GUI framework provides controls for the UI handling and graphics. There are subsystems for mobile telephony services, messaging, multimedia services, browsing utilities and sounds. Other subsystems provide support for networking interfaces such as Bluetooth, IrDA, USB and communication protocols like TCP/IP, HTTP, and WAP. On the very bottom there is the base system, which is responsible for file and memory access, date, time, and other basic system capabilities.

2.3 Instruction Sets

Currently there are three build variants for ARM-based devices. Either you compile your project using the THUMB-instruction set, the ARMI-instruction set or the ARM4-instruction set. Symbian OS devices run many applications and servers by default. These applications may be itself compiled for Thumb, ARM or ARM4. Some instruction sets allow 'Interworking' between different binary interfaces. The possible ABIs (Application Binary Interface) for executables are:

- ARM4
- ARMI
- THUMB

ARM4 is a 32-bit instruction set for targeting StrongArm processors supporting ARM4 builds. ARM4 does not include the THUMB extensions. If an applica-
tion is compiled for ARM4 it can only call function, that are itself compiled for ARM4 or ARMI. ARM4 builds run faster than THUMB builds.

**THUMB** is a 16-bit instruction set provided by older models of the ARM processor. If an application is compiled using the THUMB instruction set, it can only call functions, that are itself compiled for THUMB or ARMI. THUMB builds are more efficient on systems that have 16-bit wide ROM, because only one cycle is needed to fetch an instruction. Furthermore THUMB builds are smaller and therefore save ROM (typically about 20 percent.) Nowadays mostly THUMB based applications run slower.

**ARMI** is a 32-bit instruction set with interworking for ARM based processors. An application compiled using the ARMI instruction set can call any other function. It is entirely compiled as ARM yet still able to support installed applications compiled as Thumb. In our course ARMI is the recommended default build format for compatibility purposes.

### 2.4 IDEs

There are three IDEs for Symbian programming. First of all, the well known *MS Visual Studio C++ 6.0*, secondly *Metrowerks CodeWarrior for Symbian OS* and recently released *Nokia Carbide*. Visual Studio C++ does on one hand not cover all needed functionalities and building tools (just the very basic ones) and on the other hand it’s strictly restricted to Nokia S60 devices. Therefore VC++ will not be further discussed.

For a long time Codewarrior was the only IDE, which could handle all Symbian SDKs. It seems that Codewarrior is not being further enhanced, while Carbide is a new generation of mobile development tools from Nokia. Carbide allows developing software for multiple platforms and multiple languages. Based on the open Eclipse framework, Carbide can be extended with other eclipse plug-ins and products. As this course has been written, before Carbide was released, you will find some hints for Codewarrior developers inside this script. The goal for long terms is to alter this script tailored for Carbide developers.

### 2.5 SDKs

Nokia SDKs support development using all IDEs, Visual Studio, CodeWarrior and Carbide. The SDKs provide binaries and tools to facilitate development, building and deployment for Symbian OS devices, including a PC-based emulator, and lots of example applications, as well as a documentation of the APIs and tools. Especially the SDK-documentation will be very helpful during this course.

SonyEricsson SDKs do not support developing with MS Visual Studio at all, just CodeWarrior and Carbide IDE. Therefore UIQ development can just take place using the CodeWarrior or Carbide IDE. Of course you could build your project using command line tools\(^3\), but this is rather uncomfortable.

Motorola provides their own SDK, it is very similar to the one of SonyEricsson. Motorola manufactured Symbian devices use the same UI API as SonyEricsson, as they have a touchscreen display too. Motorola SDKs are supported by Code-warrior, but not yet using Carbide.

\(^3\)more information about command line tools will be provided later in the course
2.6 Useful links

2.6.1 Symbian OS manufacturer

- Symbian
  www.symbian.com - 'Developer' - Newsgroups - official site of the Symbian OS Manufacturer

2.6.2 Symbian OS licensees

- Forum Nokia
  www.forum.nokia.com - 'Technical Services' - Discussion Boards, Forums

- Sony Ericsson
  www.sonyericsson.com - 'Developer World' - Discussion Boards and Forum

- Motorola
  www.motocoder.com - 'Knowledge and Support'

2.6.3 3th party links

- NewLC
  www.newlc.com

- My-Symbian
  www.my-symbian.com

- AllAboutSymbian
  www.allaboutsymbian.com

- Aikon
  www.aikon.ch

2.7 Lecture Focus

Customers definitively benefit enhanced security features of Symbian 9.x Platform architecture. Nevertheless developing and especially prototyping for the secured environment became more complex. As applications, that perform interactions with the OS on a low level (such as GSM, GPRS, Messaging and File Access), need to be signed in order to be installed on a physical device, it can’t be covered within this course. Signing an application does on one hand involve handing in the application to an approved TestCenter (which takes at least one week) and on the other hand it causes costs per SigningRequest. Later in the course, the varieties (concerning code conventions and macros) between Symbian 6.0 - 8.x and Symbian 9 Platform will be addressed.
3 Framework

Similar to frameworks such as MFC (C++) and the old style of Windows programming (C), Symbian applications have to implement a basic framework, which provides the OS a necessary handle to the application. Generally, the project wizard will generate these files for you. Alternatively you can copy it from an existing project.

3.1 Classes

There are several levels of inheritance within the basic framework. In figure 2 you will see a detailed class diagram. Important is the difference between the two main Symbian platforms Series60 an UIQ. Of course, the third platform called Cristal (Nokia Communicator Series) will have a specific top level inheritance too.

![Figure 2: Framework UML](image)

3.2 Launch sequence

A simple overview of an application start:

1. **Application entry point** instantiates CEikApplication
2. **CEikApplication** creates CEikDocument and holds application specific UID
3. **CEikDocument** generates CEikAppUi and represents the data model

4. **CEikAppUi** handles the commands generated from menu options and creates (optionally) one or more views (CCoeControl)

5. **View** contains graphical controls

### 3.3 Basic Example

In that subsection, we will analyse source, project and installation files of a basic example. The main project was generated with *Series 60 Application Wizard*; there are only some modifications.

[C Complete source is in example **BApp**]

#### 3.3.1 Project File

The project file (*.mmp) specifies the properties of a project in a platform and compiler independent way. This file will be imported into the IDE, from which you can starting development.

```plaintext
// file generated by the project
TARGET BApp.mmp

// kind of project
TARGETTYPE app

// UIDs for a project
UID 0x100039CE 0x0C1F3B86

// where the project should be released
TARGETPATH \system\apps\BApp

// location of the workfiles
SOURCEPATH ..\src
SOURCE BAppApp.cpp
SOURCE BAppAppui.cpp
SOURCE BAppDocument.cpp
SOURCE BAppContainer.cpp

SOURCEPATH ..\data
RESOURCE BApp.rss
RESOURCE BApp_caption.rss

// languages code for the project
LANG SC

// project’s header files directories
USERINCLUDE .
USERINCLUDE ..\inc

// system header files directories
SYSTEMINCLUDE . \epoc32\include

// import libraries
LIBRARY euser.lib apparc.lib cone.lib eikcore.lib
LIBRARY eikcoctl.lib avkon.lib

// application information file; references to icons
```

Author: Adrian Schneider
If project is generated and built by command-line, then you need an additional file called `bld.inf`, which refers to the mmp-file.

**MMP to CodeWarrior-Project:** Metrowerks CodeWarrior offers GUI supported way to import mmp-files. It’s possible to select sdk version and build targets. *File - Import Project from *.mmp File*

**MMP to MS VC++ Project** VC++ don’t have a built-in tool to handle Symbian project files. You have to generate VC++ project files (*.dsw) by your own in command-line.

1. To select sdk version, set default sdk with command `devices -setdefault`
2. Go to mmp-file directory (group)
3. Execute cmd-command `bldmake bldfiles`, that will generate `Abld.bat`
4. Execute cmd-command `makmake BApp vc6` (BApp is to replaced with current project name)

If you receive errors like *command not found*, mostly you have to adapt your Path-Environment.

**3.3.2 Source- and Header-Files**

Following, you see the implementation of a basic set of objects (discussed in 3.2) necessarily for a *minimum application*. These files are produced by *Series 60 Application Wizard* too and are compilable without any changes.

**Application entry point:** See source comment for more information!

```cpp
#ifndef BAPPAPP_H
#define BAPPAPP_H
#include <aknapp.h>

// App UID; application specific
const TUid KUidBApp = { 0x0C1F3B86 };

class CBAppApp : public CAknApplication
{
private:
    CAppDocument* CreateDocumentL();
    TUid AppDllUid() const;
};
#endif
```

```cpp
#include "BAppApp.h"
#include "BAppDocument.h"
```

Author: Adrian Schneider
// Returns Apps UID; Will be called from Symbian OS
TUid CBAppApp::AppDllUid() const
{
    return KUidBApp;
}

// Creates Document
CApaDocument* CBAppApp::CreateDocumentL()
{
    return CBAppDocument::NewL( *this );
}

// Will be called from Symbian OS
EXPORT_C CApaApplication* NewApplication()
{
    return new CBAppApp;
}

// DLL entry point
GLDEF_C TInt E32Dll( TDllReason )
{
    return KErrNone;
}

CAknDocument See source comment for more information!

--- BAppDocument.h -------------------------------
ifndef BAPPDOCUMENT_H
#define BAPPDOCUMENT_H

#include <akndoc.h>

class CEikAppUi;

class CBAppDocument : public CAknDocument
{
    public:
        static CBAppDocument* NewL(CEikApplication& aApp);
        virtual ~CBAppDocument();

    private:
        CBAppDocument(CEikApplication& aApp);
        void ConstructL();

    private:
        CEikAppUi* CreateAppUiL();
};
#endif

--- BAppDocument.cpp -------------------------------
#include "BAppDocument.h"
#include "BAppAppui.h"

CBAppDocument::CBAppDocument(CEikApplication& aApp)
: CknDocument(aApp)
{
}

CBAppDocument::~CBAppDocument()
void CBAppDocument::ConstructL()
{
}

CBAppDocument* CBAppDocument::NewL(CEikApplication& aApp)
{
    CBAppDocument* self = new (ELeave) CBAppDocument(aApp);
    CleanupStack::PushL(self);
    self->ConstructL();
    CleanupStack::Pop();

    return self;
}

CEikAppUi* CBAppDocument::CreateAppUiL()
{
    return new (ELeave) CBAppAppUi;
}

CEikAppUi See source comment for more information!

--- BAppAppui.h ---
#ifndef BAPPAPPUI_H
#define BAPPAPPUI_H

#include <aknappui.h>

class CBAppContainer;

class CBAppAppUi : public CAknAppUi
{
public:
    void ConstructL();
    void CBAppAppUi();

private:
    void HandleCommandL(TInt aCommand);

private:
    CBAppContainer* iAppContainer;
};

@endif

--- BAppAppui.cpp ---
#include "BAppAppui.h"
#include "BAppContainer.h"
#include <BApp.rsg>
#include "BApp.hrh"
#include <avkon.hrh>

// for CAknInformationNote
#include <aknnotewrappers.h>

void CBAppAppUi::ConstructL()
{
// completes the UI framework construction
BaseConstructL();

// Creating application view
iAppContainer = new (ELeave) CBAppContainer;
iAppContainer->SetMopParent( this );

// Passing Screen-Area
iAppContainer->ConstructL( ClientRect() );

// Adding view to ViewStack
AddToStackL( iAppContainer );
}

CBAppAppUi::~CBAppAppUi()
{
if (iAppContainer)
{
    RemoveFromStack( iAppContainer );
    delete iAppContainer;
}
}

// takes care of command handling
void CBAppAppUi::HandleCommandL(TInt aCommand)
{
switch ( aCommand )
{
    case EAnSoftkeyBack:
    case EEikCmdExit:
    {
        Exit();
        break;
    }
    case EBAppCmdAppTest:
    {
        // popup a note
        CAknInformationNote* informationNote =
            new (ELeave)CAknInformationNote();
        informationNote->ExecuteLD(_L("HTI Symbian Course"));
        break;
    }
    default:
    break;
}
}

CCoeControl / View See source comment for more information!

#ifndef BAPPCONTAINER_H
#define BAPPCONTAINER_H
#include <coecntrl.h>
class CEikLabel;
class CBAppContainer : public CCoeControl

Author: Adrian Schneider
public:
    void ConstructL(const TRect& aRect);
    ~CBAppContainer();

private:
    // From CoeControl, SizeChanged.
    void SizeChanged();
    // From CoeControl, CountComponentControls.
    TInt CountComponentControls() const;
    // From CCoeControl, ComponentControl.
    CCoeControl* ComponentControl(TInt aIndex) const;
    // From CCoeControl, Draw.
    void Draw(const TRect& aRect) const;

    private:
        CEikLabel* iLabel; // example label
    
#endif

--- BAppContainer.cpp -----------------------------------------------------

#include "BAppContainer.h"
#include <eiklabel.h>

void CBAppContainer::ConstructL(const TRect& aRect)
{
    // creates the views window,
    CreateWindowL();

    // Init Label
    iLabel = new (ELeave) CEikLabel;
    iLabel->SetContainerWindowL( *this );
    iLabel->SetTextL( _L("Example View") );

    // defines the area of the view
    SetRect(aRect);

    // indicates to the framework that the view and
    // its controls are now ready for drawing
    ActivateL();
}

CBAppContainer::~CBAppContainer()
{
    delete iLabel;
}

// Called by framework when the view size is changed
void CBAppContainer::SizeChanged()
{
    iLabel->SetExtent( TPoint(20,30), iLabel->MinimumSize() );
}

// Called by framework to get number of components
TInt CBAppContainer::CountComponentControls() const
{
    return 1;
}

// Returns a pointer to the control at index aIndex

Author: Adrian Schneider 15
CCoeControl* CBAppContainer::ComponentControl(TInt aIndex) const
{
    switch ( aIndex )
    {
    case 0:
        return iLabel;
    default:
        return NULL;
    }
}

// Draw Area - Called by each redraw event
void CBAppContainer::Draw(const TRect& aRect) const
{
    CWindowGc& gc = SystemGc();
    gc.SetPenStyle( CGraphicsContext::ENullPen );
    gc.SetBrushColor( KRgbWhite );
    gc.SetBrushStyle( CGraphicsContext::ESolidBrush );
    gc.DrawRect( aRect );
}

3.3.3 Building Project

Compared to PC-Development (Java/Eclipse, C++/VC++, Delphi,...), building for embedded systems always contains some more steps. Particularly the Symbians build process is not a comfortable way to go. For the first time you can follow the instructions below, but I promise, after two or three examples you will be able to type these lines by heart.

Tools for building Symbian executables are part of the sdk installation and free for use (gcc). Depending on the IDE, there are more or less things to do. But understanding command-line tool is inevitably.

Following we look at three (there exists no more) different initial positions and learn how to get executables running on Win32 Symbian Emulator (EPOC) and target devices.

Without IDE

1. To select sdk version, set default sdk with command devices -setdefault
2. Go to mmp-file directory (group)
3. Execute cmd-command bldmake bldfiles, which will generate Abld.bat, used for further building.
4. Type abld build wins for VC++ sdk or abld build winscw for CodeWarrior sdk. Now, emulator binaries will be builded and stored in emulator directory epoc32/release/wins(cw)/. If not, trace build-output for errors.
5. Cmd-command abld build armi will make the target executables. They will be placed in epoc32/release/armi/. Of course you can use thumb or arm4 instead of armi.

If you have to build a Symbian application consisting of many projects (a frontend app and some selfwritten DLLs), pure cmd-building is the only existing solution to do that in one step. Write your own .bat file, that executes these commands for each project.

Author: Adrian Schneider
VC++  As you know, VC++ isn’t a development tool for Symbian. So you shouldn’t expect good programmers support like code completion or closer to our current topic, building. Fortunately, the variable and modular project settings of Microsofts VC++ opens the door to develop, build and debug for emulator. That means, you have to build only for target platform by your self.

1. devices -setdefault ...
2. bldmake bldfiles
3. abld build armi

CodeWarrior  Building for both, emulator and target platform, will be supported within Metrowerks CodeWarrior. Because Nokia, Symbian and SonyEricsson use CodeWarrior, new SDKs are firstly available for this IDE.

Carbide  Carbide behaves very similar to Codewarrior, building for all targets is supported. There is only one issue, that needs specific building procedere. The first released SDK (based on Symbian 6.x) is not officially supported by Carbide. Therefore building for Symbian 6.x devices must be done using command line instructions (as described above). This will be a good exercise for getting used to command line instructions. Nevertheless Carbide can then be used to run the Symbian 6.x application in the Emulator. As the Carbide IDE is very new, the example applications in this lecture are still based on Codewarrior. This implies, that you first have to import the project based on the mmp-file, and then you call the Eclipse-built-in functionality 'Update Symbian MMP File' by clicking the project with the right mouse button and choosing it in the context-menu. Afterwards you should delete the intermediate build files in the Carbide workspace (but not the sources). Now you can again import the mmp-file and everything should be fine. A big remark when using Carbide concerns the following point. Other to Codewarrior, Carbide will not allow to import and compile a project if there is already a project using the same namespace (target and source project name) in the workspace. Therefore you will need to delete all project specific intermediate and binary files, when re-importing the same project again.

3.3.4 Creating Installation File

To create a Symbian installation file, sometimes called sis-file, you have to adapt a pkg-file, which is later used by the Symbian-Installation-File-Make-Tool. A pkg-file links local machine builded files to the file-locations on the target device. Be aware that there is also a device specific Product-ID in the pkg-file (as shown below), which must be adapted according the target device and underliecing SDK. Check the Nokia Series60 Examples in the appropriate SDK, you will find the device specific Product-ID in the included examples.

```plaintext
;Languages
&EN
;
; UID is the app's UID
;
#{"BApp"},(0x0C1F3B86),1,0,0
;
;Supports Series 60 v 1.2
;(0x101F6F88), 0, 0, 0, {"Series60ProductID"}
```

Author: Adrian Schneider
The Symbian-Installation-File-Make-Tool can be called with cmd-command `makesis`.

1. `makesis BApp.pkg`

Now, the resulting file can be installed on the target device.

Author: Adrian Schneider
4 Symbian Types

4.1 Class Types

On Symbian OS there are four general class types. It’s a kind of class naming convention for grouping object types by their characteristics. Programmers are not obligated to follow that guideline. But it is helpfully to understand the Symbian API, because Symbian Ltd. programmers should made use of it.

4.1.1 T Classes

The most fundamental classes are simple value types, which class names beginning with letter T.

T characteristics:

- T types contain their value. They do not own any external object.
- T types may be allocated either on the stack or as class-members.
- Many T types don’t need a constructor. Those that do, use the constructor to initialise own data.
- T types have no destructor.

T examples:

- Integer, without constructor
  
  TInt myInt = 2;

- String, using constructor
  
  TBuf<3> myBuf(_L("abc"));

4.1.2 C Classes

Most none T classes in sdk are C classes and if you are going to implement your own thing, mostly it belongs to characteristics of a typical C class. Like T class naming, C classes beginning with letter C.

C characteristics:

- C classes are derived from CBase.
- C classes are allocated on the heap.
- C classes are passed by pointer or reference.
- C classes uses a constructor and a destructor.

C examples:

- Create a label
  
  CEikLabel* myLabel = new (ELeave) CEikLabel();

- Call function from label
  
  myLabel->SetTextL( _L("Bienne") );

- Call destructor
  
  delete myLabel;

Author: Adrian Schneider
4.1.3 R Classes

R classes are proxies for external objects, which mostly owned by the OS itself and belongs to a limited pool of ressources. Of course, classes starts with letter R.

R characteristics:

- R classes don’t make use of a constructor. Instead, they have to open or connect.
- R classes don’t have implemented a destructor. They will be released by close.

R examples:

- Open access to file system
  RFs fileserver;
  fileserver.Connect();
- Close access
  fileserver.Close();

4.1.4 M Classes

M classes define interfaces. Mostly they will be used for callbacks.

4.2 Numeric Types

Handling with Symbian numeric types is absolutely equal to standard C or C++. At compile time, the compiler will translate names of Symbian numeric to well known, standard types.

That’s part of the sdk-include e32def.h:

```c
typedef signed char TInt8;
typedef unsigned char TUint8;
typedef short int TInt16;
typedef unsigned short int TUint16;
typedef long int TInt32;
typedef unsigned long int TUint32;
typedef signed int TInt;
typedef unsigned int TUint;
typedef float TReal32;
typedef double TReal64;
typedef double TReal;
typedef int TBool;
```

In addition, Symbian offered a 64 bit integer called TInt64, which consists of two TInt16.
Booleans (TBool) will be EFalse and ETrue.

An example:

```c
TInt limit = 10;
TUint counter = 0;
TReal accum = 0.0;
TBool go = ETrue;
```

```c
while(go)
{
```
accum += ((TReal)(counter)) / 2;
counter++;
go = counter < limit;
}

4.3 Descriptors (Strings)
String handling in Symbian is a bit pedantically. There are several kind of objects, which doing approximately the same. Each of them exists in four configurations:

- UTF8, non-modifiable
- UTF8, modifiable
- UTF16, non-modifiable
- UTF16, modifiable

Further there are two equivalent macros to define descriptors content itself, but they only exists in two configurations:

- UTF8 _L8("my Text")
- UTF8 _LIT8(KStrName,"my Text")
- UTF16 _L16("my Text")
- UTF16 _LIT16(KStrName,"my Text")

In following sections we try understand some of the most used descriptor types.

4.3.1 Buffer descriptors
Buffer descriptors, TBufC and TBuf, contain their data as part of themselves, like char[] arrays in C. Because stack size on Symbian systems is very small, buffers limited to a length of 255.

_LIT(KAtoG,"abcdefg");
TBuf<20> str(KAtoG); //TBuf -> TBuf16

TChar ch = str[0]; // ch contains the character 'a'
// ch = str[13]; // Panic!!!
str[0] = 'z'; // changes str to "zbcdefg"

str.Length(); // returns 7
str.MaxLength(); // returns 20

TInt pos = -1;
pos = str.Find(_L("cd")); // returns 2
pos = str.Find(_L("xy")); // returns KErrNotFound

TBufC<20> str2 = _L("hijk");
str.Append(str2); // str contains "zbcdefghijk"
//many other string routines...

There’s a dirty manipulation to force buffers content to heap. That means, buffer can reach system limited length.

TDes* myBigBuffer = new TBuf<13000>;
...
delete myBigBuffer;

Note: It’s better to use heap descriptors instead of that work-around!
4.3.2 Heap descriptor

Heap descriptors, HBufC, contain their data in heap.

```
_LIT(KText,"Symbian");
HBufC* myHeapDesc = HBufC::NewL(13000);
myHeapDesc->Des() = KText;
TBuf<20> strCopy = *myHeapDesc; //strCopy contains "Symbian"

TPtr ptr2data = myHeapDesc->Des();
ptr2data.Append(_L(" course")); //myHeapDesc contains "Symbian course"
```

4.3.3 UTF16 ←→ UTF8

Converting between UTF formats is a frequently used functionality. Mostly, low level APIs return descriptors in UTF8 but GUI components use UTF16 as input.

```
#include <utf.h> //from library charconv.lib
...

//Convert UTF8 -> UTF16
TBuf<20> out16;
TBuf8<20> in8 = _L8("convert up");
CnvUtfConverter::ConvertToUnicodeFromUtf8(out16,in8);

//Convert UTF16 -> UTF8
TBuf<20> in16 = _L("convert down");
TBuf8<20> out8;
CnvUtfConverter::ConvertFromUnicodeToUtf8(out8,in16);
```

4.3.4 Integer ←→ String

```//Convert Int -> String
TInt intIn = 3;
TBuf<5> bufOut;
bufOut.Num(intIn); // bufOut contains "3"
bufOut.AppendNum(intIn); // bufOut contains "33"

//Convert String -> Int
TInt intOut;
TLex bufLex(bufOut);
bufLex.Val(intOut); // intOut = 33```

4.4 Arrays

Symbian offers a number of easy to use object containers. In advantage to standard arrays, Symbian’s predefined classes built in nice functionalities and error handling (comparable with JAVA Vector and basic Array).

```
TInt myBasicArray[23]; // C-Array Stack
TInt* myBasicArrayHeap = (TInt*)calloc(23,sizeof(TInt)); //C Heap
TFixedArray<TInt,23> mySymbianArray; // Symbian-Array

//examine length of one item
TInt arrLength = mySymbianArray.Length();

//max containing elements - not possible for C-Array
```
4 SYMBIAN TYPES

TInt nbrOfObjects = mySymbianArray.Count();

//Indexing out-of-range
myBasicArray[23];
mySymbianArray[23]; //USER 133 Panic – can be handled

4.4.1 Fixed Size Arrays

Fixed Size Arrays provide an inexpensive but safety programming alternative to standard C++ arrays.

/*** init Array ***/
const TInt data[] = {TInt(1), TInt(2), TInt(3), TInt(4)};

// initialise at construction time
TFixedArray<TInt, 4> intArray(&data[0], 4);

// or later using Copy()
intArray.Copy(&data[0], 4);

/*** access elements ***/

// copy element
TInt myCopy = intArray.At(2); //myCopy = 3

// edit element
intArray.At(2) = TInt(99); //intArray = [1, 2, 99, 4]

/*** delete elements ***/

intArray.Reset();

4.4.2 Dynamic Arrays

A large number of concrete classes provide dynamic arrays that are suitable for different circumstances. Unlike standard C++ arrays or the Symbian OS Fixed Size Arrays, the number of elements in a dynamic array can be altered at run-time.

The choice of a suitable container, depends on the characteristics of the array element object:

Element properties

- **Fixed length**: All elements are of the same size.
- **Variable length**: Elements can be of different sizes.
- **Pointer**: Elements are pointers to CBase-derived objects.
- **Untyped**: Elements are TAny objects.

Storage type

- **Flat buffer**: The elements are stored in a single memory area, and are of fixed length.
- **Segmented**: The elements are stored in multiple memory areas.

---

*In the purchase to the run time.*
- **Packed**: The elements are stored in a single memory area, but can be of variable length.

<table>
<thead>
<tr>
<th></th>
<th>Flat</th>
<th>Segmented</th>
<th>Packed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed length</td>
<td>CArrayFixFlat</td>
<td>CArrayFixSeg</td>
<td>-</td>
</tr>
<tr>
<td>Variable length</td>
<td>CArrayVarFlat</td>
<td>CArrayVarSeg</td>
<td>CArrayPakFlat</td>
</tr>
<tr>
<td>CBase pointer</td>
<td>CArrayPtrFlat</td>
<td>CArrayPtrSeg</td>
<td>-</td>
</tr>
<tr>
<td>Untyped (fix)</td>
<td>CArrayFixFlat&lt;TAny&gt;</td>
<td>CArrayFixSeg&lt;TAny&gt;</td>
<td>-</td>
</tr>
<tr>
<td>Untyped (var)</td>
<td>CArrayVar&lt;TAny&gt;</td>
<td></td>
<td>CArrayVar&lt;TAny&gt;</td>
</tr>
</tbody>
</table>

Table 1: Dynamic Array Decision Table

Following code defines a simple object. Later, we will use that object as array element.

```cpp
// definition
class CMyObject: public CBase
{
    public:
        CMyObject(TInt aInt, TDesC& aText); //Constructor
        ~CMyObject();  //Destructor

        TInt iInt;  // Class Memory (Stack)
        HBufC* iText;  // Heap Descriptor
};

// implementation
CMyObject::CMyObject(TInt aInt, TDesC& aText)
{
    iInt = aInt;

    //allocating 10KB memory
    iText = HBufC::NewL(10000);
    *iText = aText;
}
CMyObject::~CMyObject()
{
    //free allocated memory
    delete iText;
}

See how to handle CMyObject with dynamic array.

//*** init dynamic array ***//
TInt nbrOfElementsDyn = 2*5;
CArrayPtrFlat<CMyObject>* dynamicArr =
    new (ELeave) CArrayPtrFlat<CMyObject>(nbrOfElementsDyn);

//*** add elements ***//
//init object for later adding to array
CMyObject* objPtr1, *objPtr2;
TBuf<20> objBuf(_L("first"));
objPtr1 = new CMyObject(13, objBuf);
objBuf = _L("second");
```
objPtr2 = new CMyObject(26, objBuf);

// add generated object to array
dynamicArr->AppendL(objPtr1);
dynamicArr->AppendL(objPtr2);

//*** access and edit elements ***//
dynamicArr->At(0)->iInt = 43;

TBuf<45> newText(_L("abcdefghi"));
*(dynamicArr->At(1)->iText) = newText;

//*** delete elements ***//
// delete first element
// destructor will be called automatic
dynamicArr->Delete(0, 1);

// deletes all
dynamicArr->Reset();

delete objPtr1;
delete objPtr2;
delete dynamicArr;
5 Events & Menus

The Symbian framework provides event driven programming. That sounds difficult but it is not. It’s exactly the same as programming JAVA-GUI applications or MFC or Delphi.

In that chapter, we have a look to some important system events and where they arrives in the framework. In fact we are going to process

- Menu-Action-Events
- Menu-GUI-Events
- Key-Events

5.1 Menu based Events

Menus on Symbian is a replacement for the buttons on typical windows applications. It’s the natural way for users to interact with the application... or more generally, to control their hole mobile phone.

5.1.1 Creating Menus

Creating a menu and link the events to an action is not as easy as it is in Delphi or Java, where programmers have a powerful IDE doing a lot of invisible work. Within the Symbian framework, we have to work with a resource file, which contains GUI specific definitions such as dialogs, titles and of course menus.

A resource file have the file prefix *.rss and it based on Symbians Resource Language Definition. The resource source code will be compiled to a binary *.rsc file, which must be installed later on the device too.

Below you will see the rss-file from our example MenuEventApp:

```plaintext
// RESOURCE IDENTIFIER
NAME   AWIZ // 4 letter ID

#include <eikon.rh>
#include "BApp.hrh"
#include <avkon.rsg>
#include <avkon.rh>
#include <avkon.mbg>

RESOURCE RSS_SIGNATURE { }

RESOURCE TBUF { buf = "BApp"; }

RESOURCE EIK_APP_INFO
{
    menubar = r_bapp_menubar;
    cba = R_AVKON_SOFTKEYS_OPTIONS_BACK;
}

RESOURCE MENU_BAR r_bapp_menubar
{
    titles =
    {
        MENU_TITLE { menu_pane = r_bapp_menu; txt = "File"; }
    };
}
```

Author: Adrian Schneider
5 EVENTS & MENUS

RESOURCE MENU_PANE r_bapp_menu
{
    items =
    {
        MENU_ITEM { command = EBAppCmdAppTest; txt = "Test Btn"; },
        MENU_ITEM { command = EAknCmdExit; txt = "Exit"; },

        //menu entry with eventcode 11211
        MENU_ITEM { command = 11211; txt = "Event 1"; },

        //open submenu
        MENU_ITEM { cascade = r_sub_menu; txt = "Menu Event 2"; }
    };
}

RESOURCE MENU_PANE r_sub_menu
{
    items =
    {
        //menu entry with eventcode 22122
        MENU_ITEM { command = 22122; txt = "Event 2"; }
    };
}

It's not recommended to define event-codes like command = 22122. We did that only for illustration reasons. Usually you should define events in an enumeration!!

5.1.2 Handle Menu-Action Event

To handle a selection of one of our predefined actions, we have to capture concerning event-codes in AppUi's inherited function HandleCommandL.

```cpp
void CBAppAppUi::HandleCommandL(TInt aCommand)
{
    switch ( aCommand )
    {
        case EAknSoftkeyBack:
        case EEikCmdExit:
        {
            Exit();
            break;
        }
        case EBAppCmdAppTest:
        {
            MyPopUp(_L("HTI Symbian Course"));
            break;
        }

        //added events

        case 11211: //event 1
        {
            TBuf<20> outBuf(_L("Event 1\n"));
            outBuf.AppendNum(aCommand);

            MyPopUp(outBuf);
            break;
        }
    }
```
5 EVENTS & MENUS

```cpp
case 22122: //event 2
{
    TBuf<20> outBuf(_L("Event 2\n"));
    outBuf.AppendNum(aCommand);
    MyPopUp(outBuf);
    break;
}

default:
    break;
}
```

```cpp
void CBAppAppUi::MyPopUp(const TDesC& aStr)
{
    // popup a note containing aStr
    CAknInformationNote* informationNote = new (ELeave)CAknInformationNote;
    informationNote->ExecuteLD(aStr);
}
```

5.1.3 Handle Menu-GUI Event

Menu-GUI events are generated before the menu will be drawn. The function `DynInitMenuPaneL` have to be implemented in AppUi.

```cpp
void CBAppAppUi::DynInitMenuPaneL(TInt aResourceId, CEikMenuPane* aMenuPane)
{
    switch(aResourceId)
    {
    case R_BAPP_MENU:
    {
        TBool doDim = ((iMenuDimer++)%3) == 0;
        // dime/not dime the menu-entry EBAppCmdAppTest
        aMenuPane->SetItemDimmed(EBAppCmdAppTest,doDim);
        break;
    }
    }
}
```

5.2 Key Events

Key events are passed to the top GUI-Component of focused application. If the key event wouldn’t be processed by any of those objects (function `CoeControl::OfferKeyEventL`), at least the event will be passed in AppUi's function `HandleKeyEventL`.

```cpp
TKeyResponse CBAppAppUi::HandleKeyEventL(const TKeyEvent& aKeyEvent, TEventCode aType)
{
    switch(aType)
    {
    case EEventKey:
    {
        if( aKeyEvent.iScanCode == '1' )
        {
            MyPopUp(_L("Key 1"));
            return EKeyWasConsumed;
        }
    }
```
if( aKeyEvent.iScanCode == '3' )
{
    MyPopUp(_L("Key 3"));
    return EKeyWasConsumed;
}

break;
}

return EKeyWasNotConsumed;
6 Views & Drawing

A *view* or sometimes called *container* is nothing more than a rectangular area of the screen. If you have in mind our first very basic example (see 3.3.2), then you know that a view belongs to the group *controls*. Controls had together, that they all will be derived from base class *CCoeControl*. So your views header file will start like

```
#include <coecntrl.h>

class CMyView : public CCoeControl
{ ...
```

If a control contain several other controls (textfield, label, button,...), then we speak about *compound controls*. *Simple controls* do not contain other object of that type.

Classes which inherit from *CCoeControl* are received the obligation to implement few functions used by framework:

- `void SizeChanged()` Responds to size changes to sets the size and position of the contents of this control.
- `TInt CountComponentControls()` Gets the number of controls contained in a compound control.
- `CCoeControl* ComponentControl(TInt aIndex)` Gets the specified component of a compound control.
- `void Draw(const TRect& aRect)` Draw a control; called by window server.

### 6.1 Using a View

Example *ViewApp* show you how to make a view and how interactions between AppUi and view works.

The view contains 2 control objects

- `CEikLabel* iL1` to display some text
- `CEikEdwin* iIn` to get text input

The command *Set Label* will copy text-input to labels text. That action is controlled from AppUi.

### 6.2 Drawing

Code for drawing can be placed in the function body of `void Draw(const TRect& aRect)`. A complete overview of all drawing methods you will find in sdk-help under *CWindowGc*!

In example *DrawingApp* you will see some frequently used drawing functions:

- `void DrawRect(const TRect& aRect)` to draw a rect
- `void DrawEllipse(const TRect& aRect)` to draw a ellipse bordering

Author: Adrian Schneider
void DrawBitmap(const TRect& aRect, const CFbsBitmap* aBmp) to draw a bitmap object in aRect. There are several ways to get aBmp. In that example, we convert a bitmap file at project generation (look at mmp-file) to a multi-bitmap file called *.mbm.

Figure 3: ViewApp Example

Figure 4: DrawingApp Example
7 GUI Components

In that section we will learn to provide user interaction by using Symbian graphical user interface components. There exists a lot of GUI-Elements within Symbian and usually releasing new OS version brings up some more of theme. So we going to process only basics... in generally components existing since Symbian 6.1.

UIQ-Programmers has some more, prettier possibilities to do user interaction (grater/touchable screen). On demand we will gladly inform about it.

7.1 Notes

A Note is used as feedback component that informs the user about application states, occurred errors and much more.

There are 6 types of notes:

- Confirmation Note to show a successfully operation
- Information Note to inform about unexpected situation
- Warning Note to notify about risky actions
- Error Note by crashing actions
- Permanent Note remain on screen for indefinite time
- Wait Note for bridging a time consuming function by drawing a progress bar.

Picture 5 contains screen shots from a further course example application called NotesApp.

![NotesApp Example](image)

7.2 Dialogs

A Dialog is used to query user input. Different to Notes, a Dialog needs a resource entry in the *.rss file. First it appears to be difficult. The resource
syntax is an independently script language. It is advisable to use existing re-
source examples and adapt them to your own thing.

The following course example contains the most frequently used dialog types.

Figure 6: QueryApp Example

```cpp
//cpp
TInt passedRet = 50;
CAknNumberQueryDialog* dlgStctPT =
    CAknNumberQueryDialog::NewL(passedRet,CAknQueryDialog::ENoTone);
dlgStctPT->ExecuteLD(R_INTEGER_INPUT)
//passedRet holds the integer
```
8 Multitasking

8.1 Threads, Multitasking and Preemption

Symbian has its application area in the world of portable, battery-powered, wireless devices. Compared with desktop systems, resources are rare. First priority is efficiency, and not to claim to the most powerful processor or "unlimited" memory. Battery power considerations are always present. As the systems must be responsive to its users, and to the real-time needs of communications-intensive protocols, the number of threads running concurrently must be kept minimal. Although Symbian is implemented as a preemptive multi-tasking OS, the usage of real Threads (class RThread) is discouraged and not covered in the scope of this course, because of its inefficiency. Using many Threads (and an unlucky set priority) could slow down the OS and should only be considered for strong reasons. The paradigm "to make efficient use of the available resources" is still valid in the world of mobile devices. Each application, and each system server, is given its own thread (mostly also its own process). Thanks to Active Objects, it is extremely rare that applications require more than one thread, which contributes significantly to the OS's reliability. Active Objects are efficient and, thanks to object orientation, easy to work with.

8.2 Active Objects

Active objects are Symbians non-preemptive, cooperative multi-tasking infrastructure and an alternative to threads. Objects belonging to classes that extend CActive are commonly called Active Objects. Usually Active Objects request asynchronous services. Requesting such services can only be done by passing a reference to itself to the requested service provider. Once the request has sent, control returns to the Active Object immediately. In the meantime other tasks can be completed, the Active Object might even go to sleep. When the asynchronous task has completed, the operating system will identify the thread containing the requesting Active Object, and wake up that thread. The "Active Scheduler" will identify the object that made the request, and pass control back to that Active Object.

The implementation of Active Objects in Symbian OS is based around each thread having a "request semaphore". This is incremented when a thread is due to complete an asynchronous request, and decremented when the request has been completed. When there are no outstanding requests, the thread is put to sleep. Therefore one ActiveObject can request multiple asynchronous services without the need of implementing each requested service in its own thread. An Active Object treats multiple asynchronous requests as a sequence of requests, well defined in its own machine.

Let's take the class RTimer, a simple class containing asynchronous functions, and see how Active Objects are built: First of all our multitasking enabled class must inherit from CActive. See the Header declaration of our ActiveObject:

```c++
class MyTimer : public CActive
{
    ...  
    RTimer iTimer;
}
```

When implementing an Active Object, at least the two functions listed below must be present (by inheritance):

- `RunL();`
DoCancel();

Almost all code in an EPOC system executes under the control of active-object RunL() member function. This is the function that will be called by the operating system once the asynchronous request has completed. The iHost object is a reference to the class that constructs and uses the MyTimer object. All it does, it notifies that the request has completed and returns a time event.

```c
//From CActive
void CMyTimer::RunL()
{
  iHost.MyTimerEventNotify(iTimeEvent);
}
```

DoCancel() is the 2nd function that must be implemented when inheriting from CActive. This function can be called when an outstanding asynchronous request must be canceled. Here DoCancel() just calls the function Cancel() of class RTimer.

```c
//From CActive
void CMyTimer::DoCancel()
{
  iTimer.Cancel();
}
```

As you have seen now, how the Active Object can be canceled, you probably may ask how it can be started. The function below starts the execution of the asynchronous task. It mustn’t be named 'After' (by inheritance), it just makes sense in our context. Firstly the asynchronous function 'After' of class RTimer is initialised, and a reference to iStatus is passed to the RTimer object. Immediately afterwards the object is set active by calling CActive(). This function-call triggers the execution of the asynchronous function. An Active Object might also be triggered in the constructor. Once the asynchronous request has finished, the RunL() function, described above, will be called.

```c
void CMyTimer::After(TTimeIntervalMicroSeconds32 aDelay, TInt aTimerEvent)
{
  Cancel();
  this->iTimeEvent = aTimerEvent;
  this->iDelay = aDelay;
  iTimer.After(iStatus, this->iDelay); //iStatus is inherited from CActive
 .SetActive(); //and contains a Request status information
}
```

In the next listing you see the complete Header file declaration of the class CMyTimer. The public function NewL(), which is used to construct the CMyTimer object, is supposed to receive a MMyTimerHostNotifier object. The reason for this construct will be explained in the next section. Furthermore you see in the private data section the declaration of the member object iTimer of class RTimer.

```c
#ifndef __MYTIMER_H
#define __MYTIMER_H

Author: Gian Rossetti

35
```
```cpp
#include <e32std.h>
#include <w32std.h>

class CMyTimer : public CActive
{
    public:
        static CMyTimer* NewL( MMyTimerHostNotifier& aHost);
        "CMyTimer();
        void After(TTimeIntervalMicroSeconds32 aDelay, TInt aTimerEvent);

    protected:
        static CMyTimer* NewLC( MMyTimerHostNotifier& aHost );
        CMyTimer( MMyTimerHostNotifier& aHost );
        void ConstructL();

    private: // from CActive
        void RunL();
        void DoCancel();

    private: //data
        MMyTimerHostNotifier& iHost;
        TTimeIntervalMicroSeconds32 iDelay;
        RTimer iTimer;
        TInt iTimeEvent;
};

#endif
```

Now, let’s have a look at the definition of the Interface class MyTimerHostNotifier. It just consists of the virtual callback function MyTimerEventNotify, which is used to notify the calling class that the timer has finished. In the examples we put this declaration in the same Header file as the declaration of MyTimer, but it could also be put in a separate header file.

```cpp
class MMyTimerHostNotifier
{
    public:
        virtual void MyTimerEventNotify(TInt aEventCode) = 0;
};
```

Class MyTimer and MyTimerHostNotifier are now complete. Let’s construct an instance of our class MyTimer. Therefore we go into the Header file of our basic class BAppAppui, from which we construct the CMyTimer object. To complete the construct, class BAppAppui inherits directly from our Interface class MyTimerHostNotifier, in return we must define the function 'MyTimerEventNotify'.

```cpp
#ifndef BAPPAPPUI_H
#define BAPPAPPUI_H

#include <aknappui.h>
#include "MyTimer.h"

class CBAppContainer;

class CBAppAppUi : public CAknAppUi, public MMyTimerHostNotifier
{ 
```

Author: Gian Rossetti
Here you see, how we implemented the callback function 'MyTimerEventNotify' in BAppAppui. By calling this function, a simple Note will be displayed in the GUI.

```cpp
void CBAppAppUi::MyTimerEventNotify(TInt aEventCode)
{
    // popup a note
    TBuf<64> outBuf(_L("Return of TimerEvent: "));
    outBuf.AppendNum(aEventCode);
    C AkronInformationNote* informationNote = new (ELeave) C AkronInformationNote;
    informationNote->ExecuteLD(outBuf);
}
```

As class BAppAppui extends MyTimerHostNotifier, BAppAppui is in fact a MMyTimerHostNotifier object. The function 'NewL' of class MyTimer requests a MMyTimerHostNotifier object, therefore we just pass a reference to ourselves to the CMyTimer object. Once the object is constructed, we call the function 'After', which triggers the timer. By calling 'Cancel', the function 'DoCancel' in class CMyTimer is invoked.

```cpp
#include "BAppAppui.h"

public:
  void ConstructL();
  "CBAppAppUi() ;

  //From MMyTimerHostNotifier
  void MyTimerEventNotify(TInt aEventCode);

private:
  void HandleCommandL(TInt aCommand);

private:
  CBAppContainer* iAppContainer;
  C MyTimer* iMyTimer;
};
#endif
```

Author: Gian Rossetti
9 GSM API

9.1 Overview

Similar to the Socket-Server and the Comms-Server, there is a ETel Server infrastructure that allows to deal with the telephony API. ETel provides control over standard phone features such as answering and initiating calls. In addition the phone and line status and capabilities can be queried. The ETel Server provides the management of multiple concurrent connections and offers the capability to load plugin-modules (also called TSY), that are used to add support for several hardware types, without affecting the client side API.

![GSM Stack Diagram](image)

Figure 7: GSM Stack

9.2 ETel Implementation

The parent class of all telephony related classes is the ETelServer. The three subclasses listed below are being used when interacting with ETelServer:

- RPhone
- RLine
- RCall

9.2.1 RPhone

A telephony device is represented by the class RPhone. It’s API provides access to the status and the capabilities of the phone and provides a notification
mechanism, which is triggered when the phones status changes.

9.2.2 RLine
A phone can have one or more lines. RLine objects represents a single line. Again the capabilities, status and status changes can be gained by accessing the RLine object.

9.2.3 RCall
RCall is the class where real telephony calls are implemented. The functionality of dialing a number, hanging up a call or receiving incoming calls can be implemented using the class RCall. RCall objects represent a single active call (Each line might have zero or one active calls). Same as in the two classes described above, RCall can be used to notify status changes as well as call capabilities.

9.3 Using ETel client API
9.3.1 Initialising basic objects

There is not much to be initialised, first of all we must establish a connection to RTelServer and load the TSY-modules.

```cpp
CallModule.cpp

void CCallModule::ConstructL(MCallModuleNF* creatorClass)
{
    _LIT(KTsyName, "phonetsy.tsy");
    this->iState = EIdle;
    this->iCreaterClass = creatorClass;
}
```

9.3.2 Setting up a call

When setting up a call, we first cancel any asynchronous task and close every connection to classes, which will be used later on. Note, that this is just to make sure, that no remaining active instances are kept. As we might initialise several calls, one after the other, it could indeed occur, that some connections to classes RPhone, RLine and RPhone are still active. Now we can "Open" the classes described above and call the asynchronous function "Dial".

```cpp
CallModule.cpp

void CCallModule::InitCall(const TDesC& telNbr)
{
    this->iCall.Close();
    this->iLine.Close();
    this->iPhone.Close();
    TInt numberPhones;
    User::LeaveIfError(iServer.EnumeratePhones(numberPhones));
    if (numberPhones < 1)
    {
        User::Leave(KErrNotFound);
    }
    User::LeaveIfError(iServer.GetPhoneInfo(0, this->iTelPhoneInfo));
    User::LeaveIfError(iPhone.Open(iServer, this->iTelPhoneInfo.iName));
    User::LeaveIfError(iLine.Open(iPhone, this->iPhoneLineInfo.iName));
    User::LeaveIfError(iCall.OpenNewCall(iLine, this->iCallName));
    this->iState = EDial;
    iCall.Dial(iStatus, telNbr);
    SetActive();
}
```
9.3.3 Listening for incoming call

Listening for incoming calls is very similar to initialising a call, the first steps are exactly the same, at the very end, we call the asynchronous function "NotifyIncomingCall".

```cpp
void CCallModule::WaitingForIncomingCall()
{
    this->Cancel();
    this->iCall.Close();
    this->iLine.Close();
    this->iPhone.Close();

    TInt numberPhones;
    User::LeaveIfError(iServer.EnumeratePhones(numberPhones));
    if (numberPhones < 1)
    {
        User::Leave(KErrNotFound);
    }

    User::LeaveIfError(iServer.GetPhoneInfo(0, this->iTelPhoneInfo));
    User::LeaveIfError(iPhone.Open(iServer, this->iTelPhoneInfo.iName));
    User::LeaveIfError(iLine.Open(iPhone, this->iPhoneLineInfo.iName));

    this->iLine.NotifyIncomingCall(iStatus, this->iCallName);
    this->iState = EWaitingForIncomingCall;
    SetActive();
}
```

9.3.4 Answering incoming call

Answering incoming calls is straightforward. There exists an asynchronous function called "AnswerIncomingCall". This is exactly what we need. Because it is just three lines of code, we did not place it in a separate helper function. The code below is implemented directly in the "RunL" function (inherited by CActive).

```cpp
... //somewhere in the nirvana ;-)  
this->iState = EAnsweringIncomingCall;  
this->iCall.AnswerIncomingCall(iStatus);  
SetActive();  
... 
```

9.3.5 Observing call state changes

Other to answering incoming calls, we are going to loop over this function until any side is hanging up. Therefore this piece of code is placed in a separate helper function.

```cpp
void CCallModule::HookStateChange()
{
```
9.3.6 Hangig up a call

Hanging up a call is the last function to be implemented. We must make sure, that we just terminate the call, if there is really any opened call. This time we used the synchronous version of the "HangUp" function, as this is the last state in the call state machine and there is no strong reason not to do so.

```cpp
void CCallModule::TerminateCall()
{
    // cancel any remaining activities
    this->Cancel();
    // terminate call, if it is connected
    RCall::TStatus status = RCall::EStatusUnknown;
    iCall.GetStatus(status);
    if (status == RCall::EStatusConnected)
    {
        // indicate, that call is being hung up
        this->iCreaterClass->Call_Module_Notifier(
            MCallModuleNF::E_Call_HangingUp_Notifier, status);
        // terminate call, synchronous version
        this->iCall.HangUp();
        // indicate, that call has been hung up
        this->iCreaterClass->Call_Module_Notifier(
            MCallModuleNF::E_Call_Terminated_Notifier, status);
        // call is now terminated, the creating class can now
        // safely delete its instance of this object
    }
}
```
10 SMS & MMS API

Symbian OS provides powerful capabilities for handling messages of type SMS, MMS and email. The used API is known under the subject Messaging Architecture. The messaging architecture was defined by Series 60, and for a long time it was only available on that platform. Fortunately, manufacturers like SonyEricsson has decided in the end of 2004 to offer an API plugin emulating Series 60’s messaging architecture interface on UIQ devices.

10.1 Using Messaging Framework

Using the Messaging Framework is an easy work. There exists a basic implementation, which works for bot, SMS and MMS.

- Inherit the interface MMsvSessionObserver to your class.
- Create a CMsvSession, which represents the communication between your class an the Message Server thread
- Callbacks from CMsvSession (Message Server) will be fired toMMsvSessionObserver-Interface

//*** header file ***/
class CSMSEngine: public MMsvSessionObserver
{
    ...
    public:
        // functions from MMsvSessionObserver
        void HandleSessionEventL(TMsvSessionEvent aEvent,
            TAny* aArg1, TAny* aArg2, TAny* /*aArg3*/);

    private:
        // Client session on the message server
        CMsvSession* iSession;
        // Mtm client registry for creating new mtms
        CClientMtmRegistry* iMtmReg;
    ...
}

//*** cpp file ***/
void CSMSEngine::ConstructL(MSMSEngine* host)
{
    //will call EMsvServerReady
    iSession = CMsvSession::OpenAsyncL(*this);
}

void CSMSEngine::CompleteConstructL()
{
    iMtmReg = CClientMtmRegistry::NewL(*iSession);
    iMtm = iMtmReg->NewMtmL(KUidMsgTypeSMS);
}

void CSMSEngine::HandleSessionEventL(TMsvSessionEvent aEvent,
    TAny* aArg1, TAny* aArg2, TAny* /*aArg3*/) {
    switch (aEvent)
    {
        case EMsvServerReady:
        {
            //EMsvServerReady
        }
    }
Dependent of preferred technology, you can create **CBaseMtm** object, a high level interface for accessing and manipulating messages.

//creating CBaseMtm for SMS
iMtm = iMtmReg->NewMtmL(KUidMsgTypeSMS);

//creating CBaseMtm for MMS
iMtm = iMtmReg->NewMtmL(KUidMsgTypeMultimedia);

//using CBaseMtm for SMS
CSmsClientMtm* smsMtm = STATIC\_CAST(CSmsClientMtm*, iMtm);

//using CBaseMtm for MMS
CMmsClientMtm* mmsMtm = STATIC\_CAST(CMmsClientMtm*, iMtm);

Now you have the basics to do more operations like deleting messages, sending, receiving, and iterating the inbox as well as the outbox...
10.2 Example

The following example SMSApp implements a simple SMS-Router, based on a self defined syntax within SMS content.

General: "AppName@ControlString@Arg1@Arg2"

Example: "SMSApp@routing@0791234567@Symbian Programmierer" sends Arg2 to phone number defined in Arg1.

For MMS-Handling, which covers lots of additional details you’ll find a very good example in the SDK-Example-Library.
11 Bluetooth

11.1 Overview

Bluetooth technology is an open specification for wireless communications of data and voice. It is based on a low-cost short-range radio link (2.4 GHz) which operates on a globally-available radio frequency. Bluetooth uses FHSS and splits its frequency-band into 79 1-MHz channels. It can be used to connect to other mobile phones, computers, network access points and other devices. Class 2 devices have a range of about 10 meters, Class 1 devices even about 100 meters. Although the Bluetooth-Specification supports both, synchronous and asynchronous protocols, not all protocols can be accessed using the existing SDK’s. Nevertheless the implementation of a variety of services, such as voice and video can be done.

11.2 Symbian OS Bluetooth architecture

![Bluetooth Architecture Diagram](image)

Figure 10: Bluetooth Architecture

The diagram above describes the Bluetooth Stack architecture. Basically it’s a layered model and the most important layers are the following:
Bluetooth Radio This layer, at the very bottom, is the Bluetooth radio transceiver. Often it is also called Physical layer, although this is a bit misleading, because you might assume, that this is the only layer implemented in hardware, which is not correct.

Baseband Directly above, you’ll find the Baseband. This layer is responsible for establishing connections and controlling the data transfer using the Bluetooth Radio layer.

Bluetooth Protocols On top of the Baseband, there are several Protocols, some of them are always present, others vary on the Bluetooth implementation. Together it is a set of protocols, very similar to the ones used in IP-networks (PPP, IP, TCP). They provide support for Bluetooth communication at different levels of reliability (TCP vs UDP) and basic functionality.

Bluetooth Profiles Topmost on the Bluetooth stack there are the Bluetooth Profiles. These are high-level software components, that implement services, that ensure correct interoperability with other Bluetooth devices. Not all Bluetooth specifications (Bluetooth v1.1 vs Bluetooth v2.0) support the same set of Bluetooth Profiles. Neither do all brands support all profiles.

11.3 Bluetooth Application development

There are several components you need to deal with in order to develop a simple Bluetooth application:

11.3.1 Bluetooth Protocols

LMP The Link Manager Protocol handles the behavior of the Bluetooth link. It controls the baseband device, manages security and allows service discovery.

L2CAP The Logical Link Control and Adaptation Protocol offers connection-oriented and connectionless data services between the baseband device and the upper layers. It is responsible for stream segmentation and reassembly, which allows the upper-layer protocols to transmit data packets larger than the baseband can handle.

SDP The Service Discovery protocol allows Bluetooth devices to discover services offered by remote devices. Broadcast queries can be sent and the responses parsed, in order to determine which services are offered by the remote devices.

RFCOMM Radio Frequency Communications is a protocol which can be found in lots of protocol stacks (such as IrDA). It is a reliable transport protocol that emulates a RS-232 serial port communication.

11.3.2 Bluetooth Profiles

Object Push profile allows a Bluetooth device to send and receive OBEX objects such as vCard (Business Card) and vCal (Calendar)items.

Headset profile defines the requirements for Bluetooth devices necessary to support Headsets. The most common examples of such devices are headsets, mobile phones and PCs.
Dial-up Networking profile Two main scenarios are implemented: The usage of a mobile phone (i.e. its modem) by a computer as a wireless modem for connecting to dial-up internet access server and the usage of mobile phones by a computer in order to receive data calls.

Fax profile A Bluetooth cellular phone or modem may be used by a computer as a wireless fax modem to send or receive a fax message.

File Transfer profile Typically this protocol allows remote device browsing, transferring and manipulating objects on/with another Bluetooth device.

Serial Port profile defines the requirements necessary for setting up emulated RS232 serial cable connections using RFCOMM between two peer devices. The scenario covered by this profile deals with legacy applications using Bluetooth as a cable replacement, through a virtual serial port abstraction (which in itself is operating system-dependent). Often this profile is used as data carrier for other profiles and applications.

Cordless Telephony profile This profile defines the features and procedures that are required for interoperability between different units active in the 3-in-1 phone use case. 3-in-1 phone is a solution for providing an extra mode of operation to mobile phones, using Bluetooth as a short-range bearer for accessing fixed network telephony services via a base station.

11.3.3 Service Advertisement and Service Discovery
A Bluetooth device may offer services to other devices or connect itself to remote devices and make use of their services. A Bluetooth device that offers services to others is known as server, and one that makes use of remote services is known as client. A Bluetooth Server makes services through a process, called ServiceAdvertisement. Information about the particular services is provided to any other device, that query it. On the client side, an attempt to make a Bluetooth connection is typically done as follows: The client is searching for local Bluetooth enabled devices and if found, querying the services provided by the remote device. This functionality is done by the ServiceDiscovery.

11.4 Basic Example
As the code for the Bluetooth implementation is rather complex, it will not be explained here. Please refer to the example Bluetooth application.
A Symbian Installation Guide

A.1 Overview
Install the SDK according to your Phone. All SDKs are listed in the next paragraph. Make sure, ActivePerl is installed. Most Nokia Series60 and Series80 SDK do not include ActivePerl. When installing an UIQ SDK, ActivePerl gets installed automatically. You can download and install the latest release of ActivePerl using the link: [www.activestate.com/ActivePerl](http://www.activestate.com/ActivePerl)

Below there is a summary of all SDK’s available.

**A.2 Device Overview**

A.2.1 Nokia Series 60

Phones based on S60 3rd (Symbian OS v9.2)

- Nokia 5700
- Nokia E90

Phones based on S60 3rd (Symbian OS v9.1)

- Nokia E50
- Nokia E60
- Nokia E61
- Nokia E61i
- Nokia E65
- Nokia E70
- Nokia 3250
- Nokia 5500 Sport
- Nokia 6290
- Nokia N71

Figure 11: Bluetooth state machine
• Nokia N73
• Nokia N75
• Nokia N76
• Nokia N77
• Nokia N80
• Nokia N91
• Nokia N92
• Nokia N93i
• Nokia N95

Phones based on S60 2nd Edition FP3 (Symbian OS v8.1)
• Nokia N70
• Nokia N72
• Nokia N90

Phones based on S60 2nd Edition FP2 (Symbian OS v8.0a)
• Nokia 6630
• Lenovo P930
• Nokia 6680
• Nokia 6681
• Nokia 6682

Phones based on S60 2nd Edition FP1 (Symbian OS v7.0s enhanced)
• Nokia 3230
• Nokia 6670
• Nokia 7610
• Nokia 6620
• Nokia 6260
• Panasonic X700
• Panasonic X800
• Samsung SDH-D720

Phones based on S60 2nd Edition (Symbian OS v7.0s)
• Nokia 6600
• Samsung SDH-G720
• Samsung SDH-G730

Phones based on S60 1st Edition (Symbian OS v6.1)
• Nokia 7650
• Nokia 3650, 3600
• Nokia 3660, 3620
• Nokia N-Gage
• Nokia N-Gage QD
• Sendo X
• Siemens SX1
• Samsung SDH-D720

A.2.2 Nokia Series 80

Phones based on Nokia Series 80 Developer Platform 2.0

• Nokia 9300
• Nokia 9500

A.2.3 Other Nokia Symbian Phones with special SDK’s

• Nokia 7710
• Nokia 9200 9210 9210i 9290

A.2.4 SonyEricsson UIQ

Phones based on UIQ 3.0 (Symbian OS v9.1)

• Sony Ericsson M600i
• Sony Ericsson P990
• Sony Ericsson W950
• Motorola MOTORIZR Z8

Phones based on UIQ 2.1 (Symbian OS v7.0)

• Sony Ericsson P910
• Sony Ericsson P900
• Motorola A1000FOMA M1000Arima U308

Phones based on UIQ 2.0 (Symbian OS v7.0)

• Sony Ericsson P800, P802
• Motorola A920
• Motorola A925
• BenQ P30
A.3 IDE adaptations

A.3.1 Codewarrior

By installing the UIQ SDK, you need to adapt a single file in the SDK. If you do not adapt this file, you will get the following Error output in the Codewarrior IDE:

```
Codewarrior Console
Error : Too many include paths! Please reduce the number of include paths or use non-recursive paths
Error : Adapter initialization failed
Error : Too many include paths! Please reduce the number of include paths or use non-recursive paths
Error : Unable to load include paths
Error : Compile failed
```

CodeWarrior gets it project settings from SDK scripts. Locate the file “C:/Symbian/UIQ_21/epoc32/tools/ide_cw.pm” Change the the value “Recursive” to false (in the addSystemSearchPaths function, not in the addUserSearchPaths).

```
ide_cw.pm

textSetting(accesspath, "Recursive", "false");
```

Alternatively you could (instead of the Action described above) adapt the XML project template file. This file is contained in the same folder as above. In fact there are two different templates, it’s recommended just to adapt the one that you are actually using:

- `scw_project_template.xml`
- `scw_project_template_v2.xml`

Again, change it’s recursive value to false

```
<SETTING>
  <NAME>Recursive</NAME>
  <VALUE>false</VALUE>
</SETTING>
```

A.3.2 Carbide

Keep the directory where the whole Symbian project is located and the Carbide workspace strictly apart. The Symbian project should not be located within the workspace.

A.4 Common Errors

A.4.1 Codewarrior

- CodeWarrior does just allow to compile code from the Windows installation drive. In most cases this is c:Using networkd drives and other drives is not supported.
• You won't be able to compile your code, if the Emulator Application on your PC is still running. As long as the .app file is used for the emulator, it can't be replaced by a compile process.

• If you transfer your *.sis file to the handset, via Bluetooth, it will just work out, if the same item (same name ie. BApp.sis) is not already present in the INBOX of your Mobile phone.

• Symbian 9.x SDKs require a more recent versions of Active Perl and Java Runtime Environment than the other SDKs. Active Perl 5.6.1 build 631 for SDK build tools is required. Newer or older versions of Perl may not be fully compatible with Symbian toolchain. Perl must be installed before using the SDK to build applications. Using older versions of Perl you might get a Compiler-error like "Can't locate warnings.pm in @INC..."
  In addition Java Runtime version 1.5.0 build 06 is required.

A.4.2 Carbide

• Carbide can not handle spaces in the workspace path. Also the source file directory is not allowed to include spaces in it's path.