Localization of Mobile Platforms

Waqar Ahmad
Rabia Sirhindi
Farah Adeeba
Sarmad Hussain

Center for Language Engineering (CLE)
Al-Khawarizmi Institute of Computer Science (KICS)
University of Engineering & Technology (UET)

www.cle.org.pk

www.idrc.ca

Published by

Center for Language Engineering (CLE)
Al-Khawarizmi Institute of Computer Science (KICS)
University of Engineering & Technology (UET)
Lahore, Pakistan

ISBN: 978-969-9690-00-8

This work has been carried out with the aid of a grant from the International Development Research Center (IDRC), Ottawa, Canada, administered through the Centre for Language Engineering(CLE), Al-Khawarizmi Institute of Computer Science(KICS), University of Engineering & Technology(UET) Lahore, Pakistan.

Preface

Increasing penetration of mobile devices has resulted in their use in diverse domains such as education, health, entertainment, business, sports, and social networks. However, lack of appropriate support for local languages, which use complex scripts, on mobile devices is constraining people across developing Asia and elsewhere from using their mobile devices effectively. There are some ad hoc solutions for certain scripts, but what is needed is a comprehensive and scalable framework which would support all scripts. The Open Type Font (OTF) framework is now being widely used for supporting complex writing systems on computing platforms. If support for OTF is also enabled on mobile devices, it would allow them to support complex scripts.

This work describes a detailed methodology to enable complex scripts on mobile devices. The case study that has been discussed throughout the book covers mechanisms to enable localization support on Symbian platform. However, the approach discussed for Symbian platform can be extended to add complex scripts on other mobile platforms such as Android. The book discusses how to enable localization support — taking Pango, an open source rendering engine, and porting its language specific modules to Symbian platform in order to enable support for Open Type Fonts — and the localization process.

After a general overview in Chapter 1 (Introduction), Chapter 2 (Software Development for Smartphones—An Overview) covers a general overview of software development for mobile platforms and localization support available on existing mobile platforms. Chapter 3 (Symbian Operating System Architecture) discusses the layered architecture of Symbian operating system and the key design patterns of the operating system. Chapter 4 (Setting up Development Environment) lists steps to set up the environment for application development on Symbian platform, Chapter 5 (Symbian Application Framework) explains the architecture of various types of Symbian Applications, and Chapter 6 (Developing a Hello World Application) describes creation and structure of a HelloWorld application using view-switching architecture of the Symbian platform. Chapter 7 (Localized SMS Application) gives design and development of an SMS application that supports text entry in a complex Asian script and, finally, Chapter 8 (Pango: A Viable Open Source Font Rendering Engine for Smartphone Platforms) explains how language specific modules of Pango Cairo can be ported to Symbian platform.

This work has been made possible with the support of the PAN Localization project (www.PANL10n.net) grant by IDRC Canada (www.idrc.ca). We would like to thank the management of National University of Computer and Emerging Sciences, Al-Khawarizmi Institute of Computer Science and University of Engineering and Technology for making the work possible.

Authors

PAN Localization Project

Enabling local language computing is essential for access and generation of information, and also urgently required for development of Asian countries. PAN Localization project is regional initiative to develop local language computing capacity in Asia. It is partnership, sampling eight countries from South and South-East Asia, to research into the challenges and solutions for local language computing development. One of the basic principles of the project is to develop and enhance capacity of local institutions and resources to develop their own language solutions.

The PAN Localization Project has three broad objectives:

- To raise sustainable human resource capacity in the Asian region for R&D in local language computing
- To develop local language computing support for Asian languages
- To advance policy for local language content creation and access across Asia for development

Human resource development is being addressed through national and regional trainings and through a regional support network being established. The trainings are both short and long term, to address the needs of relevant Asian community. In partner countries, resource and organizational development is also carried out by their involvement in development of local language computing solutions. This also caters to the second objective. The research being carried out by the partner countries is strategically located at different research entry points along the technology spectrum, with each country conducting research that is critical in terms of the applications that need to be delivered to the country's user market. Moreover, PAN Localizations project is playing an active role in raising awareness of the potential of local language computing for the development of Asian population. This will help focus the required attention and urgency to this important aspect of ICTs, and create the appropriate policy framework for its sustainable growth across Asia.

The scope of the PAN Localization project encompasses language computing in a broader sense, including linguistic standardization, computing applications, development platforms, content publishing and access, effective marketing and dissemination strategies and intellectual property right issues. As the Pan Localization project researches into problems and solutions for local language computing across Asia, it is designed to sample the cultural and linguistic diversity in the whole region. The project also builds an Asian network of researchers to share learning and knowledge and publishes research outputs, including a comprehensive review at the end of the project, documenting effective processes, results and recommendations.

Countries (and languages) directly involved in the project include Afghanistan (Pashto and Dari), Bangladesh (Bangla), Bhutan (Dzongkha), Cambodia (Khmer), China (Tibetan), Indonesia (Bahasa Indonesia), Laos (Lao), Mongolia (Mongolian), Nepal (Nepali), Pakistan (Urdu, Torwali, Sindhi) and Sri Lanka (Sinhala and Tamil). The project started in January 2004. Further details of the project, is partner organizations, activities and outputs are available from its website at www.PANL10n.net.

Table of Contents

1 Introduction

Mobile phone penetration is increasing worldwide as well as in developing countries of Asia at a rapid pace [1, 2]. While past usage of mobile devices has mostly been for voice, there is a significant increase in text and other data services using smart-phones [10]. It is expected that more than 85% of mobile handsets will be equipped for mobile web access by the end of 2011 [1], as many smart-phones today have processing power and other capabilities comparable to desktop computers of early 1990s.

As the hardware capabilities of mobile devices improve, they are increasingly being used in areas like education, health, entertainment, news, sports, and social networks. This usage of smart-phones requires that text and other data services are made available in local languages. However, most of the mobile devices that are currently in use only support Latin script. There is limited or no support available for many other languages and scripts, specifically those of developing Asia. The devices generally support basic Latin, bitmap and True Type Fonts (TTF). Most Asian languages scripts, on the other hand, are very cursive, context sensitive and complex [3, 4] and can only be realized using more elaborate font frameworks, e.g. Open Type Fonts (OTF) [7]. Such frameworks are not supported on most mobile devices and smart-phones at this time. Many people in developing Asia are only literate in their own languages and are, therefore, unable to utilize their mobile devices for anything other than voice calls. Developing font support is an essential pre-cursor to make content available in local scripts. Once support is in place, content can be created, allowing people to utilize the additional capabilities of mobile phones for their socio-economic gains.

Whether focusing on iPhone [9], Symbian based Nokia Phones [6], Google Android [8], Windows Mobile [7], or Blackberry, the worldwide web is full of queries and posts showcasing the needs and concerns of developers and end-users, which are looking for particular language support on their devices. While there is extensive localisation support for desktop computers, mobile devices are lagging behind. Smart-phone software developers try to find workarounds for resolving localisation issues and sometimes achieve limited success. However, total success can only be achieved if the underlying device platform provides comprehensive support. If the underlying platform has limitations, these are also reflected in the workarounds produced by software developers. A major problem is that mobile platforms provide limited software internationalisation support and therefore, localisation for certain languages may become very difficult.

In this book we have suggested a solution for solving some of the problems associated with the support of complex Asian scripts on mobile devices using Pango—an open source library for text layout and rendering with an emphasis on internationalisation [5]. A Research and development project has been carried out with a focus on evaluating the viability of Pango as a text layout and rendering engine on mobile platforms. For this project, Symbian was chosen as the mobile platform. The project has two primary components: one component deals with porting the script specific modules of Pango to the Symbian platform; the other component is the development of an application that can send/receive SMS in local languages using Pango on mobiles.

Although all of the language specific modules of Pango are ported successfully to Symbian platform, extensive testing is performed for Urdu and an initial level of testing is performed for Khmer and Hindi. The results of the tests are quite promising and confirm the viability of Pango as a font engine for mobile devices. The SMSLocalized application contains features specialized for local scripts. This application has been tested for Urdu; however, the architecture of the application is very flexible and allows quick application customization for other languages.

This book presents the relevant background and details of this work. Chapter 2 (Software Development for Smartphones—An Overview) covers a general overview of software development for mobile platforms and localization support available on existing mobile platforms. Chapter 3 (Symbian Operating System Architecture) discusses the layered architecture of Symbian operating system and the key design patterns of the operating system. Chapter 4 (Setting up Development Environment) lists steps to set up the environment for application development on Symbian platform, Chapter 5 (Symbian Application Framework) explains the architecture of various types of Symbian Applications, and Chapter 6 (Developing a Hello World Application) describes creation and structure of a HelloWorld application using view-switching architecture of the Symbian platform. Chapter 7 (Localized SMS Application) gives design and development of an SMS application that supports text entry in a complex Asian script and, finally, Chapter 8 (Pango: A Viable Open Source Font Rendering Engine for Smartphone Platforms) explains how language specific modules of Pango Cairo can be ported to Symbian platform.

References

[1]. MobiThinking(2010) Global Mobile Stats: all Latest Quality Research on Mobile Web and Marketing [online], available : http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats [accessed 16 Aug 2010].

[2]. International Telecommunication Unit (2010) ITU sees 5 Million Mobile Subscription Globally in 2010 [online], available: http://www.itu.int/itu-d/newsroom/press_release/2010/06.html [accessed 18 Aug 2010].

[3]. Hussain S. (2003) "Computational Linguistics in Pakistan: Issues and Proposals", in proceedings of *European Chapter of the Association for Computational Linguistics (EACL), Workshop in Computational Linguistics for Languages of South Asia*, Hungary.

[4]. Wali A. and Hussain S. (2006) "Context Sensitive Shape-Substitution in Nastaliq Writing System: Analysis and Formulation", in proceedings of *International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE2006)*.

[5]. Taylor O. (2001) "*Pango: Internationalized Text Handling*" [online], available: http://www.lwn.net/2001/features/OLS/pdf/pdf/pango.pdf [accessed 10 Jun 2009].

[6]. Forum.Nokia Users (2009) Discussion Board [online], available:http://www.developer.nokia.com/Community/Discussion [accessed 7 Oct 2009].

[7]. Windows Embedded CE 6.0 R3 Development (2011) [online], available: http://www.microsoft.com/windowsembedded/en-us/develop/windows-embedded-ce-6-for-developers.aspx [accessed 10 Dec 2011].

[8]. Google (2010) Android 2.2 Platform [online], available: http://developer.android.com/sdk/android-2.2.html [accessed 10 Oct 2010].

[9]. Apple (2010) iPhone 4 Technical Specifications [online], available: http://www.apple.com/iphone/specs.html [accessed 20 Aug 2010].

[10]. adMob (2010) AdMob Mobile Metrics [online], available:http://metrics.admob.com[15 Aug 2010].

2 Software Development for Smartphones

A smartphone combines features of a mobile phone and a handheld computer into a single device. Smartphones have an operating system, application development framework, local data storage mechanism, and a number of other useful features such as internet connectivity, e-mail, contact management software, games, media software and additional hardware components like a camera. One key feature that distinguishes smartphones from ordinary phones is that smartphones allow end users install third-party software applications [6].These days the term 'mobile phone' is frequently used to refer to smartphones as well.

A smartphone has quite a few unique characteristics. A smartphone is a *personal device* which is not usually shared with others. It is a *handheld device*; small in size, battery powered, and has wireless connectivity. It is also a *communication device* which is switched on and connected with some network almost all the time. As a *general purpose device*, it is used for voice calls [7].

Smartphones are pervasive globally in recent times. Increasing penetration of Smartphones has resulted in tremendous increase in software development for these platforms. Software development for mobile platforms, however, is quite complex and differs from traditional software development for desktops and servers in a number of noteworthy aspects.

2.1 Application Development for Smartphones—An Overview

Following sections highlight some of the unique aspects of mobile software development:

Large Number of Platforms

There is considerably large number of distinct operating systems for mobile devices. Major operating systems include Google Android, Windows Phone, Symbian, iOS, BlackBerry and Linux. Platforms are different from each other in their characteristics such as programming languages and application architectures. An application written for one platform may require complete redesign and redevelopment when ported to other platforms. Even though Java ME, a Java platform for mobile devices, was created with the goal of achieving application portability on mobile platforms, the dream of write-once-and-run-anywhere has not been realized for mobile platforms so far.

Development vs. Deployment Platform

For desktop applications, it is possible to develop and test an application on a platform which is same as target deployment platform. For instance, application development, testing and deployment can be done on Windows Platform. For mobile applications, however, development is done on desktop operating systems such as Windows, Linux, and Mac OS X and deployment is done on mobile device platforms such as iOS, Symbian and Android.

Application Debugging and Testing During Development

For desktop operating systems such as Windows and Max OS, mobile platform simulators/emulators are available. Simulators/Emulators enable developers to debug and test their applications on development platform before they are deployed on target mobile platforms. Many of the simulators, however, do not fully simulate the target handset platforms. Therefore, an application which works on a simulator correctly may not work properly on a real device and may require further tweaking.

Another important drawback associated with testing on simulators is that usability testing cannot be reliably performed. In fact, results of any usability testing done on simulators may be misleading. User experience on a real device is quite different from user experience on a simulator running on a desktop machine.

Variations in Hardware Features

Mobile devices vary significantly in their hardware capabilities such as network connectivity standards and camera qualities. Consider, for instance, an application that requires extensive use of network data connectivity for its operations; performance of such an application on an EDGE enabled devices may be considerably different from a GPRS enabled device. An application developer may have to optimize application data requirements so that it can work well when available data bandwidth is limited. Similarly some devices may support Bluetooth, WiFi, Camera and others may support only some of these.

Limited Processing Power

Many of the mobile devices available in the market today have very limited processing power when compared to average desktop computers. This requires mobile application developers to be very careful in designing and coding applications so that application uses minimal computational power. For instance, algorithms of higher complexity may render a mobile device non-responsive. Similarly any application that requires extensive network communications for data operations may deteriorate the performance of the device. Though new devices having much better processing power are coming into the market, many of the existing devices in market have limited processing power.

Limited Memory

Many of the mobile devices available in the market today have very limited memory available for applications when compared to that of average desktop computers. This requires mobile application developers to be very careful in designing and coding mobile applications so that memory requirements of application remain small. Though quite a few new devices have relatively large amount of memory, application size (also called footprint) must be kept small if application developers want to target large base of existing handsets.

Not only the size of application itself should be small, its data requirements must also be minimal i.e. the data that is stored by the application on device should be kept in a range so that memory overflow does not occur.

Limited Battery Power

Battery power is an extremely important resource. For people on the move, it may not be possible to charge batteries frequently. Overuse of batteries also reduces their life. Therefore, applications must use minimal battery power. For instance, an application can be designed in such a way that it loads required resources only when they are required because certain resources may be required only if user performs some specific actions. Backlight, voice call, data traffic, and computational activities consume relatively large amount of battery power.

Limited Data Bandwidth

Mobile applications have access to a data network which is, most of the time, quite constrained. For instance, EDGE offers data bandwidth quite lower than DSL or WiMax does. Due to availability of limited data bandwidth on most of the deployed mobile networks today, intelligent application design may be required so that application make optimum use of network resources. Techniques such as data compression and caching may be used to reduce load on the network resources.

Input Mechanisms

Mobile handsets support various input mechanisms such as ITU-T numeric keypad, QWERTY keypad, and Touch. Large amount of text input is quite difficult using numeric keypad. Though many devices support predictable text input, yet it alleviates the text input problem only to a limited extent. QWERTY keypad on a mobile device is better than numeric keypad in its capability to input text; however, small size of QWERTY keypad does not let it become as efficient as the standard typewriter keyboard. Text input using Touch method is also not very efficient. Therefore, a good mobile application would be designed in a way that it enables users make selections rather than input large amount of text.

Small Screen Size

Most of the mobile handsets have smaller screen size than that of desktops. Therefore, information that can be displayed on screens of common mobile phones is very small as compared to what can be displayed on screens of common desktop computers. Intelligent interface design is required so that users can access application features with ease. Therefore, designing an interactive application for a mobile device having small screen is quite challenging.

Small screen size plays significant role while defining number and type of features that can be built into a mobile application. For instance, application should not require the user to perform a large number of navigation steps to access certain features. It is generally recommended that depth of navigation does not exceed 4 levels.

Variations in Screen Size

While small screen size poses its specific challenges, variations in screen sizes of mobile devices make development further challenging. Therefore, application designers need to design their applications in a way that applications can either intelligently adjust themselves according to screen size or they have to perform individual application tweaking for a family (or even for a single device) of devices.

Therefore, an application that is required to be developed for multiple different devices may require creation of different specifications to accommodate variations in device screen sizes. For instance, UI specifications developed for an application of a device that supports QWERTY keyboard based input may vary greatly from UI specifications of the same application for touch enabled device.

Application Usability

Mobile phones are used by both tech savvy as well as non-technical users. It cannot be assumed that end users would even be skilled at using a smartphone device. Therefore, application usability may become a key factor for consumers when they make application purchase decision. Due to differences in form factors of mobile devices and variety of context of application use, unique user modeling and interaction design techniques may be required for building usable mobile applications.

Operations in Multiple Data Network Modes

Mobile devices can operate in various modes depending upon the availability of data network. A user may have deliberately put the device in offline mode or device goes offline just because of unavailability of data network. It is also possible that a device only intermittently connects to a data network. These characteristics of data networks impact the design of applications that are dependent on availability of data connectivity for performing their operations. A good application would be robust enough to operate in most of these modes which would require a robust data synchronization engine. For instance, an application that requires data-upload operations on a network server may behave in following ways:

- When connected, it uploads data directly on server.
- When offline, it stores data in local device memory and uploads data on server later when data connection is available.

Devices and Type of Applications

An application may not be suitable for certain type of devices. For instance, an application that requires users to input relatively large amount of data may be suitable for a device having QWERTY keypad but not for a device having only Numeric keypad. Therefore, developers must ensure that they do not try to make a device into something it was not designed for. Such features which are not befitting to a device may badly impact the user experience.

Target Handsets

Target list of handsets (devices) include the list of devices on which application will be deployed when in production. It is very important to decide the target list of devices as early as possible during application development lifecycle. Variations in devices significantly impact many factors such as application architecture, project scope, plan and cost. In general, as the number of target handsets (which vary in their capabilities) increases so does the cost of application development. Therefore, it is very important that the list of target handsets along with their relevant characteristics is known upfront.

Information about characteristics of target handsets may be obtained from sources such as device vendors and device profile databases such as WURFL and DeviceAtlas. Statistics such as number of users of a particular handset may be helpful in making a decision whether to develop application for that handset or not.

Network Operator Requirements

Network operators (wireless network carriers) may also impose certain rules for launching an application from their portals. For instance, an operator may require exclusive rights on application or specific application UI branding.

Operators may customize the device like change look & feel or install additional applications. Therefore, a device which is available from an operator's deck may have configurations different from that of a same device model available from the device manufacturer. These factors must also be on check list of application developers.

Application Code Management

Multiple versions of the code may have to be produced to meet requirements of diverse handsets. This may require extensive code management during development and maintenance phases.

Application Testing and Signing

Many of the mobile platform vendors now also require that an application is approved by a competitive authority before it can be commercially installed on target handsets. Therefore, applications may have to go through a Certified Testing Process by a third party after Internal Testing has been completed by developing organization. Only successful completion of testing by third party certification authority will allow application installation on commercially available devices. For on-device testing, developers may use "developer certificates" for testing their application during development; however, for launching an application commercially, testing from an accredited authority may be required. Some of the testing processes include Java verified process, Symbian signed, and Apple review/verification process.

Application Distribution

Though hard for many non technical users, yet it is possible to connect a device to desktop computer and install applications manually. Many users may find it cumbersome to download an application from web on a desktop computer and then transfer it to a mobile device. Moreover, it would also become extremely difficult to install application upgrades in future.

Solution to this problem is Over the Air (OTA) installation. In case of OTA, an end user simply specifies a URL from where an application can be downloaded and installed. End user only needs to provide a URL where application is available, rest of the installation process is taken care of by application management software on the device.

Most of the mobile application stores including Android Marketplace, Apple App Store, and Nokia Ovi offer on-device built-in applications that help users perform many useful tasks such as search, purchase, and download applications from respective application portals, making application access very easy.

Application Engineering

Many of the mobile applications have relatively smaller number of features than those of desktop applications. Therefore, rapid application development (Agile) techniques like SCRUM are generally suitable for mobile application development.

Mobile Application Stakeholders

In addition to typical stakeholders of a software application i.e. application developers and end users, there are some other stakeholders specific to applications developed for mobile platforms. Device manufacturers and platform vendors may directly or indirectly control the process of application installation on the devices. Apple, for instance, has defined an approval process that must be followed to deliver an application to end users. Network operators may require that applications are launched through their designated content providers only. Therefore, in addition to requirements of typical stakeholders, concerns of multiple other parties may have to be addressed before an application can be considered ready for publishing.

2.2 Mobilize, Don’t Miniaturize

As discussed earlier, mobile devices are generally constrained in many of their capabilities as compared to desktop computers. Mobile devices, however, offer some unique features which open doors for application innovation nonexistent in desktop world. Location based services, for instance, are possible because of always-connected nature of mobile devices. While designing an application, a mobile application developer, therefore, needs to consider not only limitations of mobile platforms but also their capabilities that can be used to create innovative applications. Miniaturization is designing a mobile application considering mobile device constraints only. Mobilization, on the other hand, is designing an application considering constraints as well as capabilities of mobile devices [15].

2.3 Localization Support on Existing Mobile Platforms

Limitations in script support on mobile devices are often due to constraints specific to mobile handsets such as a small amount of memory, limited processing power and other factors. During our research, we have learnt that most of the issues related to localisation on mobile phones fall in one or more of following patterns:

- Localisation features supported on a mobile device may not be adequately documented. As a result of this, information about localisation features may only become known after acquiring and evaluating the device by installing localised software.
- Only a limited set of features for a language may be supported on the device. For instance, True Type Fonts (TTF) may be supported but not Open Type Fonts (OTF), which will results in lack of support of a various languages and their scripts.
- In mobile device system software, language support may exist at the level of menu items but may be missing at application level. For instance, a device may have an operating system with a properly localised user interface but an on-device messenger application may not allow the user to input text in a local language.
- A particular device platform may support many languages as a whole. However, when a device is released in the market, it may only be equipped with a subset of the platform's supported languages. For instance, a language-pack may be missing or the font rendering engine may be constrained by its multilingual language support.

Software developers continue trying to find workarounds for the localisation issues which are, in many ways, limited by the support provided by the underlying device platform. The following sections give an overview of the extent of localisation support on some of the major smart-phone platforms.

2.3.1 Symbian

Symbian OS, currently owned by Nokia, is the most widely deployed operating system on mobile phones. It supports application development using Java Micro Edition (Java ME) and C/C++. Symbian operating system supports a very basic level of user interface which does not make it usable by layman users. Therefore, on top of Symbian operating system, some mobile device vendors have developed rich user interfaces. Two such user interfaces are S60, developed by Nokia, and UIQ, developed by UIQ technology [8].

Symbian supports a number of languages. However, it does not support Open Type Fonts [9]. Its default engine is based on the FreeType font library [9]. The Symbian operating system, however, can be extended by plugging in an external font engine to add support for languages or scripts not already supported [8]. For instance, an engine can be developed or adapted from open source that adds support for open type fonts with complex scripts i.e. if a third party developer wants open type font support, s/he can develop and plug the font engine into the operating system which can then be used by any software application on the device.

Existing text layout and font rendering engine on Symbian platform use FreeType open source library.

2.3.2 Windows Mobile and Windows Phone

Windows Mobile is a Windows CE based operating system developed by Microsoft. Windows CE is primarily designed for constrained devices like PDAs and can be customized to match the hardware components of the underlying device [10]. Windows Mobile supports the Microsoft .Net Compact Framework for application development, which in turn supports a subset of Microsoft .Net Framework features.

According to the Microsoft website [10], WordPad, Inbox, Windows Messenger, and File Viewer applications are not enabled for complex scripts like Arabic, Thai, and Hindi.

There are some commercial solutions for localisation on the Windows Mobile platform. One such solution is Language Extender. It supports Arabic, Czech, English, Estonian, Farsi, Greek, Hebrew, Hungarian, Latvian, Lithuanian, Polish, Romanian, Russian, Slovak, and Turkish [11]. However, Open Type Fonts for other complex writing systems, e.g. Urdu Nataleeq [4] are not available.

Windows Phone, a platform launched by Microsoft in near past, supports following languages only: English, French, German, Italian, and Spanish. Complex scripts are not supported on Windows Phone platform.

2.3.3 Android

Android is a relatively new mobile software stack based on Linux. It allows application development using the Java programming language. However, a native SDK is also available from the Android developer website that can be used to develop native applications in C/C++.

Localisation on the Android platform is still limited to a few languages. Independent developers have tried workarounds with limited success [12]. There is lot of debate on language support issues on Android forums [13]. However, it has still not been made clear, officially, from Google as when support for OTF will be included.

Google Android has localisation support for German, French, and English but there is no information available about languages that use non-Latin scripts.

Current support for text layout and font rendering has been enabled on Google Android platform using FreeType open source library.

2.3.4 Apple iOS

Apple iOS is the operating system of iPhone, iPod Touch and iPad devices. The iOS SDK provides tools and technologies (Xcode and iOS simulator) required to develop, debug, install, run, and test native applications. Native applications are built using Objective-C programming language. Development on an actual device requires signing up for Apple's paid iOS Developer Program.

According to Apple [14], the Apple iPhone 4 supports a number of languages including English (U.S), English (UK), French (France), German, Traditional Chinese, Simplified Chinese, Dutch, Spanish, Portuguese (Brazil), Portuguese (Portugal), Danish, Swedish, Finnish, Norwegian, Korean, Japanese, Russian, Polish, Turkish, Ukrainian, Hungarian, Arabic, Thai, Czech, Greek, Hebrew, Indonesian, Malay, Romanian, Slovak, Croatian, Catalan, and Vietnamese.

Apple iOS comprises multiple layers and each layer contains multiple frameworks. The framework named 'Core Text Framework' is responsible for laying out text and handling fonts.

2.3.5 Monotype Imaging Rasterization and Layout Engines for Mobile Phones

Monotype imaging provides engines for font rasterization (iType Font Engine) and layout (WorldType Layout Engine) for smart-phones. The solution is ANSI C based and is available for integration with Android, Symbian and Windows CE. However, full Open Type Font support is not available in solutions provided by Monotype Imaging.

2.3.6 Other Smart-phone Platforms

In current research work, other smart-phone platforms like RIM Blackberry, Palm WebOS etc. are not investigated in detail from localisation perspective. However, their limitations from localisation perspective seem to be similar to those mentioned above for other platforms, as discussed on online developer and end-user forums [11].

2.4 Extending localization capabilities of a mobile phone platform

Depending upon its architecture, a mobile phone platform can be extended to include additional localization support in either of following ways:

- **Plug in** a new text layout and font rendering engine into the platform.
- Port a text layout and font rendering engine as a **library** to the platform.

In the former case i.e. when a text layout and font rendering engine is plugged into an operating system as an additional font engine, any application available on the device would be able to use it. For instance, if system allows replacing an existing font engine with a new one, all device applications would automatically be able to use the new engine. This approach, however, is only possible if mobile platform supports it and Symbian is one such platform that allows this. A specific example of this approach is to adapt Pango library to meet Symbian font engine interface requirements. Other major mobile platform i.e. Google Android, Apple iOS, and Microsoft Windows Phone do not allow plugging in a new text layout and font engine.

In the latter case i.e. porting a text layout and font rendering engine as a library (say dynamic link library), an application that requires the new font engine will have to explicitly include and link with the font engine library. This approach is virtually possible on all platforms. For instance, Pango can be compiled as library for Symbian platform—this is what we have done in our research project. This compiled library can later be used by any application on Symbian platform. Similarly, Pango may be compiled as a dynamic or static link library for Android or other platforms.

References

[1]. MobiThinking(2010) Global Mobile Stats: all Latest Quality Research on Mobile Web and Marketing [online], available : http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats [accessed 16 Aug 2010]

[2]. International Telecommunication Unit (2010) ITU sees 5 Million Mobile Subscription Globally in 2010, [online], available: http://www.itu.int/itu-d/newsroom/press_release/2010/06.html [accessed 18 Aug 2010].

[3]. Hussain S. (2003) "Computational Linguistics in Pakistan: Issues and Proposals", in proceedings of European Chapter of the Association for Computational Linguistics (EACL), Workshop in Computational Linguistics for Languages of South Asia, Hungary.

[4]. Wali A. and Hussain S. (2006) "Context Sensitive Shape-Substitution in Nastaliq Writing System: Analysis and Formulation", in proceedings of International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CISSE2006).

[5]. Taylor O. (2001) "Pango: Internationalized Text Handling " [online], available: http://www.lwn.net/2001/features/OLS/pdf/pdf/pango.pdf [accessed 10 Jun 2009].

[6]. PureMobile (2011) What are Smartphones? [online], available: http://www.puremobile.com/smartphones.asp [accessed 10 Dec 2011].

[7]. Barbara B. (2007) "*Designing the Mobile User Experience*", John Wiley and Sons.

[8]. Morris B. (2007) *"The Symbian OS architecture sourcebook: design and evolution of a mobile phone OS*", John Wiley & Sons.

[9]. Forum.Nokia Users (2009) Discussion Board [online], available:http://www.developer.nokia.com/Community/Discussion [accessed 7 Oct 2009].

[10]. Windows Embedded CE 6.0 R3 Development (2011) [online], available: http://www.microsoft.com/windowsembedded/en-us/develop/windows-embedded-ce-6-for-developers.aspx [accessed 10 Dec 2011].

[11]. Paragon Software Group (2010) Language Extender for Windows Mobile Pocket PC [online], available: http://pocket-pc.penreader.com/ [accessed 16 Aug 2010]

[12]. Kblog (2009) Arabic Language in Android [online], available: http://blog.amr-gawish.com/39/arabic-language-in-android/ [accessed 19 Aug 2010]

[13]. Google (2010) Android 2.2 Platform [online], available: http://developer.android.com/sdk/android-2.2.html [accessed 10 Oct 2010].

[14]. Apple (2010) iPhone 4 Technical Specifications [online], available: http://www.apple.com/iphone/specs.html [accessed 20 Aug 2010].

[15]. Spring Design (2011) Mobilize, Don't Miniaturize [online], available: http://www.littlespringsdesign.com/mobilize [accessed 10 Dec 2011].

[16]. Ahmad W. and Hussain S. (2011) "Enabling Complex Asian Scripts on Mobile Devices". *Localization Focus: International Journal of Localization Vol 10, Issue 1.*

3 Symbian Operating System Architecture

The Symbian operating system is composed of a set of layers. Most prominent logical layers of the operating system include *User Interface Framework, Application Services, Operating System Services, Base Services, and Kernel Services and Hardware Interface.* The Figure 3.1 shows stack of Symbian operating system layers. Symbian operating system has headless configuration i.e. minimal user interface features are supported by core operating system. Therefore, third parties have developed user interface layers on top of core Symbian operating system, depicted in Figure 1 as top most layers. These third party libraries include S60—developed by Nokia, UIQ—developed by Sony Ericsson and MOAP—developed by NTT DoCoMo. User applications reside typically on top of S60, UIQ or MOAP. Java Micro Edition libraries exist as separate component in the operating system.

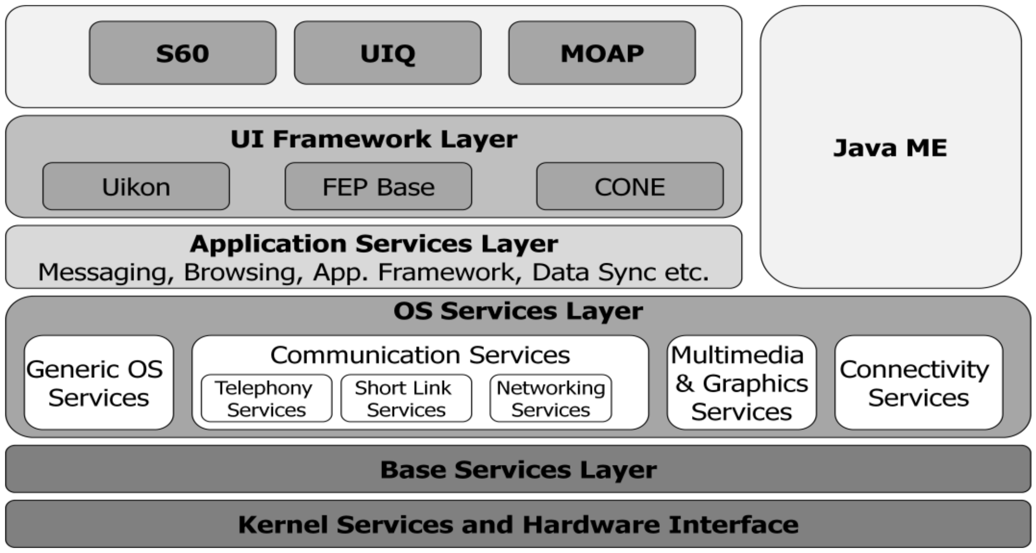


Figure 3.1: Symbian OS Layered Architecture [1]

Symbian OS layers are further divided into blocks and sub-blocks. Each block and sub-block is a collection of individual components. Thus, a layer is the highest level of abstraction, while a component is a lower level of abstraction. Components are physical realization of more logical concepts such as layers and blocks. Components consist of software code including source code, executables, libraries, and documentation.

Each layer abstracts the functionality of layer below it and provides services to layer above it. The level of abstraction increases as we move up from the hardware (at the lowest level) to the user interface (at the highest level).

While layers provide a basic categorization of OS services, blocks and sub-blocks correspond to specific technology domains. Each block consists of a collection of components that provides a set of related services. For example, The OS Services layer contains a Communication Services block which is further decomposed into Telephony, Short Link and Networking Services sub-blocks.

The following sections briefly describe Symbian OS layers and their basic functionality. All Symbian OS releases from v7.0 to v9.3 have the same layer decomposition.

User Interface (UI) Framework Layer

The top most layer of Symbian OS provides libraries and framework for constructing a graphical user interface. This layer includes class hierarchies of user interface controls and concrete widget classes. Third party graphical user interface libraries such as S60 and UIQ have been built by extending the functionality available in user interface framework layer.

User Interface Layers on Symbian OS

User interface framework is the topmost layer of Symbian OS. It provides the framework support on which a production user interface is built. The three currently available custom user interfaces are S60, UIQ and MOAP.

S60 –S60 platform is developed and licensed by Nokia. It supports touch screen, keypad, 5-way navigator, soft keys. Lenovo, LG, Panasonic and Samsung have also shipped S60 enabled phones by licensing S60 from Nokia in the past [1]. In the past, S60 has been shipped in various versions such as 1st, 2nd, 3rd, and 5th editions. After S60 5th edition, Nokia has shipped S60 and Symbian as one open source package under the umbrella of *Symbian Foundation* and various versions of packages have been named Symbian^1, Symbian ^2, and, more recently, Symbian ^3.

UIQ (User Interface Quartz) – UIQ was developed and licensed by UIQ Technology owned by Sony Ericsson. It is most commonly used on Sony Ericsson's P series of smart phones, such as the P990. Other devices shipped with UIQ include Sony Ericsson P990, W950 and W960i. UIQ, however, has not been made part of latest breed of Symbian operating system versions i.e. Symbian ^1 and later editions.

MOAP (Mobile Oriented Application Platform) – MOAP has been developed by FOMA (Freedom of Mobile Access) consortium in Japan. It is a proprietary platform used only by NTTDoCoMo (i.e. not licensed to others).

Application Services Layer

This layer provides application support independent of user interface layer. Application services are broadly classified into three main categories:

1. System level services that provide basic application framework support to all applications,
2. Technology-specific services such as multimedia, telephony, mail, messaging, and browsing,
3. Services that support generic types of applications such as Personal Information Management (PIM) and Alarm Server.

OS Services Layer

This layer acts as a middle-ware between the base services layer at a lower level and the application services layer at an upper level. The services provided by this layer can be divided into four broad categories:

1. Generic operating system services such as task scheduler.
2. Communications services such as telephony, short-link services, and network services.
3. Multimedia services such as windows server, font server, and multimedia framework.
4. Connectivity services such as services for interaction with desktop for file browsing and services for software installation.

Base Services Layer

The base services layer serves as the user side of the two-layer Symbian OS base system. It encapsulates servers, libraries and frameworks that are built on the kernel layer in order to provide upper layers basic operating system services such as file server, basic programming library, persistence model and cryptography library.

Kernel Services and Hardware Interface Layer

The lowest layer of the Symbian operating system contains the operating system kernel and includes components that interface with underlying system hardware. It includes logical and physical device drivers, scheduler and interrupt handler, timers, mutexes etc. In order to port Symbian OS to a new hardware, kernel layer is customized.

Java Micro Edition (Java ME)

Java ME has been built into Symbian operating system as a separate component and it interacts with multiple system layers. It contains MIDP and CLDC libraries, Java Virtual Machine (JVM) and plugins for interaction with native operating system layers.

3.1 Overview of Key Design Patterns

Symbian OS architecture has been structured around a number of design patterns. Some key design patterns are described below:

Microkernel

The Symbian OS kernel is a microkernel; core services that are generally part of the operating system in a monolithic architecture have been moved outside the kernel. All file system services, communication services (including networking) and window services execute at the user side. Therefore, this design places minimum responsibilities on the kernel.

Client-Server Relationship between System Components

All system resources are managed by servers in Symbian OS. The system kernel itself is a server that manages CPU cycles and memory. This pattern is observed throughout the system from lower to higher layers. For example, display is managed by the Windows server, display fonts and bitmaps are managed by the Font and Bitmap server, file services are managed by the File server, data communication hardware is managed by the Serial server, etc. Clients request services from the servers, which own and share resources among multiple clients. Clients and servers reside on same devices but run in their own separate processes in separate memory segments.

Pervasive Asynchronous Methods in Client-Server Communication

In asynchronous processing, a client requests the services of a server by issuing asynchronous requests, i.e. the requesting function does not block after issuing a request. The server informs the client when the service request is complete. Asynchronous services are used throughout Symbian OS, most commonly in communication between client applications and system servers.

Event Based Application Model

User interaction is captured as events. All events are sent to the event queue and event queue is responsible for delivering the event to target application.

Plug-In Framework Model (ECOM)

ECOM enables extension of the Symbian OS. Additional components such as device drivers and font rendering engines can be plugged into the system without recompiling the system code. Plug-ins are independent components that can be integrated into the system framework. The plug-in framework allows plug-ins to register their availability as accessible modules. The framework acts as an enclosing structure for plug-ins i.e. applications request for certain plug-ins and framework loads the requested plug-ins. This provides both extensibility and flexibility in Symbian OS. Flexibility allows loading functionality on-demand and extensibility allows addition of new behavior in the operating system without re-engineering it.

Threads and Processes

Symbian OS supports both multi-threading and multi-processing. Threads are units of execution which the kernel scheduler runs. Processes are collections of one or more threads sharing the same heap memory, but having different stacks. Servers and clients run in their own separate processes in Symbian OS.

3.2 Application Development Concepts Unique to Symbian OS

Symbian OS introduces some development idioms that are unique to Symbian applications. These are discussed in detail in the following sections.

Exception Handling

Exceptions are run-time errors that may be caused by conditions such as out of memory, loss of connectivity, disk full, unavailability of file system when a removable media card is removed or loss of power. These are all likely occurrences in a resource-constrained environment such as of mobile phones. In Symbian, Leave-Trap exception handing mechanism has remained most dominant and is still being used by many applications. In newer versions, Symbian operating system supports standard C/C++ exception handling mechanism (i.e. try-catch mechanism).

In traditional Symbian OS, exceptions are characterized as 'leaves'. A leave is a call to the function User::Leave(), and it causes program execution to return immediately to the trap harness within which the function was executed. By convention, all leaving functions are superseded by the letter 'L'. When a function has such a suffix, it can return a special error state that will propagate the need for return. This error state is captured using a trap harness. For example, a leaving function can be declared as follows.

```
void AllocateMemoryL() {

        //Allocate some memory here

}

To catch the leave, a trap harness can be setup as using the TRAP or TRAPD.

TRAPD(error, AllocateMemoryL());

if (error!=KErrNone){//Do some error coding}
```

Any functions called by allocateMemoryL() are also executed within the trap harness, as are any functions called by them, and so on. Therefore, a leave occurring in any function nested within allocateMemoryL() will return to this trap harness.

Cleanup Stack

Whenever a leave occurs, effective cleanup of resources is very important. Symbian features a 'cleanup stack' to store pointers to heap-allocated objects that need to be freed when a leave occurs. For example consider the following code.

```
void SomeFunctionL(){

CSomeClass *someObject = new CSomeClass ();

LeavingFunctionL();

delete someObject;}
```

If the LeavingFunctionL()leaves, the next statement will not be executed and someObject will be left allocated on the heap i.e. the someObject will not be cleaned up from the heap. The cleanup stack helps in avoiding this problem. The cleanup stack is used to store the pointer to the newly allocated object so that it can be destroyed later in case a leave occurs. Thus, the correct way of writing the above code is as follows.

```
void SomeFunctionL()

{

        CSomeClass *someObject = new CSomeClass ();

        CleanupStack::PushL(someObject);

        LeavingFunctionL();

        CleanupStack::PopAndDestroy(someObject);

}
```

This way the objects whose memory needs to be free in case a Leave (exception) occurs, are pushed onto a de-allocation (Cleanup) stack. If the execution is performed normally the objects are popped from the stack and destroyed. If a leave occurs, the system TRAPD macro pops and destroys everything on the cleanup stack that was pushed to it before the beginning of the trap. Thus, cleanup stack ensures that no memory leaks occur in case of a leave.

Two-Phase Construction

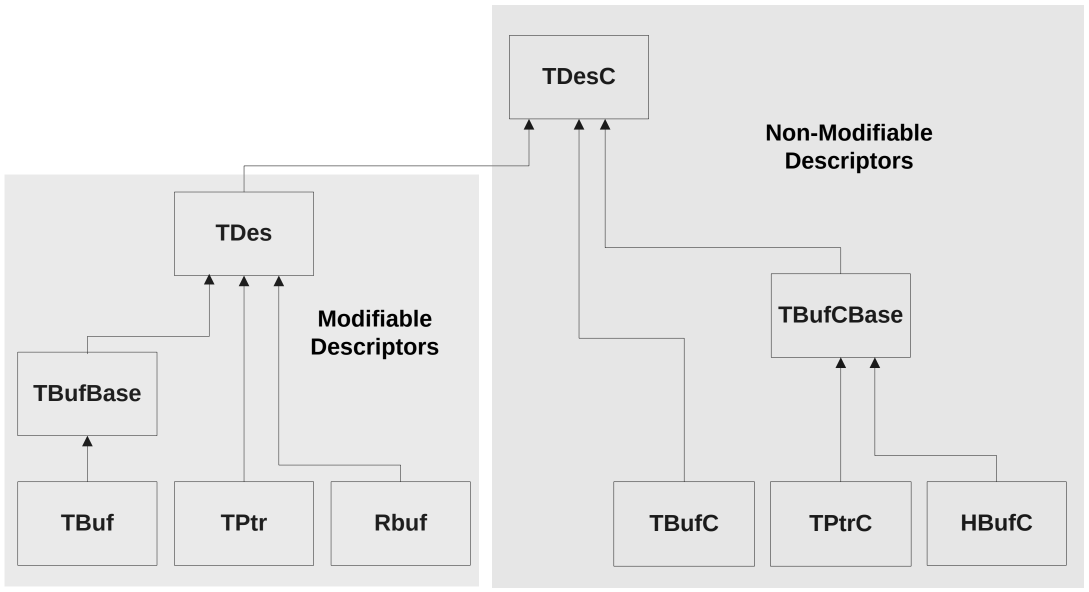
Two-phase construction guarantees that C++ construction of an object will always succeed. This is achieved by moving all statements that may raise an exception (i.e. where a Leave might occur) out of the normal C++ class constructor to a secondary constructor. A compound class constructor may leave while allocating memory for its contained objects. Thus the constructor may not be executed properly and memory allocated to the object may be orphaned. Therefore, all complex objects in Symbian are constructed in two phases as described below:

- All normal C++ constructors are made private. Minimal non-leaving code is placed inside the normal constructors.
    - All initialization code that might leave is located in a separate function ConstructL(), referred to as a second-phase constructor. The second phase constructor is only called after the object being initialized has been pushed onto the cleanup stack. Both these phases of construction can be combined in a single function called NewL or NewLC. See the example code below for an overview of the two-phase construction [5].

```
// Phase #1

CMyClass::CMyClass()

        {

        }

// Phase #2

void CMyClass::ConstructL()

        {

        // Member data initialization.

        }

// Put both phases together in one function...

CMyClass * CMyClass::NewL()

        {

        CMyClass * self = new (ELeave) CMyClass();

        CleanupStack::PushL(self);

        self->ConstructL();

        CleanupStack::Pop(self);

        return self;

        }

CMyClass * CMyClass::NewLC()

        {

        CMyClass * self = new (ELeave) CMyClass();

        CleanupStack::PushL(self);

        self->ConstructL();

        return self;

        }
```

Descriptors

String and binary data manipulation is done using Descriptors. Descriptors are the classes that encapsulate various types of data and allow functions to manipulate them. For instance, TPtr8 is an 8-bit pointer descriptor and TBuf is a stack-based buffer descriptor. Descriptors (or 'safe strings') are used to manipulate text and binary data in Symbian OS. These are known as 'descriptors' because they are self describing. Each descriptor object consists of two parts: the length of the data buffer and the data itself. It also contains the type of the data it holds which identifies the underlying memory allocation layout and associated operations that can be performed on it [2]. Since descriptors encode their type and length in bytes in their headers, therefore they protect against buffer overflows and out-of-memory accesses. Hence the term 'safe' strings [1]. There are a number of descriptor classes that can hold either 8-bit (Narrow descriptors) or 16-bit (Wide or Unicode descriptors) characters. The character width of descriptor classes can be identified from their names. If the class name ends in 8 (for example, TPtr8) it has narrow (8-bit) characters, while a descriptor class name ending with 16 (for example, TPtr16) refers to 16-bit character strings including Unicode text.



ggFigure 3.2: Symbian Descriptor Class Hierarchy [3]

All descriptor classess in Symbian OS inherit from the base class TDesC as shown in Figure 3.2. These can be further classified into the following types.

1. Generic Descriptors (Non-modifiable and Modifiable) – TDesC and TDes
2. Stack-Based Buffer Descriptors (Non-modifiable and Modifiable) – TBufC and TBuf
3. Pointer Descriptors (Non-modifiable and Modifiable) – TPtrC and TPtr
4. Heap-Based Buffer Descriptors (Non-modifiable and Modifiable) – HbufC and RBuf

The content of un-modifiable or constant (Descriptor with suffix ‘C’ in its class name) descriptors cannot be changed, although it can be replaced, whereas modifiable descriptors can be altered, up to the maximum size specified when descriptor was constructed. An important distinction between buffer and pointer descriptor classes is that the buffer descriptors actually contain data, whereas pointer descriptors point to data stored elsewhere. Moreover, a distinction between stack-based and heap-based buffer descriptors is that the stack-based descriptors are relatively transient and should be used for small strings because they are created directly on the stack and heap-based descriptors, on the other hand, are intended to have longer life and are likely to be shared throughout the run-time life of a program.

Active Objects

Almost all system services in Symbian are provided through servers running in their own processes. Servers enable access to supported services using Asynchronous calls. Symbian OS provides an active object framework to manage asynchronous service requests. The active object framework consists of Active Objects and Active Scheduler. These are used in together for event-driven multitasking.

An asynchronous function runs the service completion request in the background, returning control to the caller immediately after the request is made. When the requested service is complete, the caller is sent a signal, known as ‘event’. Events are managed by an event handler [4]. An active object encapsulates a task, requests an asynchronous service from a server and handles the completion event when the active scheduler calls it. The active scheduler maintains a list of all active objects which have made a request for an asynchronous service. It also receives notifications of events when the service requests are completed.

All asynchronous requesting objects are implemented as Active Objects. An ‘Active Object’ performs following:

- Places requests for an asynchronous service,
- Handles the service completion event.
- May ask to cancel a request
- Is registered with active scheduler

When an asynchronous service completes, it generates events to notify Active Scheduler. The Active Scheduler performs following:

- Detects service completion events
- Determines associated Active Object
- Calls the Active Object to handle the event completion.

The class implementing active objects must derive from CActive, an abstract class containing two pure virtual functions RunL() and DoCancel(). The RunL() is the event handling method when a request completes and it cannot be preempted. The DoCancel() method is used to terminate an outstanding service request and must be implemented by the active object.

Figure 3.3 shows the sequence of events that occur when an active objects places a request for an asynchronous service provider.

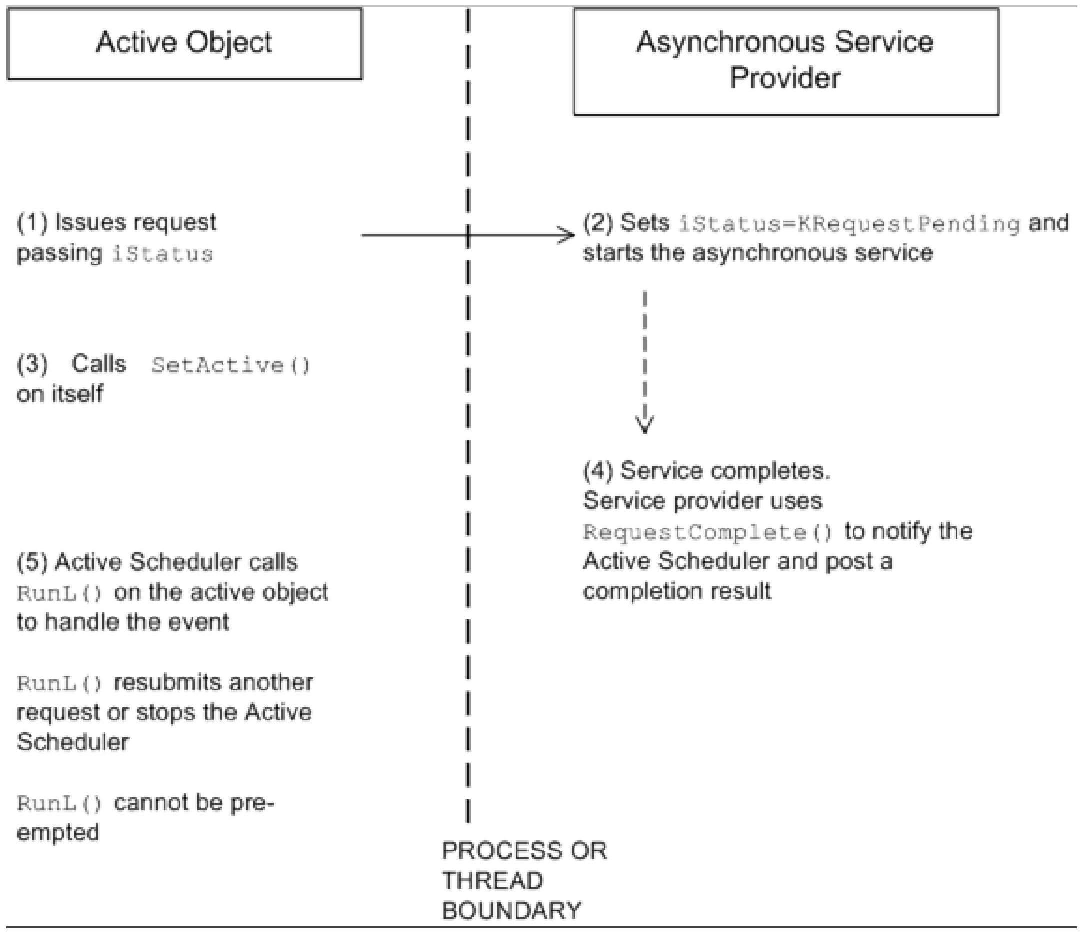

Active Object
Asynchronous Service Provider
(1) Issues request passing iStatus
(2) Sets iStatus=KRequestPending and starts the asynchronous service
(3) Calls SetActive() on itself
(4) Service completes. Service provider uses RequestComplete() to notify the Active Scheduler and post a completion result
(5) Active Scheduler calls RunL() on the active object to handle the event
RunL() resubmits another request or stops the Active Scheduler
RunL() cannot be pre-empted
PROCESS OR THREAD BOUNDARY

Figure 3.3: Sequence of Actions Performed When an Active Object Submits Service Request [4]

References

[1]. Morris, B. (2007) "*The Symbian OS Architecture Sourcebook: Design and Evolution of a Mobile Phone OS*", John Wiley & Sons.

[2]. Symbian (2010) Descriptors (Fundamentals of Symbian C++) [online], available: http://developer.symbian.org/wiki/index.php/Descriptors_%28Fundamentals_of_Symbian_C%2B%2B%29 [accessed 15 Dec 2010].

[3]. Symbian (2010) Descriptors [online] available: http://developer.symbian.org/wiki/index.php/File:Descriptors.png_[accessed 15 Dec 2010].

[4]. Symbian (2010) Active Objects (Fundamentals of Symbian C++) [online], available: http://developer.symbian.org/wiki/index.php/Active_Objects_%28Fundamentals_of_Symbian_C%2B%2B%29 [accessed 15 Dec 2010].

[5]. Nokia Forum (2010) Two Phase Construction [online], available: http://wiki.forum.nokia.com/index.php/Two-phase_construction [accessed 15 Dec 2010].

4 Setting up the Development Environment

The first step in the development of Symbian applications is to setup the development environment. This includes installing and setting up the SDK as well an IDE for project creation, compilation and debugging, etc. The SDK can be downloaded from www.forum.nokia.com. A number of SDK versions are available depending upon which version of Symbian OS and s^0 edition are required by the application. For example Nokia E51 comes with Symbian OS v9.2 and S60 3rd Edition, Feature Pack 1. Feature packs are additional libraries that are added to the SDK as new features appear on phones.

In theory, a text editor and an SDK are sufficient to develop software for Symbian OS. However, in order to quickly produce effective code, an Integrated Development Environment (IDE) is vital. An IDE performs a number of useful functions such as color-coding the source code, grouping together files, compiling code and interpreting error messages from the compiler. The more advanced IDEs provide debugging tools on the emulator or the target device. An IDE may also provide access to additional tools such as project wizards that help in the creation of new projects. Three IDEs are available for this purpose.

i. Carbide.c++, based on the open source IDE Eclipse
ii. Microsoft Visual Studio
iii. CodeWarrior

Carbide.c++ is provided by Nokia Installation of tools.The system requirements for SDK are as follows:

Before installing the S60 SDK, some additional tools need to be installed. These include Active Perl version 5.6.1 or newer and Java Runtime Environment (JRE). The order of installations should be as follows.

i. Active Perl version 5.6.1 or newer.
ii. JRE version 1.6 or newer.
iii. S60 SDK
iv. Carbide.c++ IDE version 2.3

4.1 Project Creation

Once the SDK and IDE are successfully installed, Carbide.c++ can be started from the Windows Start menu → All Programs → Nokia → Carbide.c++ → Carbide.c++ as shown in Figure 4.1. It prompts for configuring the workspace folder path. Workspace is the working directory where Carbide.c++ stores all projects. It is important to note here that the workspace directory must be created on the same drive where the Symbian SDK is installed. Also, the path must not contain any spaces or non-alphanumeric characters. Figure 4.2 shows a correct workspace path if the SDK has been installed on C:\ drive.

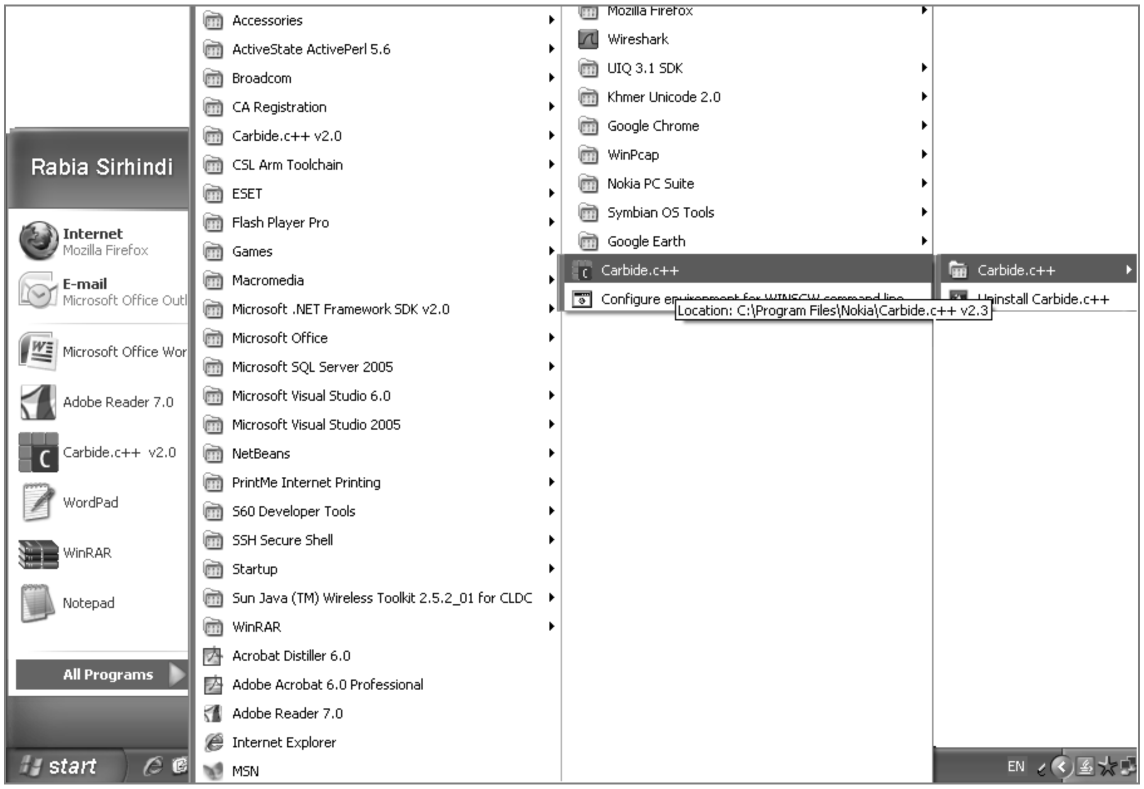
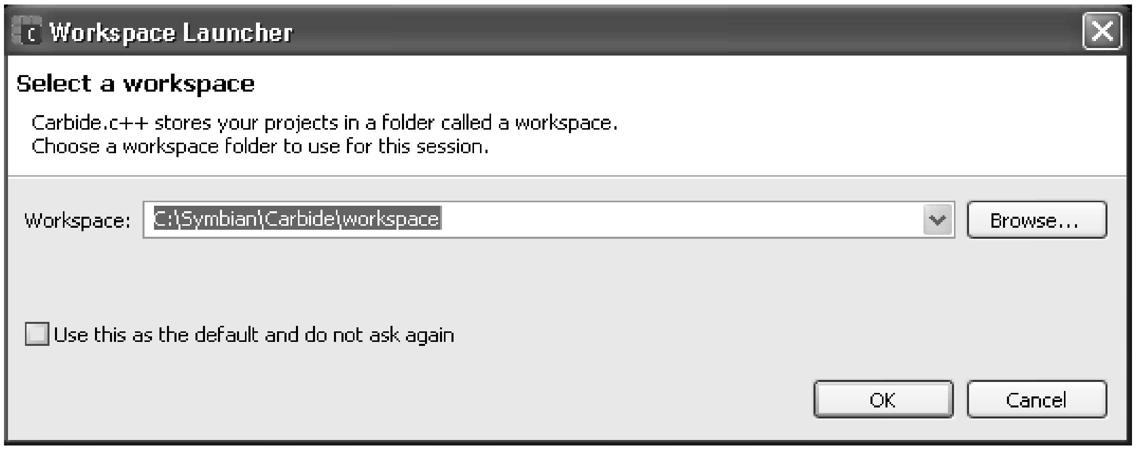


Figure 4.1: Starting Carbide.c++


Workspace Launcher

Select a workspace

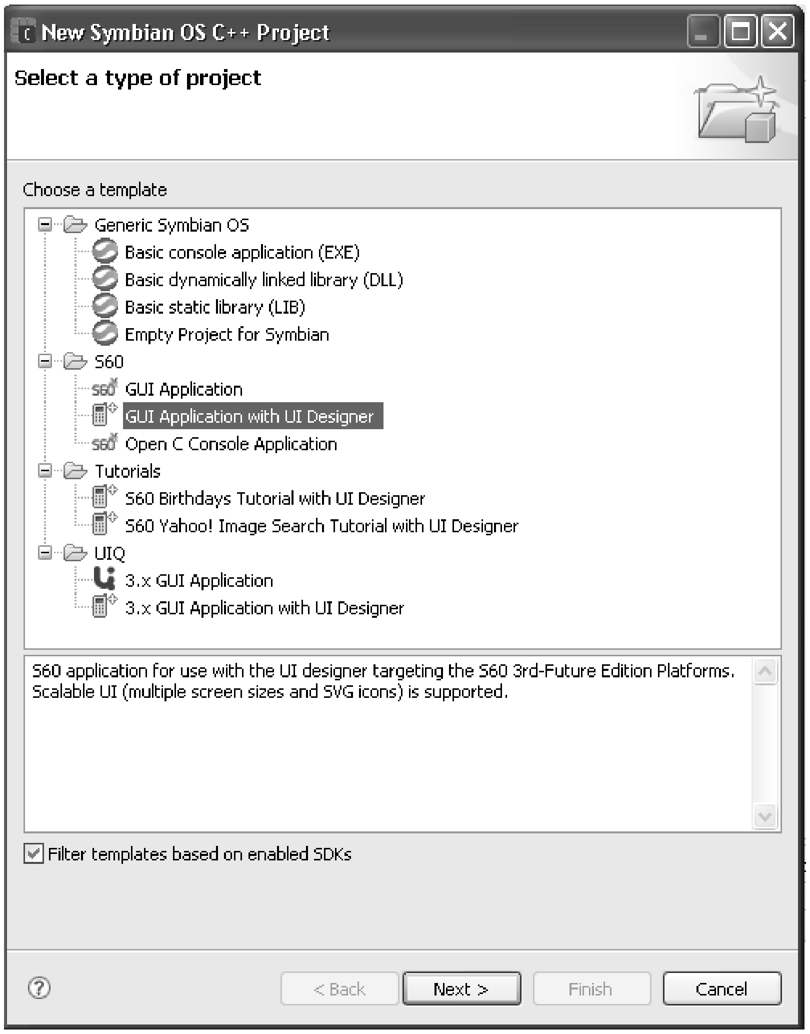
Carbide.c++ stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\Symbian\Carbide\workspace

Use this as the default and do not ask again

Figure 4.2: Setting Workspace Directory

The next step is to create a new Symbian OS project to start development. The Create Project Wizard can be launched from File → New → Symbian OS C++ Project. Figure 4.3 shows the window that is displayed.


New Symbian OS C++ Project

Select a type of project

Choose a template

- Generic Symbian OS
  - Basic console application (EXE)
  - Basic dynamically linked library (DLL)
  - Basic static library (LIB)
  - Empty Project for Symbian
- S60
  - GUI Application
  - GUI Application with UI Designer
  - Open C Console Application
- Tutorials
  - S60 Birthdays Tutorial with UI Designer
  - S60 Yahoo! Image Search Tutorial with UI Designer
- UIQ
  - 3.x GUI Application
  - 3.x GUI Application with UI Designer

S60 application for use with the UI designer targeting the S60 3rd-Future Edition Platforms. Scalable UI (multiple screen sizes and SVG icons) is supported.

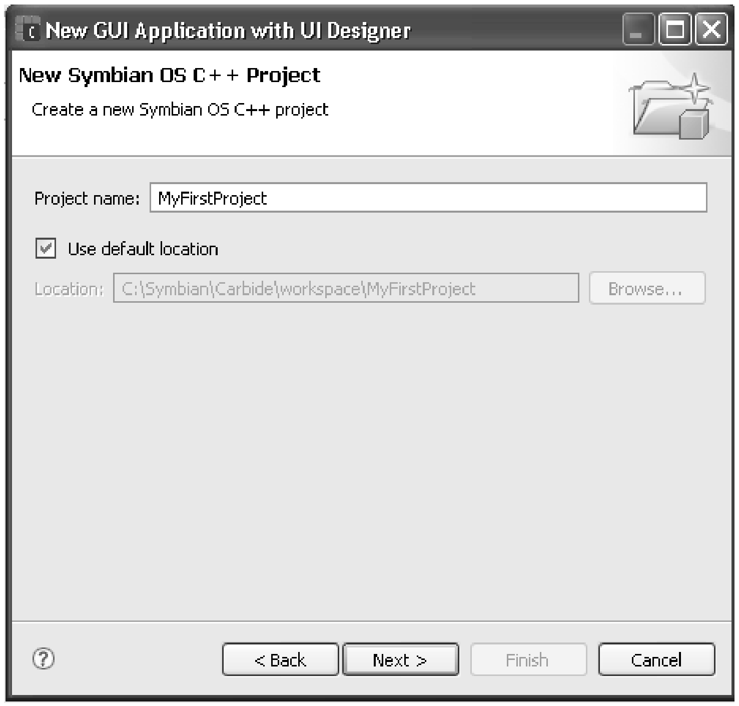
[✓] Filter templates based on enabled SDKs

< Back | Next > | Finish | Cancel

Figure 4.3: Choosing New Symbian OS Project

Basic Console Application (EXE) creates an application without a GUI, with only command line interface for interaction with user. Figure 4.3 shows two options to create GUI based applications. A traditional Symbian OS application can be created using S60 GUI Application from the above menu. Alternatively, Carbide.c++ provides a UI Designer tool, allowing views to be created using drag-and-drop components. This can be activated if S60 GUI Application with UI Designer option is selected from the above list.

The next page of the wizard brings up some basic project specifications such as name and location as shown in Figure 4.4. By default all new projects are saved in Carbide.c++ workspace directory. The project name must not contain any spaces or special characters.



New Symbian OS C++ Project

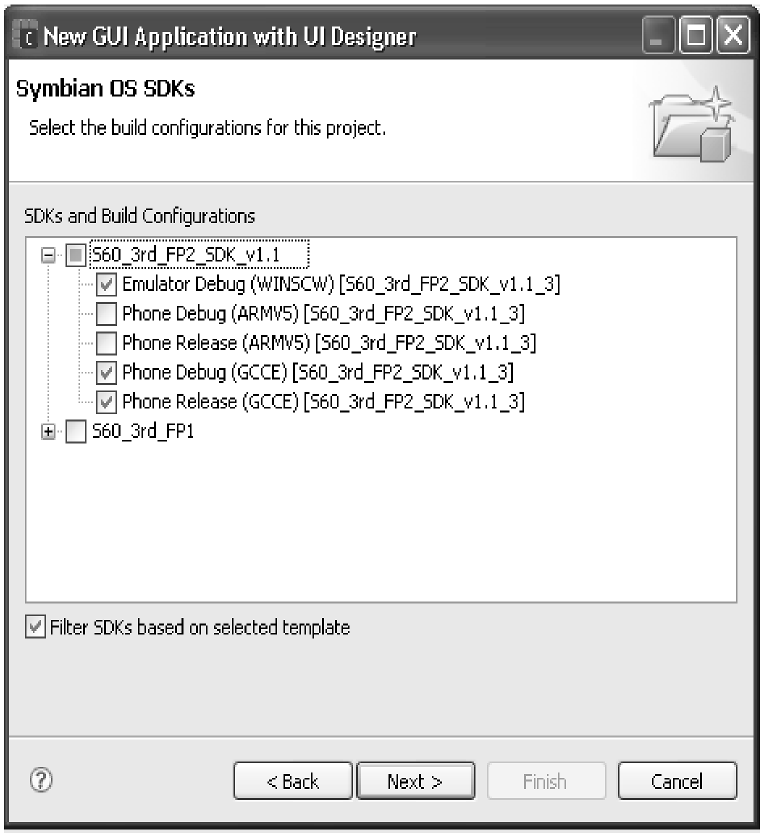
Create a new Symbian OS C++ project

Project name: MyFirstProject

☑ Use default location

Location: C:\Symbian\Carbide\workspace\MyFirstProject

Figure 4.4: Project Details



Symbian OS SDKs

Select the build configurations for this project.

SDKs and Build Configurations

- S60_3rd_FP2_SDK_v1.1
  - ☑ Emulator Debug (WINSCW) [S60_3rd_FP2_SDK_v1.1_3]
  - ☐ Phone Debug (ARMV5) [S60_3rd_FP2_SDK_v1.1_3]
  - ☐ Phone Release (ARMV5) [S60_3rd_FP2_SDK_v1.1_3]
  - ☑ Phone Debug (GCCE) [S60_3rd_FP2_SDK_v1.1_3]
  - ☑ Phone Release (GCCE) [S60_3rd_FP2_SDK_v1.1_3]
- ☐ S60_3rd_FP1

☑ Filter SDKs based on selected template

Figure 4.5: Selecting SDK and Build Configurations

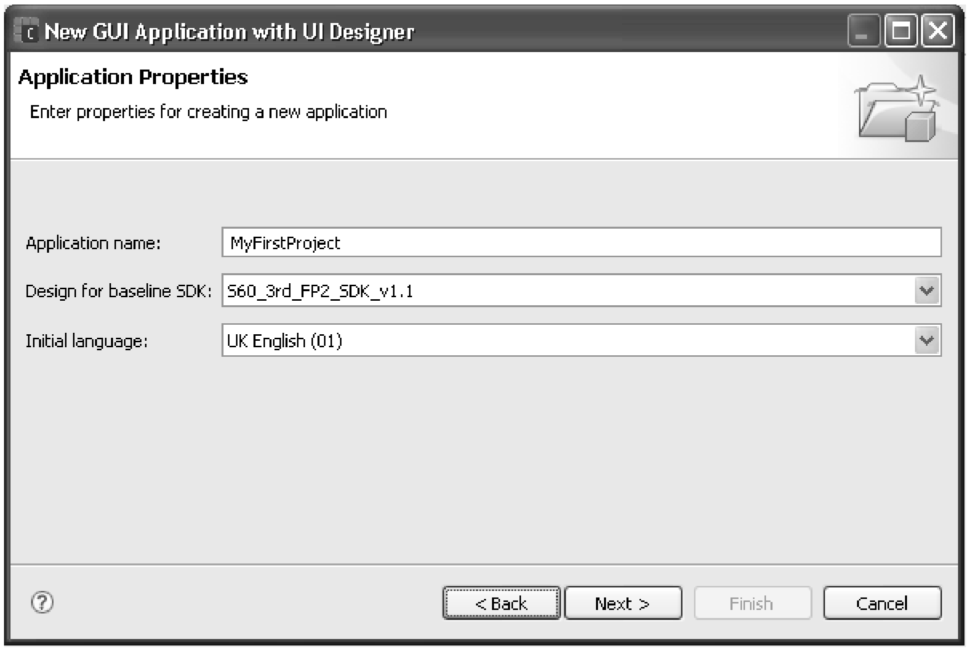
The next page of the wizard prompts for the choice of SDK (Figure 4.5). For example in the figure below, there are two SDKs installed on the system, S60 SDK FP1 and S60 SDK FP2 v1.1.This phase in project creation also prompts for the choice of build configurations. There are number of build configurations and by default all are selected. These are mainly classified as building the project for Emulator and building it for the target device. Both configurations are discussed in details in subsequent sections.

Emulator Debug (WINSCW) – builds binaries for the Windows-hosted emulator.

Phone Debug/Release (GCCE) – builds binaries for the phone using the GCCE compiler that is installed with the SDK.

Phone Debug/Release (ARMV5) – builds binaries for the phone using the ARM Real View Compiler (RVCT). RVCT produces code that is optimized than the current versions of GCCE supported for Symbian C++ development, but must be separately licensed from ARM. RVCT is primarily used by phone manufacturers to build binaries for device ROM.

More specifically, to build the code and generate .sis file for target mobile device, the option Phone Release (GCCE) must be checked.


New GUI Application with UI Designer

Application Properties

Enter properties for creating a new application

Application name: MyFirstProject

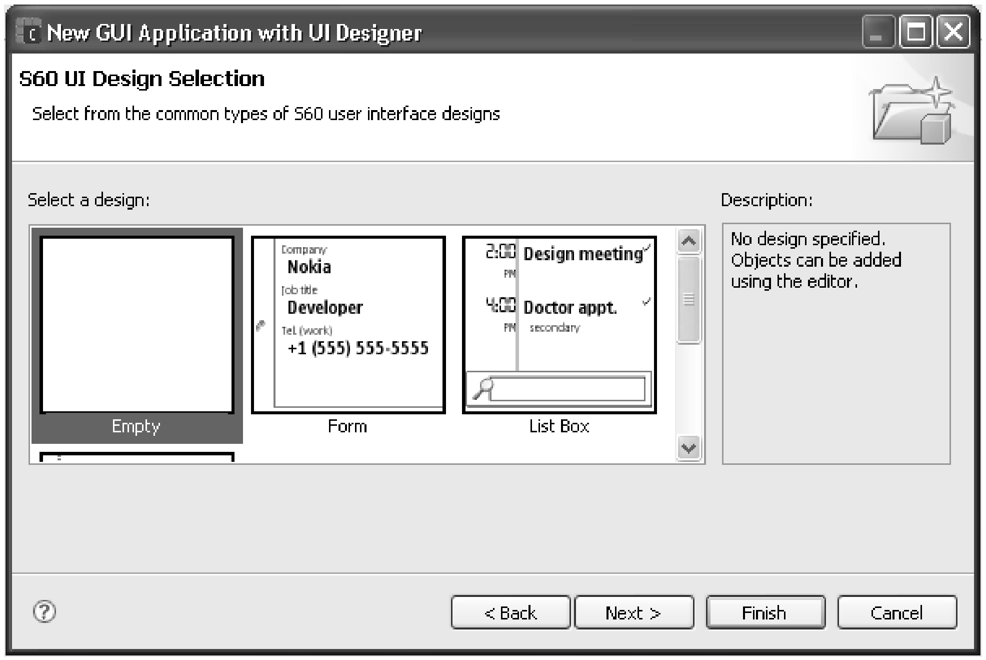
Design for baseline SDK: S60_3rd_FP2_SDK_v1.1

Initial language: UK English (01)

Figure 4.6: Setting Project Properties

The next page shows the application properties as shown in Figure 4.6. Usually default values are kept for these.

The next step allows selecting a UI design. Here Empty UI variant is selected so that components can later be added to the design. Figure 4.7 shows the options for design selection.


New GUI Application with UI Designer

S60 UI Design Selection

Select from the common types of S60 user interface designs

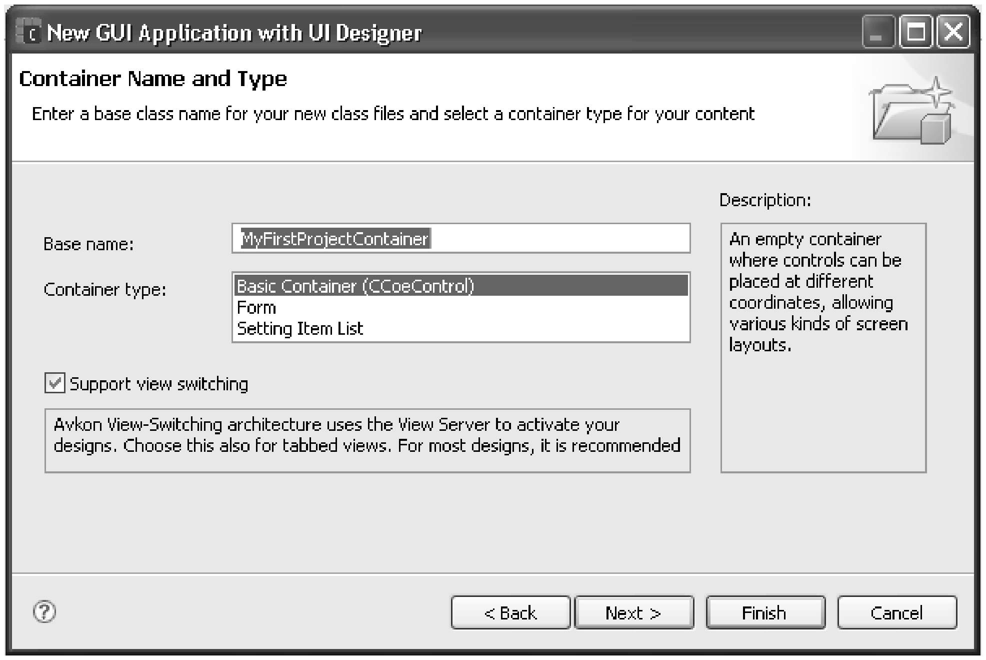
Select a design:

Description:

No design specified. Objects can be added using the editor.

Figure 4.7: Selecting UI Design

The next step is to choose the base container's class name as shown in Figure 4.8. Initially, the project has only one view; however, view switching is enabled to allow additional views in the application.


New GUI Application with UI Designer

Container Name and Type

Enter a base class name for your new class files and select a container type for your content

Base name: MyFirstProjectContainer

Container type: Basic Container (CCoeControl) / Form / Setting Item List

☑ Support view switching

Avkon View-Switching architecture uses the View Server to activate your designs. Choose this also for tabbed views. For most designs, it is recommended

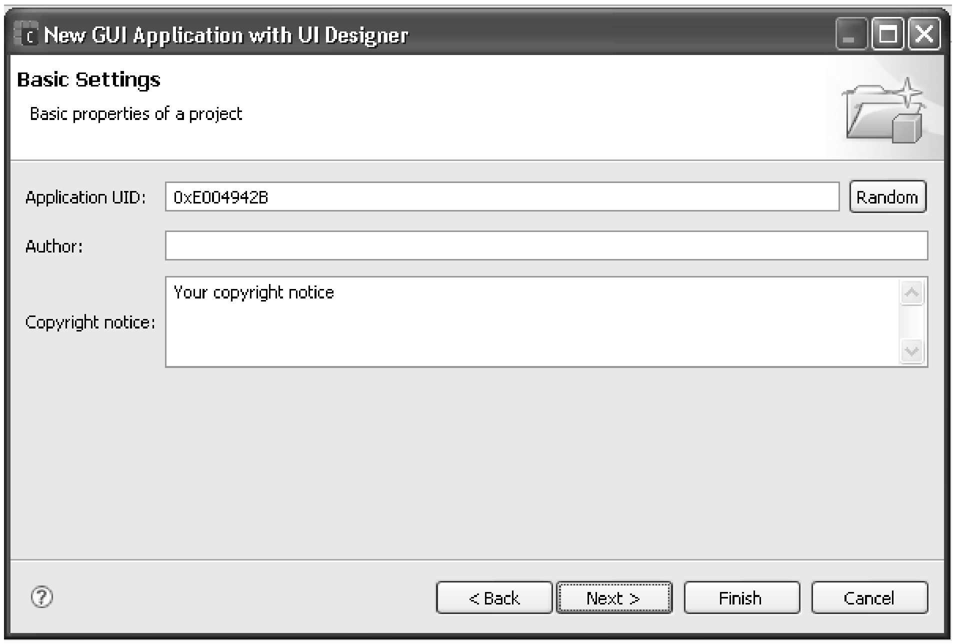
Description:

An empty container where controls can be placed at different coordinates, allowing various kinds of screen layouts.

Figure 4.8: Setting UI Container Details

Subsequent steps include setting the application's Unique Identifier (UID) and project sub-directories as shown in Figure 4.9. The UID (a number in the 32 bit range 0x00000000 to 0xFFFFFFFF) defines the private area in the file system in which the application can store its data. Among other things the UID can also be used to programmatically identify and/or start the application [1].

Carbide.c++ generates a random UID value for you starting with '0xE', which indicates the range of UIDs reserved for internal development and testing. If the application has to be publicly released it has to be assigned a unique UID allocated by Symbian Signed.


New GUI Application with UI Designer

Basic Settings

Basic properties of a project

Application UID: 0xE004942B

Author:

Copyright notice: Your copyright notice

Figure 4.9: Setting Project UID

The Figure 4.10 shows project directory structure.

/inc – list of project header files (e.g., .h files)

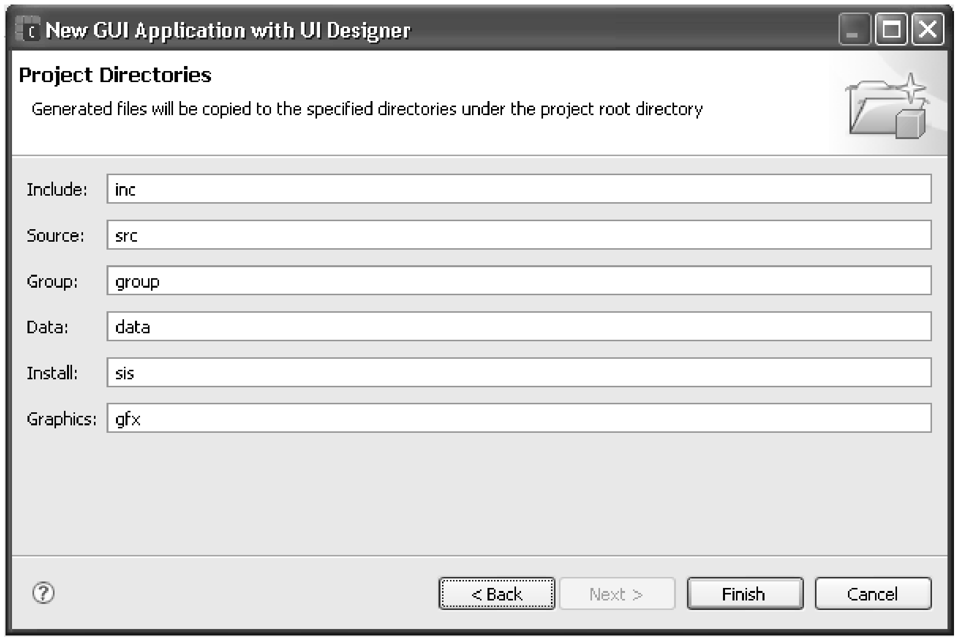
/src – list of project source files (e.g., .cpp files)

/group – list of project definition and make files

/data – list of resource and localization files

/sis – list of package and installation files needed to run application on device

/gfx – list of default bitmap icons used by the project



Project Directories

Generated files will be copied to the specified directories under the project root directory

Include: inc
Source: src
Group: group
Data: data
Install: sis
Graphics: gfx

Figure 4.10: Setting Project Directories

Figure 4.11 shows the final UI screen when a project has been created.

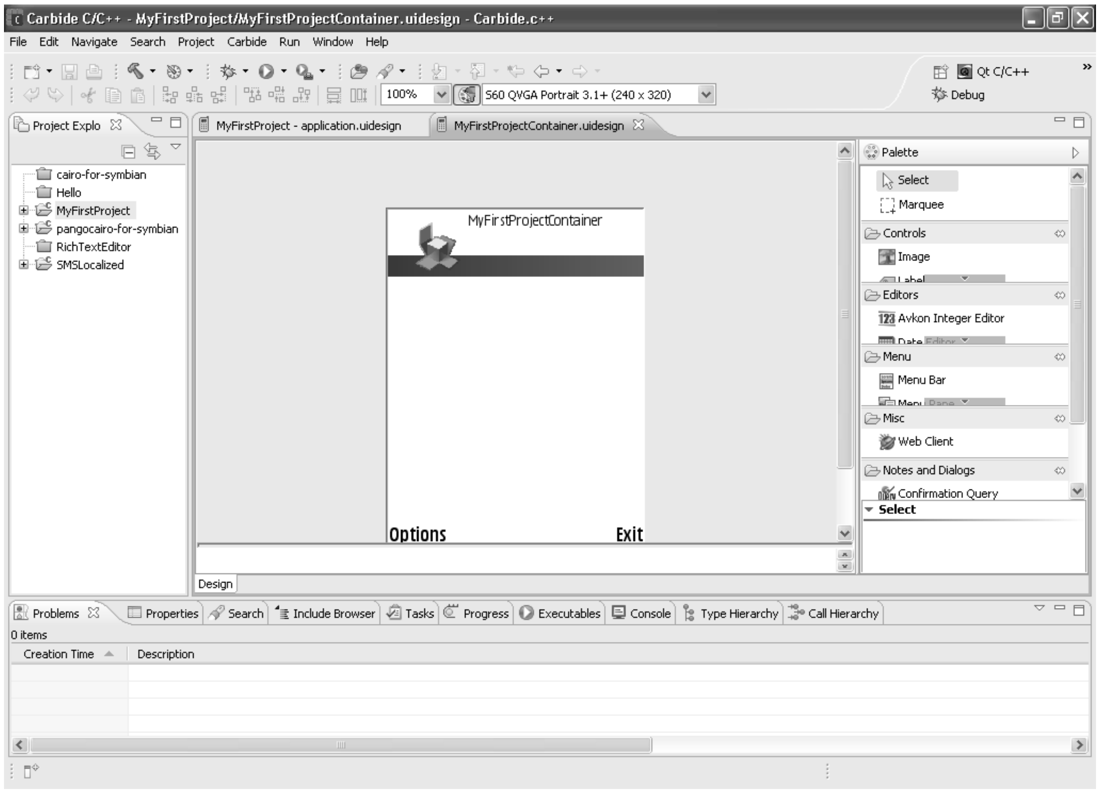


Figure 4.11: Project View

4.2 S60 SDK Emulator

A mobile phone emulator is a Windows application that simulates the mobile phone software and hardware on a personal computer as shown in the Figure 4.12. Use of the emulator saves time in the early stages of development, since the development IDE can be used to debug the code easily and to resolve most initial coding and design problems. For example, if a panic (run-time error) occurs in the code, the debugger can provide comprehensive information to diagnose the error condition that caused it. Using the emulator also eliminates the need for creating an installation package, signing and installing it to phone, which would otherwise be time consuming in the early phases of development.

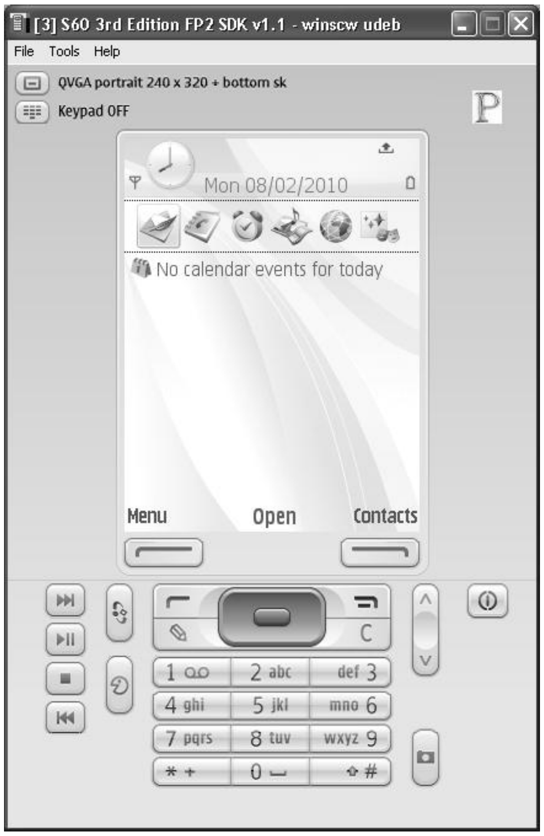

[3] S60 3rd Edition FP2 SDK v1.1 - winscw udeb
File Tools Help
QVGA portrait 240 x 320 + bottom sk
Keypad OFF
Mon 08/02/2010
No calendar events for today
Menu
Open
Contacts

Figure 4.12: S60 SDK Emulator

The emulator software is installed with the SDK and can be launched in one of the following ways.

1. Launch the executable epoc.exe from %EPOCROOT%\epoc32\release\winscw\udeb. Here %EPOCROOT% is the directory where the SDK is installed.

2. Select All Programs from the Start menu, and under S60 Developer Tools, 3rd Edition FP2 SDK, and select Emulator as shown in Figure 4.13.

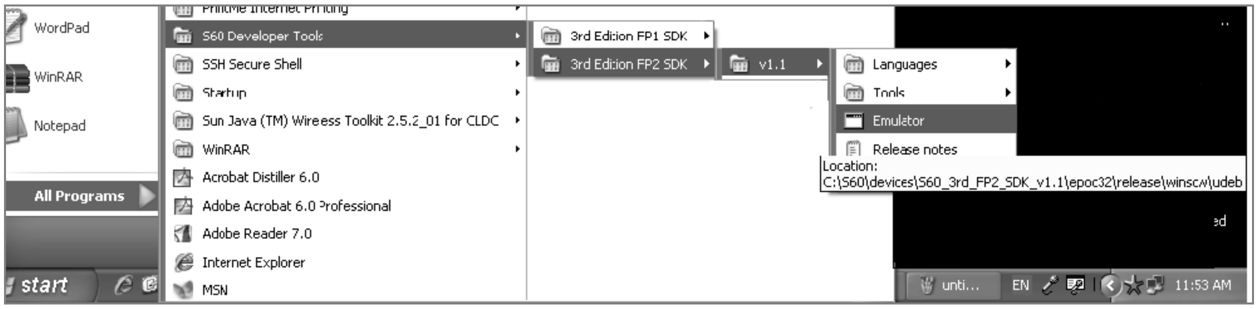


Figure 4.13: Starting the Emulator for Start Menu

Applications can be launched using the Application Launcher in the emulator. As its name indicates, the application launcher enables you to start installed applications.

4.2.1 Debugging and Testing on Emulator

The emulator maps features of the target device onto features of the PC environment. An emulator configuration directory and startup directory completes the list of directories required by the emulator.

\epoc32\data\ is the emulator configuration directory. It contains the initialization parameters for the emulator (epoc.ini), the bitmap used as the fascia surround for the screen (epoc.bmp), and variants for screens of different sizes.

\epoc32\release\winscw\udeb\ is the emulator startup directory. It contains the Windows emulator (epoc.exe) and the entire shared library DLLs.

\epoc32\release\winscw\udeb\z\ is the emulated Z: drive. It contains everything that the EPOC Z: drive should contain, except shared library DLLs, which are in the parent directory.

\epoc32\winscw\c\ is the emulated C: drive. It contains data and files. It does not contain compiled C++ programs – those should all be on Z. In the emulator, all compiled applications become part of the pseudo-ROM that is the emulated Z: drive.

In the simplest configurations, the project has to be built for the Emulator. You can do this by clicking the Manage Configurations icon in the toolbar or by selecting Menu → Project → Build Configurations → Set Active and select Emulator Debug as shown in the Figure 4.14.

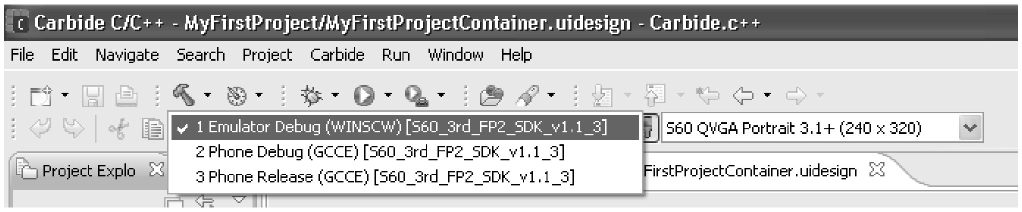
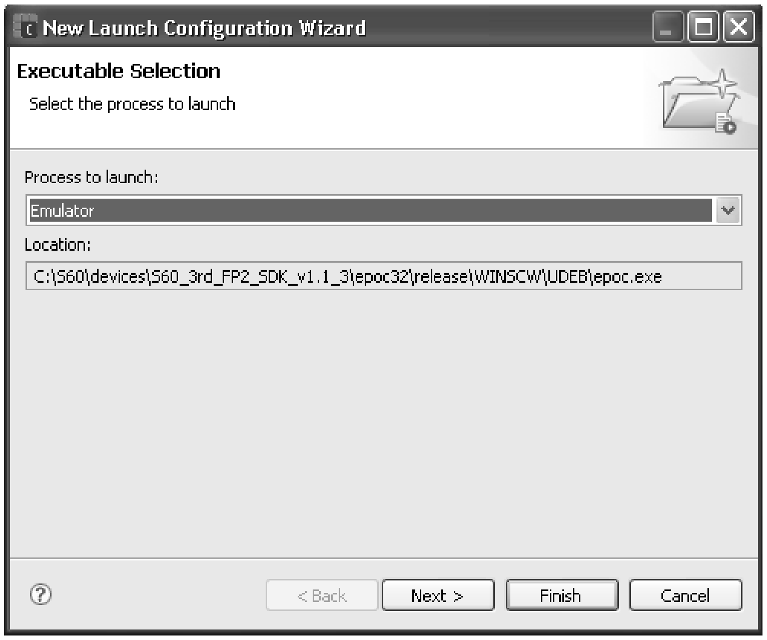

Carbide C/C++ - MyFirstProject/MyFirstProjectContainer.uidesign - Carbide.c++
File Edit Navigate Search Project Carbide Run Window Help
✓ 1 Emulator Debug (WINSCW) [S60_3rd_FP2_SDK_v1.1_3]
2 Phone Debug (GCCE) [S60_3rd_FP2_SDK_v1.1_3]
3 Phone Release (GCCE) [S60_3rd_FP2_SDK_v1.1_3]
S60 QVGA Portrait 3.1+ (240 x 320)
Project Explo
FirstProjectContainer.uidesign

Figure 4.14: Building for Emulator

To build the current configuration the Build icon in the toolbar is used. If the application builds successfully, then the Run button (Ctrl + F11) can be used to launch the emulator. Upon the first launch of the project, Carbide.c++ will prompt for the executable that has to be launched. This is because no active run configuration is currently set.

If the project executable (<*ProjectName*>.exe) is selected from the menu as shown in Figure 4.15, then the emulator is launched and the application starts automatically. The emulator will close once you exit your application. If, however, Emulator option is selected, then the emulator (epoc.exe) will be launched and you will have to navigate to the application and start it by clicking on the icon. This is shown in Figures 4.15 and 4.16.


New Launch Configuration Wizard

Executable Selection

Select the process to launch

Process to launch: Emulator

Location: C:\S60\devices\S60_3rd_FP2_SDK_v1.1_3\epoc32\release\WINSCW\UDEB\epoc.exe

Figure 4.15: Launching Configuration for Emulator

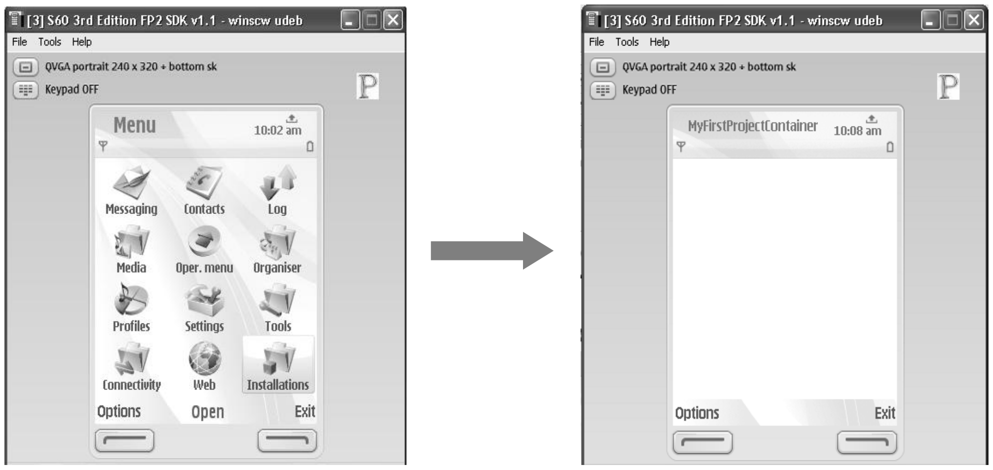


Figure 4.16: Running Project on Emulator

4.3 Packaging a .SIS File and installation on device

Symbian applications are packaged for installation in (.sis) files based on a specification in a package (.pkg) file. Many manufacturers further require that only digitally signed .sis files may be installed - these files use file extension ".sisx".

To be able to run the application on an actual device, it has first to be compiled for the device. This can be done by selecting Phone Release (GCCE) option from the build configurations as shown in Figure 4.17.

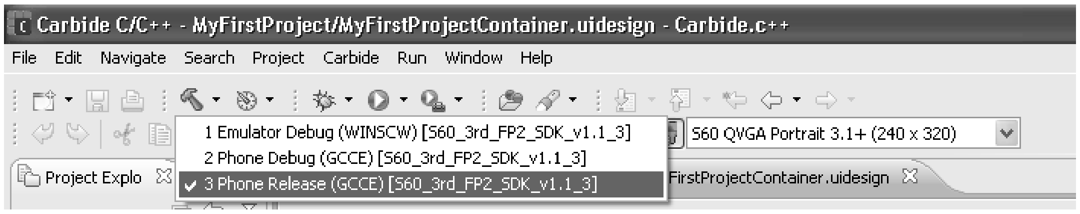

Carbide C/C++ - MyFirstProject/MyFirstProjectContainer.uidesign - Carbide.c++
File Edit Navigate Search Project Carbide Run Window Help
1 Emulator Debug (WINSCW) [S60_3rd_FP2_SDK_v1.1_3]
2 Phone Debug (GCCE) [S60_3rd_FP2_SDK_v1.1_3]
✓ 3 Phone Release (GCCE) [S60_3rd_FP2_SDK_v1.1_3]
S60 QVGA Portrait 3.1+ (240 x 320)
Project Explo
FirstProjectContainer.uidesign

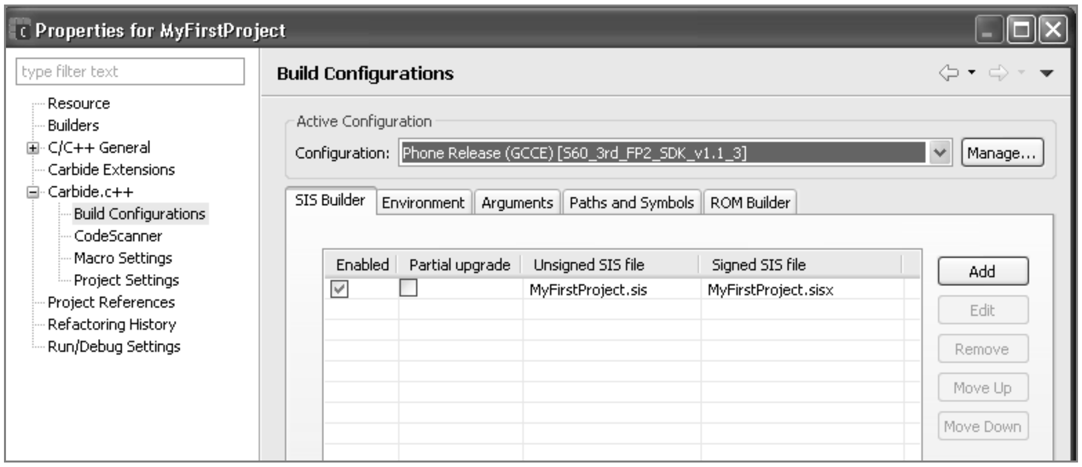
Figure 4.17: Building for Target Device

SIS file creation configuration can be done through the Project → Properties menu as shown in Figure 4.18. In the Build Configurations window, activate Phone Release (GCCE) build. If there is no profile in the SIS Builder tab, then a new entry has to be manually added to the SIS Builder tab. This will open a SIS File Properties dialog for the new .sis file as shown in Figure 4.19. A few parameters have to be specified here.

- PKG File — name of the PKG file to build. Click Browse to locate or else type in the file path and name.
- Output File Name — unsigned file name that is generated from the PKG file selected
- Content Search Location — root location where PKG files are specified to search
- Generate partial upgrade when appropriate — enable to create package update files that only contain files changed since the last build

There are also a number of signing options for the installation file.

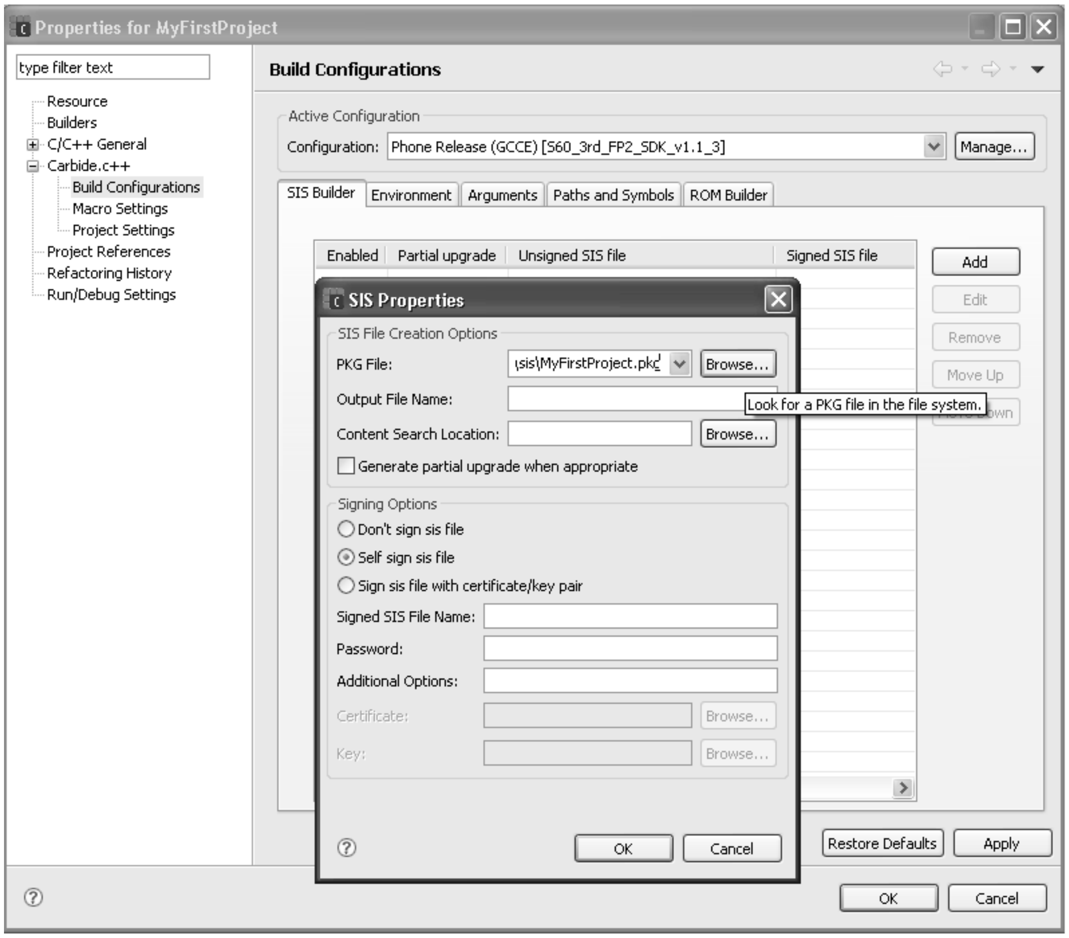
1. Self-Sign SIS File – If the application has only user-grantable (or no) capabilities, the self-signed option can be selected and the package file path can be supplied. The Output File Name and Signed SIS File Name text fields can be left blank as they are filled with automatic values. The certificate will also be created automatically [2].
2. Sign SIS file with Certificate/Key Pair – If a developer certificate has been acquired, then the second option can be selected. The output and signed SIS file names are taken as default as before.



Configuration: Phone Release (GCCE) [S60_3rd_FP2_SDK_v1.1_3]

Enabled	Partial upgrade	Unsigned SIS file	Signed SIS file
✓	☐	MyFirstProject.sis	MyFirstProject.sisx

Figure 4.18: Configuring .SIS File Creation



Configuration: Phone Release (GCCE) [S60_3rd_FP2_SDK_v1.1_3]

PKG File: \sis\MyFirstProject.pkg
Output File Name:
Content Search Location:
☐ Generate partial upgrade when appropriate

Don't sign sis file [✓ Self sign sis file] Sign sis file with certificate/key pair

Signed SIS File Name:
Password:
Additional Options:

Figure 4.19: .SIS File Properties Dialogue

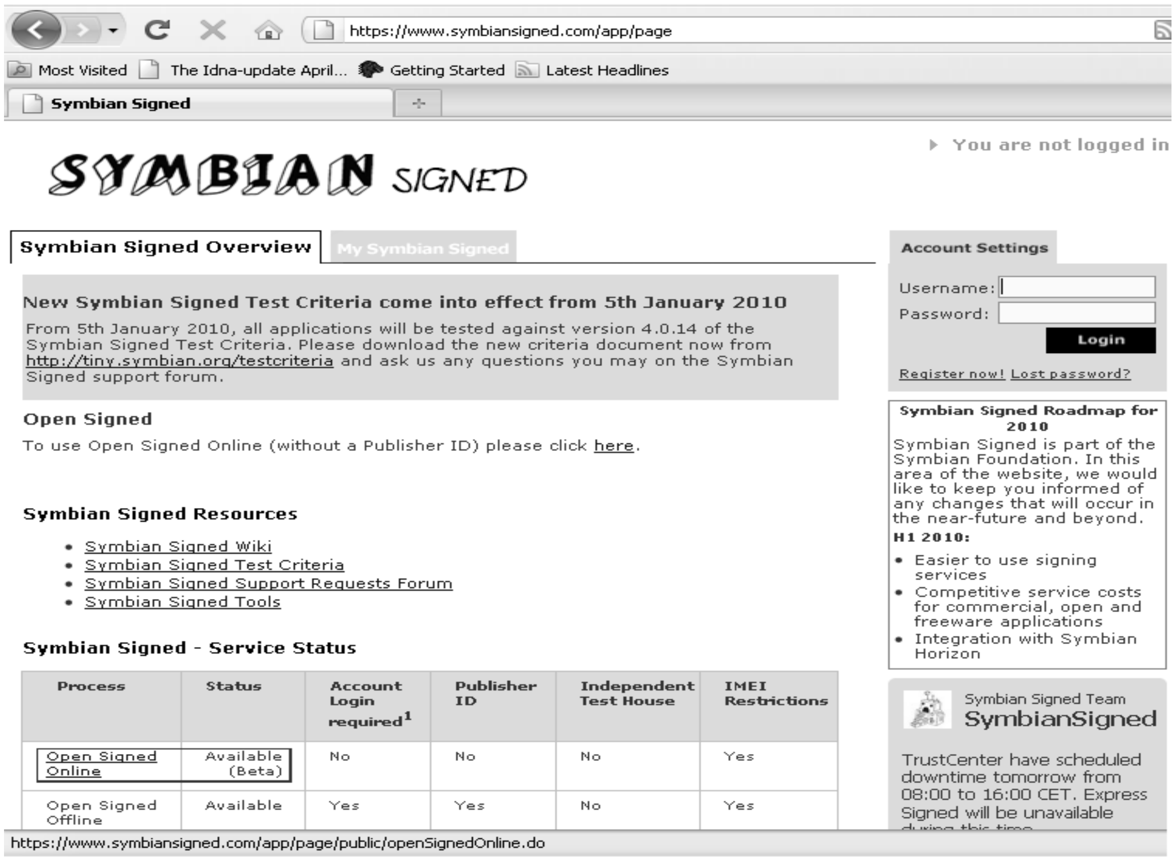
4.4 Symbian Signed and Signing .SIS File

Symbian Signed is the online signing program administered by the Symbian Foundation. To deploy an application on Symbian-based phones, it has to be signed [3]. Once the application has gone through Symbian Signed process successfully, it can be distributed depending upon the signing option.

Signing is the process of encoding digital signature into an application that makes the installation file tamper-proof. The digital certificate identifies the origin of the application by including information on the Publisher ID used during the signing process. Once the application origin is known, it can access more sensitive features of the platform. An unsigned application may not even install on the device depending on the security settings incorporated by the manufacturer [4].

There are various applications signing options available depending upon how widely the application has to be distributed. These are Open Signed Online, Open Signed Offline, Express Signed and Certified Signed. To install an application onto a single device for testing purposes, the Open Signed Online option is used (Figure 4.20).

To sign the application using Open Signed Online, a Symbian Signed account and Publisher ID are not required. Although the application is signed for only one device, it provides signing of applications for free. It only requires a valid email address (this cannot be a public email account like gmail, yahoo or hotmail) and access to the email account during the signing process. The Open Signed Online web-based interface is shown in Figure 4.21. The IMEI number of the device is needed onto which the application has to be installed. This can be obtained using the code *#06#SEND from the handset. The application is signed online against a Developer Certificate and then the signed application can be downloaded from the email account. The Developer Certificate used to sign the application is not available for download using this option.



New Symbian Signed Test Criteria come into effect from 5th January 2010

From 5th January 2010, all applications will be tested against version 4.0.14 of the Symbian Signed Test Criteria. Please download the new criteria document now from http://tiny.symbian.org/testcriteria and ask us any questions you may on the Symbian Signed support forum.

Open Signed

To use Open Signed Online (without a Publisher ID) please click here.

Symbian Signed Resources

- Symbian Signed Wiki
- Symbian Signed Test Criteria
- Symbian Signed Support Requests Forum
- Symbian Signed Tools

Symbian Signed - Service Status

Process	Status	Account Login required[1]	Publisher ID	Independent Test House	IMEI Restrictions
Open Signed Online	Available (Beta)	No	No	No	Yes
Open Signed Offline	Available	Yes	Yes	No	Yes

Symbian Signed Roadmap for 2010

Symbian Signed is part of the Symbian Foundation. In this area of the website, we would like to keep you informed of any changes that will occur in the near-future and beyond.

H1 2010:

- Easier to use signing services
- Competitive service costs for commercial, open and freeware applications
- Integration with Symbian Horizon

Symbian Signed Team
SymbianSigned

TrustCenter have scheduled downtime tomorrow from 08:00 to 16:00 CET. Express Signed will be unavailable

https://www.symbiansigned.com/app/page/public/openSignedOnline.do

Figure 4.20: Symbian Signed Online

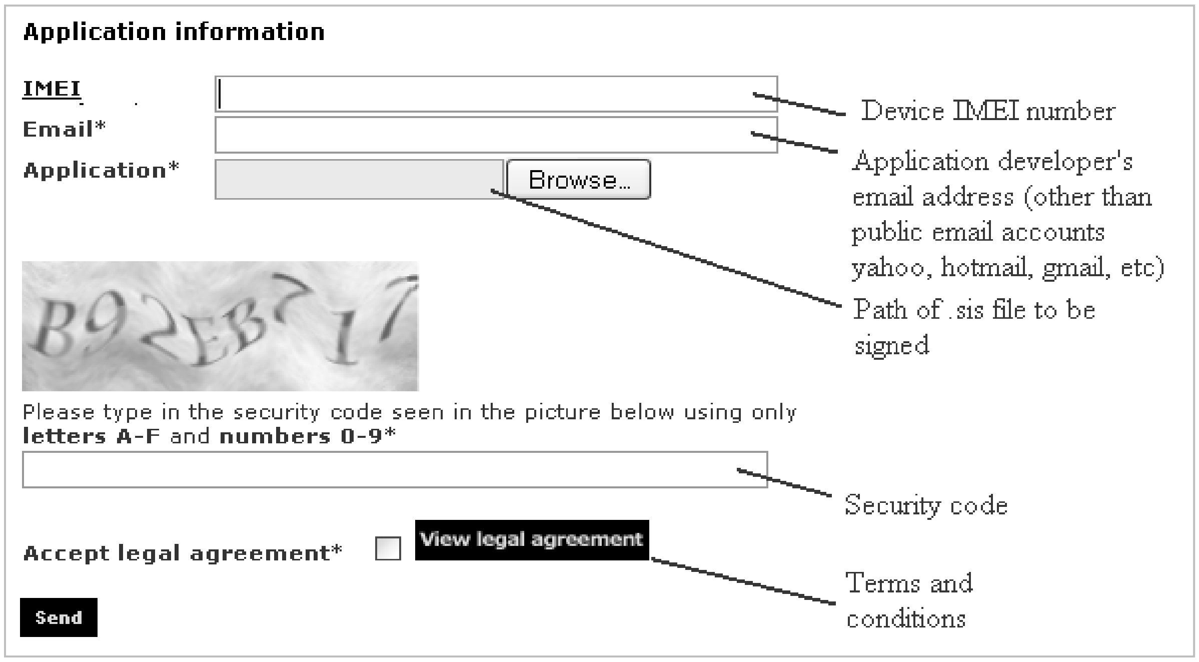

Application information
IMEI
Email*
Application*
Browse...
Device IMEI number
Application developer's email address (other than public email accounts yahoo, hotmail, gmail, etc)
Path of .sis file to be signed
B92EB717
Please type in the security code seen in the picture below using only letters A-F and numbers 0-9*
Security code
Accept legal agreement*
View legal agreement
Terms and conditions
Send

Figure 4.21: SIS File Information

4.4.1 Signing Sis file using makekeys

Sis file can be self signed by using makekeys utility. This signing mechanism is very helpful during application development phase. The signing of sis file and application is carried out in two steps:

Step – 1 creating Certificate and Keys

- Open C:\S60\devices\S60_3rd_FP2_SDK_v1.1\epoc32\tools\makekeys.exe
- Open Cmd
- Drag makekeys.exe to cmd and write command
- -cert -password World123 -len 1024 -dname "CN=World User OU=Development OR=WorldCompany CO=FI EM=World@test.com" WorldKey.key WorldCert.cer

Step-2 Sign Application

- Open SignSis.exe
- Write Command
  - HelloWorld.sis HelloWorld.sis WorldCert.cer WorldKey.key World123

4.5 On-Device Debugging and Testing using TRK

On-device debugging refers to a Symbian feature in which a .sis file already installed on the mobile device can be debugged from the Carbide.c++ IDE via a connection between the PC and the device. It is required in cases where the emulator is unable to fully reflect all device capabilities and to monitor the application behavior on the device.

In Emulator Debug configuration, Carbide.c++ treats the emulator as an application and deals with the emulator as it would deal with any other software being debugged. As a result, the entire emulated Symbian OS system acts as an application that the Carbide.c++ debugger is monitoring [5].

On-device or target debugging works differently. Since Symbian OS is already installed on the device and has its security rules imposed, the debugger cannot control the entire device OS. Also the control interface of the debugger resides on a PC. This means that a debugging controller must exist on the device and that Carbide.c++ needs to communicate with that controller [5]. This controller is the Target Resident Kernel (TRK), a Symbian OS application that is installed on the target device and communicates with Carbide.c++ debugging agent on the PC.

The TRK communicates with Carbide.c++ using a remote debugging protocol that works over a serial connection. Once a serial connection has been established (using USB serial connection or Bluetooth), the TRK acts as a client to the Carbide.c++ debugger and normal debugging process follows (as it does for the emulator debug configuration).

The following section describes a step-by-step procedure for enabling on-device debugging using Carbide.c++.

1. Select debug configuration for the project as shown in the Figure 4.22 and 4.23.

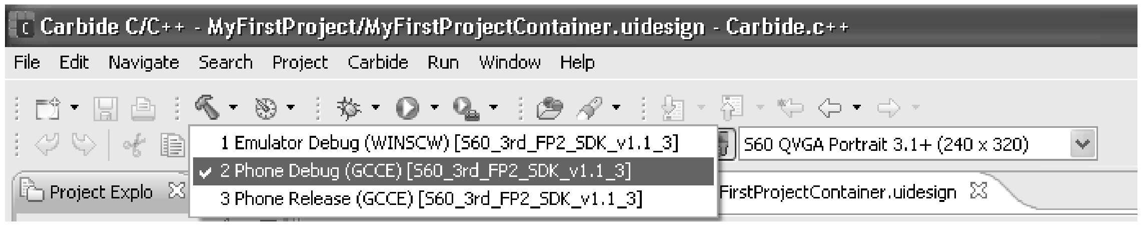

Carbide C/C++ - MyFirstProject/MyFirstProjectContainer.uidesign - Carbide.c++
File Edit Navigate Search Project Carbide Run Window Help
1 Emulator Debug (WINSCW) [S60_3rd_FP2_SDK_v1.1_3]
✓ 2 Phone Debug (GCCE) [S60_3rd_FP2_SDK_v1.1_3]
3 Phone Release (GCCE) [S60_3rd_FP2_SDK_v1.1_3]
S60 QVGA Portrait 3.1+ (240 x 320)
Project Explo
FirstProjectContainer.uidesign

Figure 4.22: Debug Configuration for Device

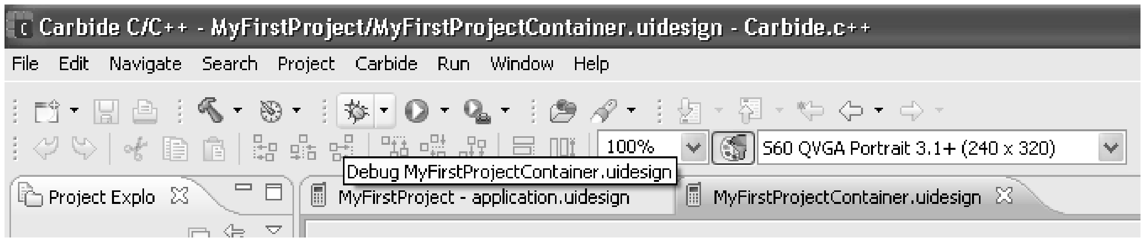
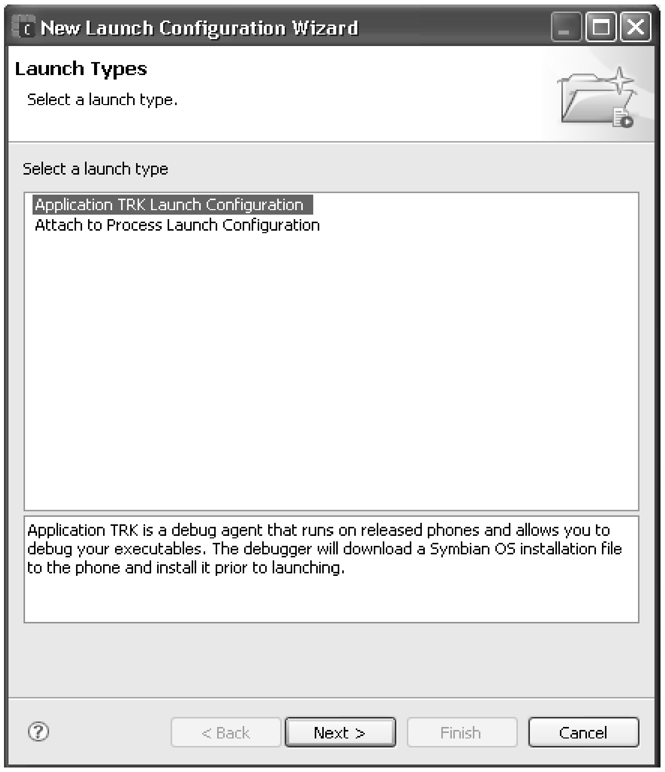

Carbide C/C++ - MyFirstProject/MyFirstProjectContainer.uidesign - Carbide.c++
File Edit Navigate Search Project Carbide Run Window Help
100%
S60 QVGA Portrait 3.1+ (240 x 320)
Debug MyFirstProjectContainer.uidesign
Project Explo
MyFirstProject - application.uidesign
MyFirstProjectContainer.uidesign

Figure 4.23: Debugging on Device

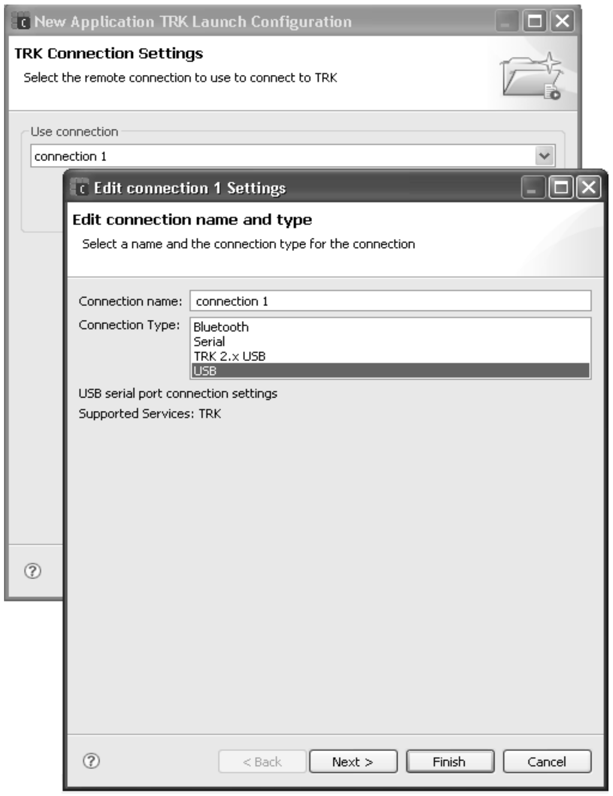
2. Select Application TRK Launch Configuration option as application launch type (Figure 4.24).


New Launch Configuration Wizard
Launch Types
Select a launch type.
Select a launch type
Application TRK Launch Configuration
Attach to Process Launch Configuration
< Back
Next >
Finish
Cancel

Application TRK is a debug agent that runs on released phones and allows you to debug your executables. The debugger will download a Symbian OS installation file to the phone and install it prior to launching.

*Figure **4.24**: Project Launch Configuration*

3. Configure connection with the TRK agent on device as shown in Figure 4.25 and 4.26.


New Application TRK Launch Configuration

TRK Connection Settings

Select the remote connection to use to connect to TRK

Use connection: connection 1

Edit connection 1 Settings

Edit connection name and type

Select a name and the connection type for the connection

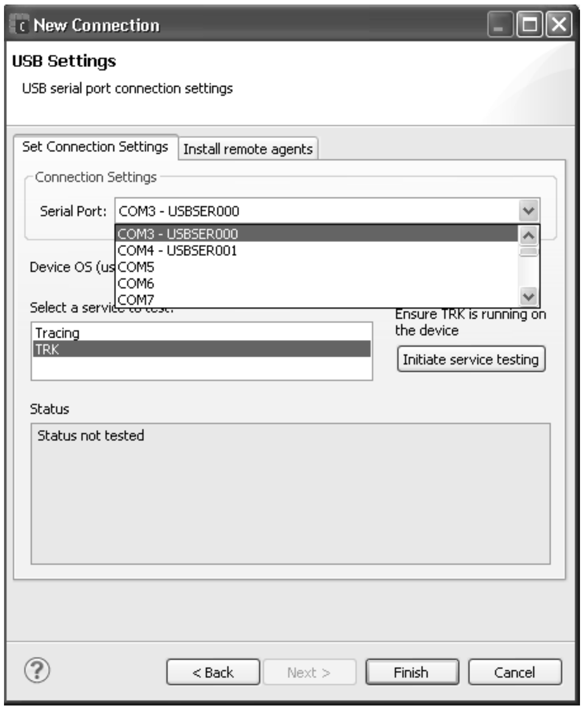
Connection name: connection 1

Connection Type: Bluetooth, Serial, TRK 2.x USB, USB (selected)

USB serial port connection settings

Supported Services: TRK

Figure 4.25: Configuring Connection to the Device


New Connection

USB Settings

USB serial port connection settings

Set Connection Settings | Install remote agents

Connection Settings

Serial Port: COM3 - USBSER000

COM3 - USBSER000 (selected), COM4 - USBSER001, COM5, COM6, COM7

Device OS (us

Select a servic

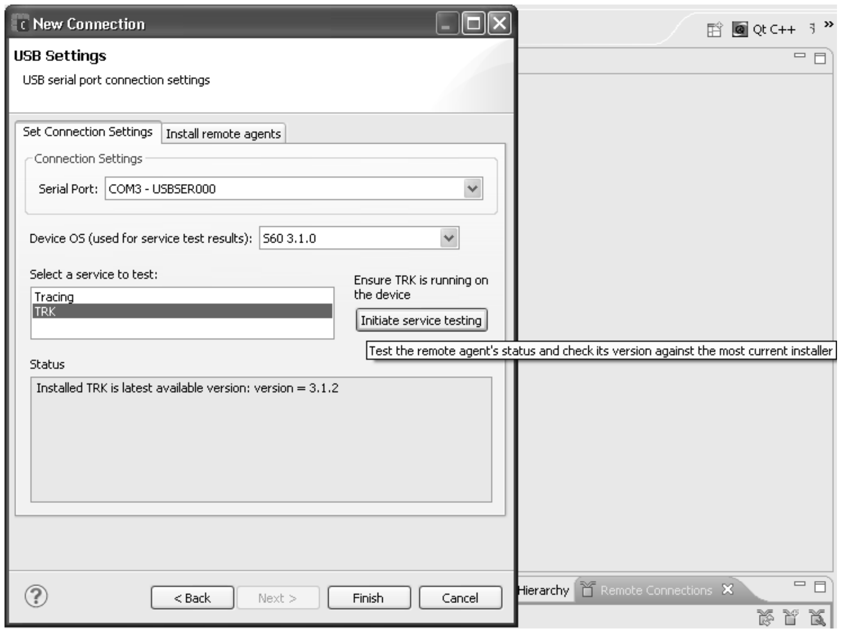
Tracing, TRK (selected)

Ensure TRK is running on the device

Status: Status not tested

Figure 4.26: New Connection Dialogue to Choose Serial Port

4. Test connection settings as shown in Figure 4.27.


New Connection

USB Settings

USB serial port connection settings

Set Connection Settings | Install remote agents

Connection Settings

Serial Port: COM3 - USBSER000

Device OS (used for service test results): S60 3.1.0

Select a service to test:

Tracing

TRK

Ensure TRK is running on the device

Initiate service testing

Test the remote agent's status and check its version against the most current installer

Status

Installed TRK is latest available version: version = 3.1.2

Figure 4.27: Testing Connection to TRK

5. Run debug session.

References

[1]. Symbian (2010) Getting Started with Symbian [online], available: http://developer.symbian.org/wiki/index.php/Symbian_C++_Quick_Start [accessed 18 Aug 2010].

[2]. Symbian (2010) Building a SIS File in Carbide.c++ [online], available: http://developer.symbian.org/wiki/index.php/Building_a_SIS_File_in_Carbide.c%2B%2B [accessed 17 Jul 2010].

[3]. Symbian(2010)Symbian Signed [online], available: http://developer.symbian.org/wiki/index.php/Category:Symbian_Signed [accessed 18 Aug 2010]

[4]. Symbian (2010) Complete Guide to Symbian Signed [online], available: http://developer.symbian.org/wiki/index.php/Complete_Guide_To_Symbian_Signed [accessed 1 Dec 2010].

[5]. Nokia Developer (2010) Carbide.c++ On-device Debugging Quick Start [online], available: http://www.developer.nokia.com/Community/Wiki/Carbide.c%2B%2B_On-device_Debugging_Quick_Start [accessed 10 Dec 2010].

5 Symbian Application Framework

The application framework in Symbian operating system is designed in layers. The Symbian application framework sub-system is called UIKON. It is fundamental to all Symbian GUI applications. The Uikon framework allows for a flexible UI architecture by enabling a variety of GUI frameworks to run on the core operating system. It uses the Model-View-Controller (MVC) design pattern. The application framework provides separate classes for the Model, View and Controller components of the application. The following base classes are provided by Uikon for this purpose.

1. Application Class (CEikApplication)
2. Document Class (CEikDocument)
3. Application User Interface Class (CEikAppUi)

Classes in the Uikon/Eikon framework are labeled with '*Eik'. Each class corresponds to a separate entity of the MVC design as discussed above. The Document class servers as a Model, the Application UI class serves as a Controller whereas the CCoeControl-derived class serves as a View. Uikon itself is based on two important frameworks as shown in the Figure 5.1.

1. CONE – stands for Control Environment. Classes in this sub-system interact with the Symbian window server mainly and provide means for handling user inputs and graphical interaction. These begin with prefix '*Coe', for example CCoeControl.
2. APARC – stands for Application Architecture. Classes in this component provide the basic application architecture and serve as a means to deliver system information to the application and storing data using the Symbian file server. These begin with a prefix '*Apa', for example CApaApplication.

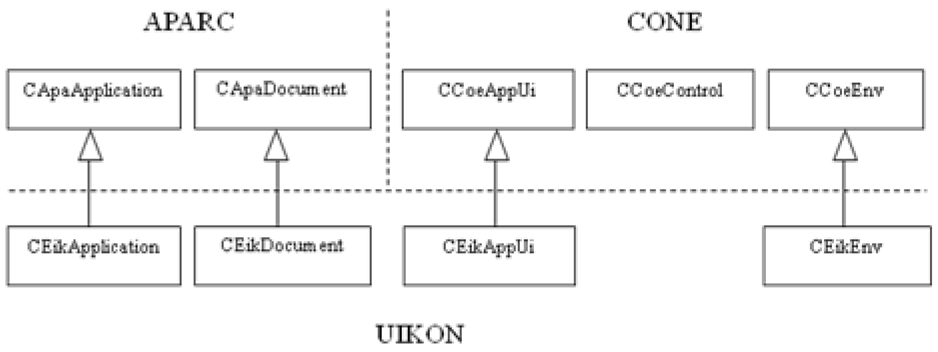


Figure 5.1: UIKON Framework [2]

Uikon provides two GUI frameworks on top of the core OS UI. These are,

1. S60 (Series 60)
2. UIQ (User Interface Quartz)

The following section discusses in detail the S60 application architecture.

5.1 S60 Perspective

S60 and UIQ platforms extend the framework by adding libraries to provide platform-specific controls. The UIQ-specific library is called Qikon and the S60-specific library is called Avkon; these UI layers work on top of the core Symbian OS UI. Each contains different components; however, because they both have UIKON as a base, their APIs are often similar. Whenever a UI application is created, the main classes are derived from platform specific base classes which are in turn derived from the core Symbian OS framework classes (Uikon in this case). Table 5.1 shows the list of Framework classes and their parent S60 and UIQ class.

The CEik prefix of the generic Symbian OS classes is replaced with CQik for UIQ classes and CAkn for S60 classes. This convention is used throughout the UI application framework, for classes, headers and libraries.

Table 5.1: UIQ and S60 Application Framework

Framework Class	Generic UIKON Class	S60 (Avkon) Class	UIQ (Qikon) Class
Application	CEikApplication (inherited from CApaApplication)	CAknApplication	CQikApplication
Document	CEikDocument (inherited from CApaDocument)	CAknDocument	CQikDocument
Application UI	CEikAppUi (inherited from CApaAppUi)	CAknAppUi/ CAknViewAppUi	CQikAppUi
View	CCoeControl	CCoeControl	CQikViewBase (derives from CCoeControl and MCoeView)

UI Application Design

Three common approaches exist for developing UI applications for Symbian OS. These are [1],

1. Traditional Symbian OS Control-Based Architecture
2. Dialog-Based Architecture
3. Avkon View-Switching Architecture

As discussed previously, the Symbian OS application follows a model-view-controller pattern. The term 'view' characterizes any representation of the model's data on the screen and does not refer to any specific UI controls. However, one or more CCoeControl derived UI controls are used in a hierarchy to render a view, where the parent control is called a Container. Each of the above architectures offers different approaches to designing application user interfaces. All provide a means of delivering application data on the screen in form of views, and a mechanism by which users can interact with it. Applications having multiple views have more than one means to display application data on the screen.

The following section discusses salient features of each type of application architectures from an S60 perspective.

5.1.1 Traditional Symbian OS Architecture

The traditional Symbian OS control-based architecture is such that the views are owned by the AppUi directly. These controls are inherited from CCoeControl and the term used for such a class is Container. CCoeControl acts like a blank canvas, on which different UI controls can be drawn. The AppUi class is responsible for handling user initiated view-switch requests, thus providing a mechanism to activate and deactivate containers according to user input. Figure 5.2 illustrates the Traditional Symbian OS Control-based architecture.

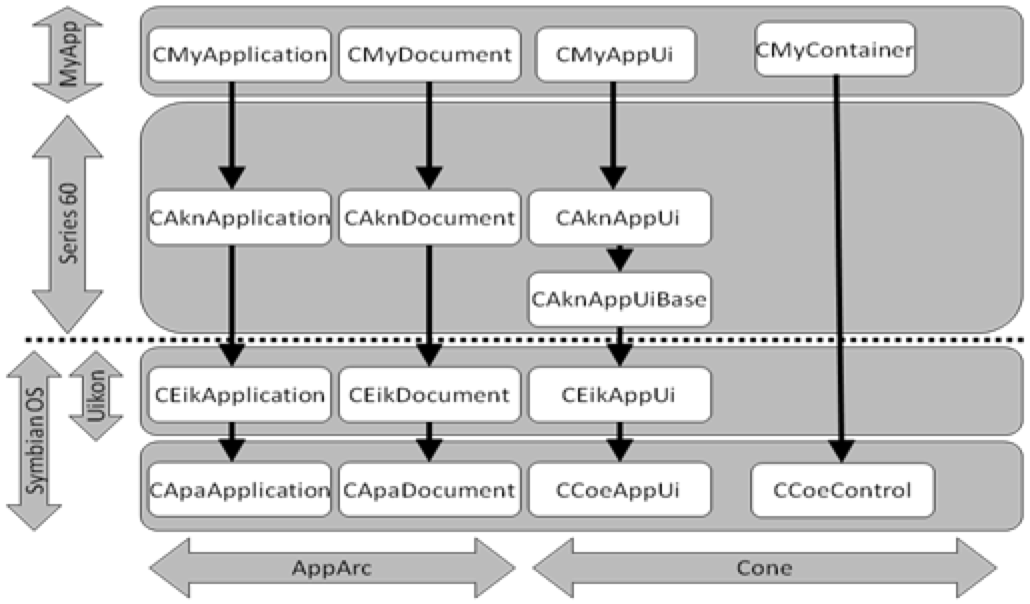


Figure 5.2: Traditional Symbian OS Control-Based Architecture

5.1.2 Dialog Based Architecture

Like the Traditional Symbian OS-Based Architecture just described, the Dialog-Based Architecture similarly establishes the AppUi as the control-owning class. The difference is that the control that it owns inherits directly from one of a family of dialog classes. The idea is to use the built-in features of these classes in order to render data views and to handle switching between them. Dialogs are used extensively by system and application user interfaces for simple notification as well as highly sophisticated data presentation. Dialogs provide a wide variety of ways to interact with a user. They can be used to notify, obtain a response, present fixed information, or to allow the user to enter data. Series 60 provides a comprehensive set of dialog classes and base classes that support the typical dialog functionality required by most applications. Figure 5.3 shows dialog-based application architecture.

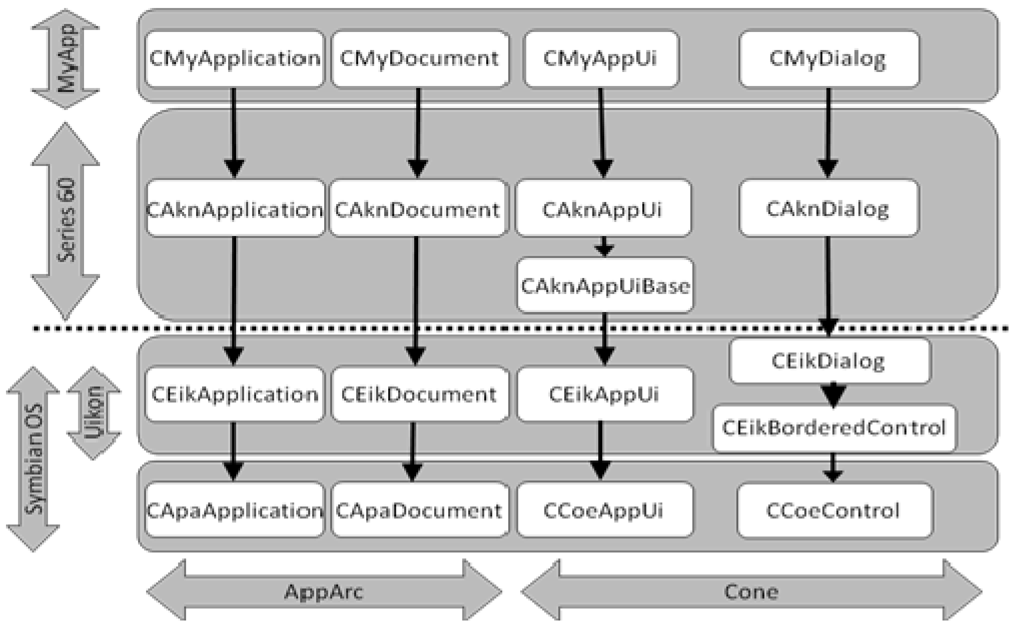


Figure 5.3: Dialog Based Architecture

5.1.3 View Switching Architecture

A characteristic view-switching architecture is shown in Figure 5.4. The AppUi class inherits from the CAknViewAppUi class instead of CAknAppUi in a view switching application. Also, an additional class is added to the architecture between the AppUi and the container class namely the CAknView.

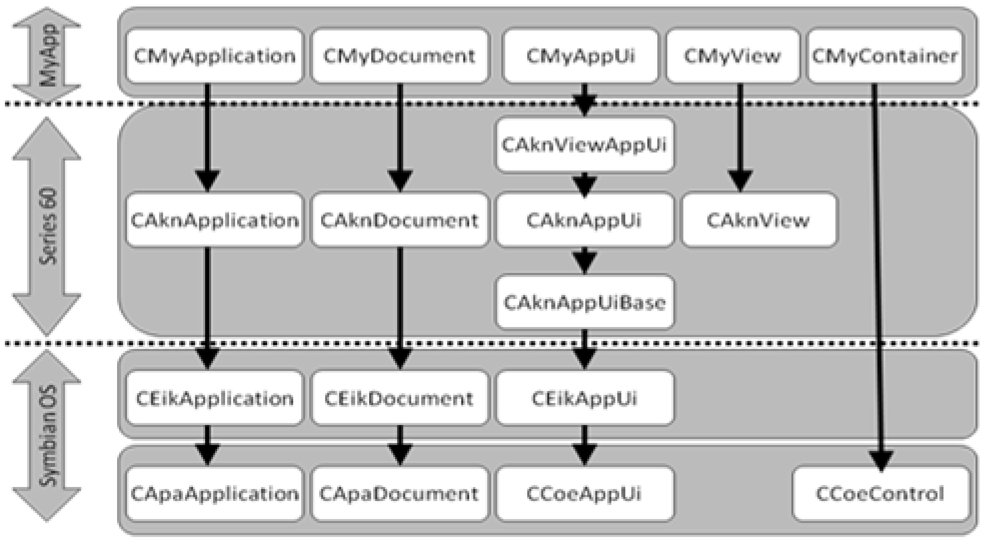


Figure 5.4: View Switching Architecture

In the previous architectures, the AppUi class was directly responsible for the instantiation, deletion and display of view-rendering UI controls. In a view switching architecture another class is introduced between the AppUi and Container, the CAknView based class. In S60, the application UI is derived from the CAknViewAppUi class instead of the standard CAknAppUi. Now the AppUi only calls view activation functions which make an activation request to the View Server. Each application registers its views with the Symbian OS View Server.

The role of view server is to ensure that only one view is active per application, at any time. Avkon views are identified by two UIDs: one to identify the application and second to uniquely identify the view within that application. Figure 5.5 shows the view server and its relationship with application views.

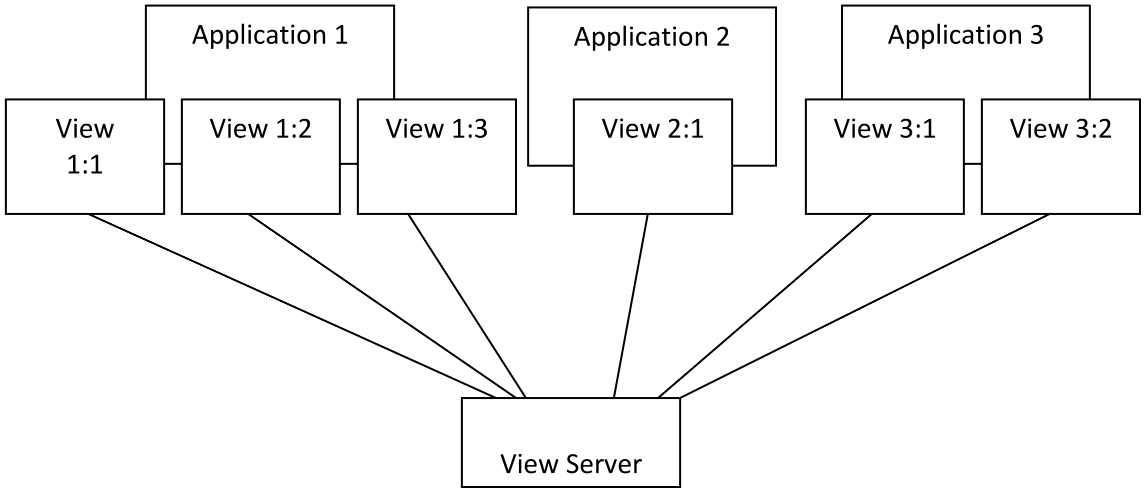


Figure 5.5: View Server Relationship with Application Views [2]

References

[1]. Harrison R. and Shackman M. (2003) "Symbian *OS C++ for Mobile Phones*" John Wiley and Sons.

[2]. *Coulton P. and Edward*s R. (2007) "*S60 Programming: A Tutorial Guide*" John Wiley and Sons.

[3]. Babin S. (2008) "*Developing Software For Symbian OS: A Beginner's Guide to Creating Symbian OS V9 Smartphone Applications in C++*" Wiley India Pvt. Ltd.

[4]. Stichbury J. and Jacobs M. (2006) "*The Accredited Symbian Developer Primer: Fundamentals of Symbian OS*" Wiley.

[5]. Talukder A. and Roopa Y. (2006) "*Mobile Computing : Technology, Applications, and Service Creation*" McGraw-Hill

6 Developing a HelloWorld Application

This section explains the architecture of a Symbian OS view-switching application. To start with, a sample HelloWorld application is developed and its components are explained.

Start Carbide.c++, go to File → New → Symbian OS C++ Project and select GUI Project with UI Designer from the options in the dialog box. The project creation wizard guides through the steps as described in the previous chapter.

6.1 Architecture

All component classes derive from core Symbian OS classes (Application architecture and Control Environment). Application classes can be divided into four main categories: (i) View, (ii) Document, (iii) Application and (iv) Application UI (AppUi). These classes interact in the following way (Figure 6.1).

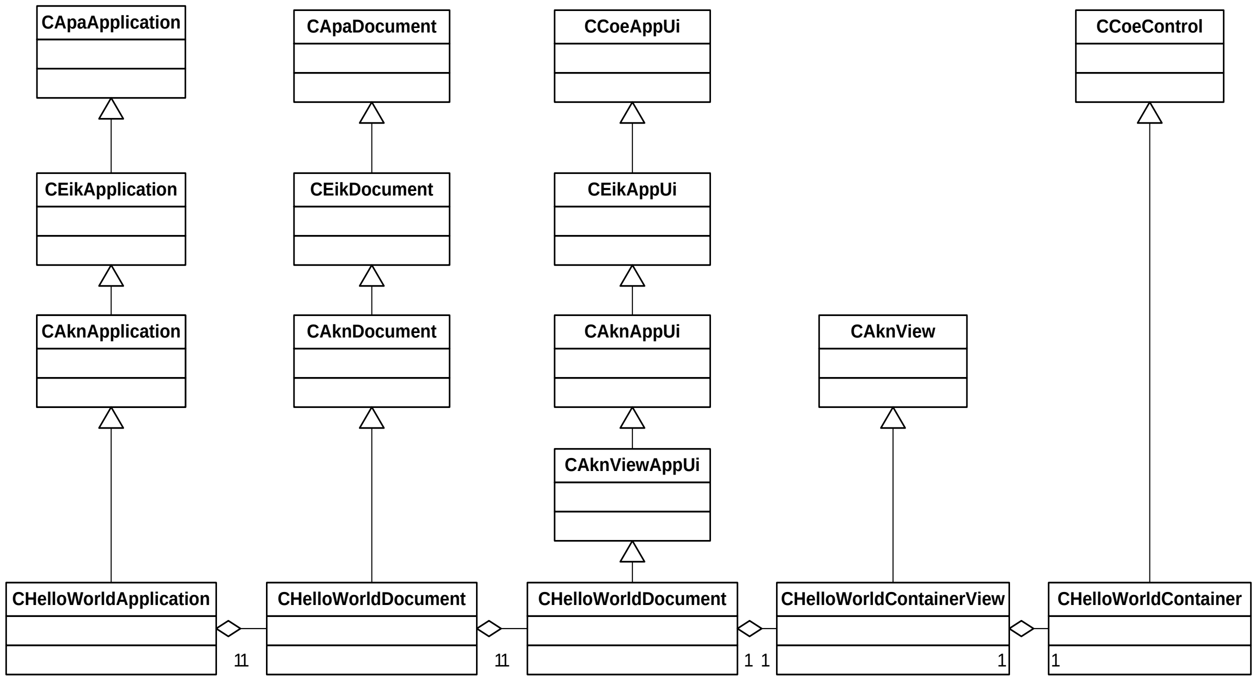


Figure 6.1: HelloWorld View Switching Application Architecture

CHelloWorldApplication – The Application class has a fairly static role and is the least coupled. It does not involve itself with application's data and algorithms. It represents the properties that are the same for every instance of the application such as registration information, capabilities, and the UID. Two functions of this class must be implemented namely AppDllUid() and CreateDocumentL(). AppDllUid(), inherited from CApaApplication, supplies a globally unique 32-bit identifier [3] which is always associated with the application. This class is responsible for creating the document class object through the CEiKDocument::CreateDocumentL() function.

CHelloWorldDocument – Created by the application class, the CHelloWorldDocument class is responsible for persisting and internalizing data [3]. This class represents the 'model' component of the MVC Symbian application architecture. For example, in a file based application the document class represents the data in the file. If this data is modifiable, the application requires the document to create an application user interface that can be used to edit the document. In applications that do not have persistent storage requirements, the document class simply instantiates the AppUi class through CreateAppUiDL() function inherited from CEikDocument.

CHelloWorldAppUi – CHelloWorldAppUi represents the 'controller' part of the MVC pattern. It captures user input in the form of key presses, mouse movements and menu commands [4]. AppUi acts like a global event handler for all events generated by the application. Its role is to get commands to the application and distribute key strokes to controls and application views. Application views are constructed and owned by the AppUi. Thus, the AppUi class changes the Model data based on input received from the user and reflects the changes on the application View. Two functions HandleKeyEventL() and HandleCommandL() are used to perform event handling. In a view switching architecture this class must be derived from CAknViewAppUi

CHelloWorldContainerView – Views provide user entry point into the application. The view class provides screens of application and is the 'view' part of the MVC architecture. An application can have multiple views, for example a camera application can have main picture capturing view and a photo album view which displays a list of photos stored in the memory of phone. The CHelloWorldContainer class is derived from CAknView and acts as a view controller. It is responsible for creating corresponding CCoeControl-derived container objects, registering controls for event handling, and retrieving menu resources from resource files. Two methods DoActivateL() and DoDeactiavteL() of this class are used to switch between multiple views of an application.

CHelloWorldContainer – This is the CCoeControl-derived class that displays data on the screen using controls. It serves as the main window of the application and all other controls are drawn by this. CHelloWorldContainer implements four methods from CCoeControl—all of them are called by the framework. SizeChanged() allows the control to respond to a change in its size. Draw() is called to draw the control. CountComponentcontrols() returns the number of controls the Container owns. For each control owned by the Container, the framework makes a call to ComponentControl() to retrieve it.

6.2 Application Initialization

Two functions are called by the framework to start an application. These must be implemented by all Symbian OS applications.

1. CApaApplication* NewApplication()
2. TInt E32Main()

The Figure 6.2 illustrates the application initialization process. The function NewApplication() is a non-leaving function which creates an instance of the CHelloWorldApplication class and returns a pointer to it. It returns NULL if the application class cannot be instantiated.

For example,

```
LOCAL_C CApaApplication* NewApplication(){

return new CHelloWorldApplication;

}
```

The second function serves as the entry point of the application. E32Main() calls EikStart::RunApplication() method which takes argument a pointer to the NewApplication() function. This creates an instance of the application class.

```
GLDEF_C TInt E32Main(){

return EikStart::RunApplication( NewApplication );

}
```

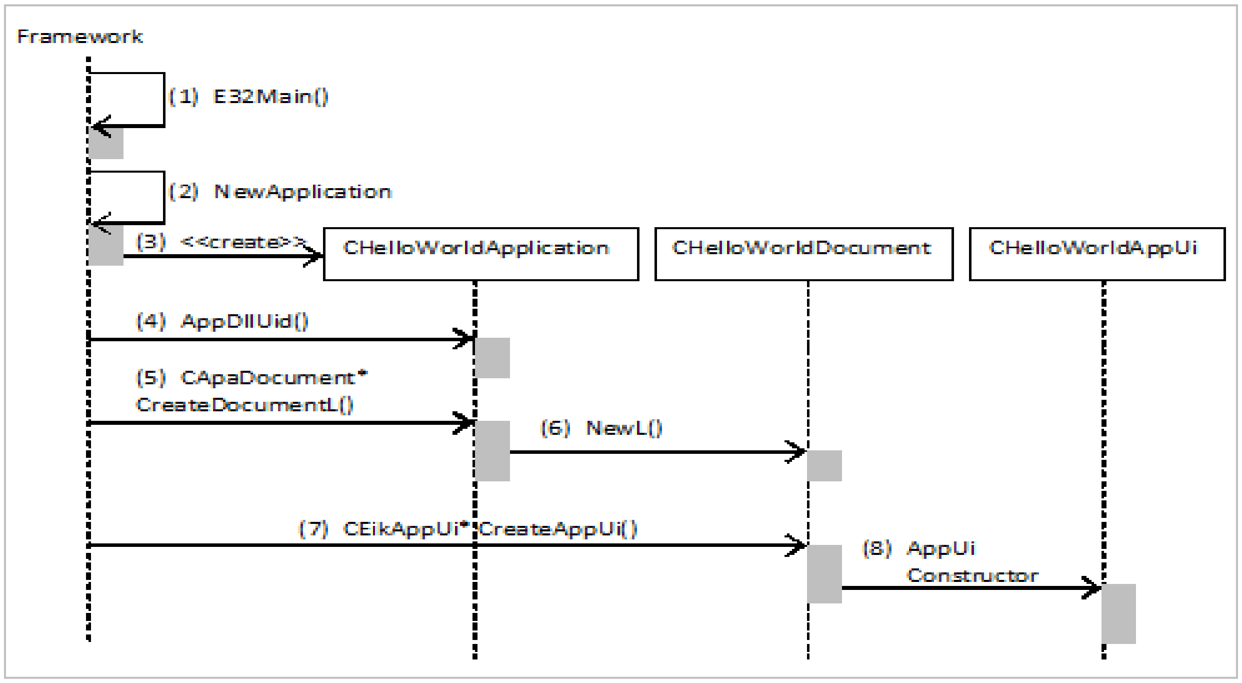


Figure 6.2: Application Initialization Steps [1]

6.3 Application Files

In addition to the above mentioned source files, Symbian OS applications include other files as well usually under the /data, /sis, and /group directories. These are described in the following sections.

6.3.1 Resource Files

A resource file is a text file with a .rss file extension as shown in Figure 6.3. It is used to specify the user interface components and their properties separate from the source files. These include UI elements such as menus, dialogs and lists, as well as any user-visible text (application name, etc) used by the application. Every S60 application has at least one resource file associated with it. If an application has multiple views, each container corresponding to a view has its own resource file. These are included in the main application's .rss file using #include preprocessor directive.

```
NAME HELL

#include <avkon.rsg>
#include <avkon.rh>
#include <eikon.rh>
#include <appinfo.rh>
#include "HelloWorld.hrh"
#include "HelloWorld.loc"

RESOURCE RSS_SIGNATURE
      {
      }
RESOURCE TBUF
      {
      buf = "HelloWorld";
      }
RESOURCE EIK_APP_INFO r_application_hello_world_app_ui
      {
      cba = R_AVKON_SOFTKEYS_OPTIONS_EXIT;
      status_pane = r_application_status_pane;
      }
RESOURCE STATUS_PANE_APP_MODEL r_application_status_pane
      {
      }
RESOURCE LOCALISABLE_APP_INFO r_localisable_app_info
      {
      short_caption = STR_HelloWorldApplication_5;
      caption_and_icon = CAPTION_AND_ICON_INFO
            {
            caption = STR_HelloWorldApplication_4;
            number_of_icons = 0;
            };
      }
RESOURCE TBUF r_application_akn_view_reference1
      {
      }

RESOURCE TBUF r_application_akn_view_reference2
      {
      }
```

Figure 6.3: Sample Resource File for HelloWorld Symbian OS Application

A resource file is compiled into a binary file (.rsg) by the resource compiler (called RCOMP), as part of the standard abld build process. This binary file is in turn included in the project's source (.cpp) files. The .rsg file is opened by the application framework when the application starts and individual resources are loaded into source code using resource identifiers specified in the .rsg file. For e.g., the HelloWorldContainer.rssi contains the following resource,

```
RESOURCE TITLE_PANE r_hello_world_container_title_resource

{

txt = “HelloWorld”;

}
```

This is used in the HelloWorldContainer.cpp file as follows,

```
TResourceReader reader;

iEikonEnv->CreateResourceReaderLC( reader, R_HELLO_WORLD_CONTAINER_TITLE_RESOURCE );

title->SetFromResourceL( reader );
```

Here **R_HELLO_WORLD_CONTAINER_TITLE_RESOURCE** is the resource identifier as compiled in the .rsg file. Figure 6.3 shows the syntax of the HelloWorld.rss resource file.

Keeping resource information separate from source code provides modularity; the interface level appearance of the application can be substantially modified, the need to change the source code or recompile the application. This makes applications much easier to localize as for only the resource file needs to be recompiled. Different translations of the same text in multilingual applications can be defined in separate files which are in turn referred to in .rss files. This allows a multilingual application to be supplied as a single executable along number of language-specific resource files [2].

6.3.2 Hrh Files

.hrh files in Symbian OS applications provide a means to define commands and view identifiers that are in used in the application toolbars, etc. This file contains a list of enumerations that are used in the .rss, .h and .cpp files. The HelloWorld.hrh file has the following contents.

```
Name       : HelloWorld.hrh

 Author     :

 Copyright   : Your copyright notice

 Description :

============================================================================

*/

enum THelloWorldViewUids
```

```
{

EHelloWorldContainerViewId = 1,

EHelloWorldListBoxViewId

};
```

The above code defines identifiers for the two application views. These are used to refer to views in the source code. For example, in CHelloWorldContainer::HandleResourceChangedL() function, the window is set to the container view using EHelloWorldContainerViewId defined above.

SetRect(**iAvkonViewAppUi**->**View**(TUid::**Uid**(*EHelloWorldContainerViewId*))->**ClientRect**());

6.3.3 Localization Files

In addition to resource file, the Nokia S60 project creation wizard also defines localization files with .loc extension. These files contain string constants used by the applications. Localization files aid in defining localized strings for each of the languages that the application supports. For multilingual applications, the application resource file (HelloWorld.rss) is locale independent, where as the .loc file (HelloWorld.loc) contains relevant information about all the locales that the application supports. For example HelloWorld.rss includes HelloWorld.loc which in turn has the following,

```
#ifdef LANGUAGE_01

#include "HelloWorld.l01"

#endif
```

Each individual .loc (*.l01, *.l02 and so on) file then contains all the text strings for each locale. The project definition file contains all the locales for which the application has to be compiled. HelloWorld.l01 contains the following static text strings.

```
// localized strings for language: UK English (01)
#define STR_HelloWorldApplication_3 ""
#define STR_HelloWorldApplication_4 "HelloWorld"
#define STR_HelloWorldApplication_5 "HelloWorld"
#define STR_HelloWorldApplication_1 ""

#define STR_HelloWorldApplication_2 ""
```

6.3.4 Project Definition (.mmp) File

The project definition file defines all the components that the project requires, including source files, bitmap files, and library files and specifies other compile time options for the project. It has extension '.mmp'. The Figure 6.4 shows the content of HelloWorld.mmp file.

```
TARGET            HelloWorld.exe
UID               0x100039CE 0xEBCD6169
VENDORID          0
TARGETTYPE        exe
EPOCSTACKSIZE     0x5000

SYSTEMINCLUDE     \epoc32\include \epoc32\include\variant
\epoc32\include\ecom
USERINCLUDE       ..\inc ..\data

SOURCEPATH        ..\data
START RESOURCE    HelloWorld.rss
HEADER
TARGETPATH        resource\apps
END //RESOURCE

START RESOURCE    HelloWorld_reg.rss
TARGETPATH        \private\10003a3f\apps
END //RESOURCE

LIBRARY           euser.lib apparc.lib cone.lib eikcore.lib avkon.lib
LIBRARY           commonengine.lib efsrv.lib estor.lib eikcoctl.lib
eikdlg.lib
LIBRARY           eikctl.lib bafl.lib fbscli.lib aknnotify.lib aknicon.lib
LIBRARY           etext.lib gdi.lib egul.lib insock.lib
LIBRARY           ecom.lib InetProtUtil.lib http.lib esock.lib

LANG              01

START BITMAP      HelloWorld.mbm
HEADER
TARGETPATH        \resource\apps
SOURCEPATH        ..\gfx
SOURCE            c12,1 list_icon.bmp list_icon_mask.bmp
END

SOURCEPATH        ..\src

#ifdef ENABLE_ABIV2_MODE
DEBUGGABLE_UDEBONLY
#endif

SOURCE            HelloWorldContainerView.cpp HelloWorldContainer.cpp
HelloWorldApplication.cpp HelloWorldAppUi.cpp HelloWorldDocument.cpp
HelloWorldListBoxView.cpp HelloWorldListBox.cpp
```

Figure 6. 4: Sample MMP file for HelloWorld Symbian OS Application

TARGET is the name of the application including its extension, for example HelloWorld.exe.

TARGETTYPE gives the extension for the application, which is 'EXE' in the above case.

UID specifies identifiers for the application.

- UID1 specifies the category of an object. In S60 application projects, UID1 is automatically specified by the build tools based on the TARGETTYPE keyword of the project mmp file.
- UID2 indicates the type of application and is fixed for all applications in one category [2]. For example Symbian OS GUI based applications have UID 2 value 0x100039CE, where as static DLLs have UID 2 value 0x1000008d [5].
- UID 3 is used to identify the application itself (i.e. a particular exe or dll file). UID3 is used among others in the following places of your application:
    - in the project mmp file
    - in your application code (in the CAknApplication class)
    - in the pkg file

TARGETPATH defines the location where the built application will be released. For Win32 platforms the target path will be interpreted as a location on the z: drive, and the release path will therefore be %EPOCROOT%\epoc32\release\*platform**variant*\z*target-path*\. In the HelloWorld example it is %EPOCROOT%\epoc32\release\winscw\udeb\z\resource\apps.

SOURCEPATH is the location of the source file to be compiled. There can be more than one such statements; however, the build tool requires subsequent SOURCE and RESOURCE statements. For example SOURCEPATH../src specifies relative path to the /src directory.

SOURCE lines follow the SOURCEPATH statement and define the source files as *<name>.extension*.

START RESOURCE defines the resource files in the application. These are files in the /data directory. This is followed by a TARGETPATH statement which specifies the path on z:\ drive where the resources are compiled and an END statement.

STARTBITMAP defines how bitmap (.bmp) files can be compiled to Symbian OS multibitmap (.mbm) file. It is followed by TARGETPATH and SOURCEPATH statements specifying where the bitmap files can be loaded and where the .mbm files are placed after compilation. SOURCE statement specifies the names and color depths of bitmap files that are to be compiled. For example c12,1 list_icon.bmp list_icon_mask.bmp tells that both the images are colored and have color depths of 12 and 1 bits per pixel, respectively.

LANG defines the languages supported by the application using a two digit code.

USERINCLUDE defines the path for the include directory. The linker checks this directory for the files specified with #include statement in the source and resource files.

SYSTEMINCLUDE is used to define the directories containing system specific header files like eikstart.h, e32def.h, etc.

LIBRARY is a list of libraries that the application uses at run time. For example, if the application intends to establish a session with the SendAs server to send messages, then sendas2.lib should be specified in the project definition file.

CAPABILITY attribute indicates permissions that are required in the applications certificate to be able to install the application on device. This allows the application to access sensitive platform functionality [6]. For example the capability ReadUserData allows read access to data belonging to the phone user, such as contacts, messages and calendar data.

6.3.5 Project Build File

Project build file is a component definition file containing the list of project files. Build file of project resides under the /group with name bld.inf. Build files are used by `bldmake` to define the `abld.bat` and makefiles to be created. The file is made up of a number of sections, with headers. Each section header can appear any number of times in the file (including none). The code snippet of build file is shown in figure

```
PRJ_PLATFORMS

WINSCW ARMV5 GCCE

#ifdef SBSV2

PRJ_EXTENSIONS

      START EXTENSION s60/mifconv

      OPTION TARGETFILE testing_0xE232FAFA.mif

      OPTION HEADERFILE testing_0xE232FAFA.mbg

      OPTION SOURCEDIR ../gfx

      OPTION SOURCES -c32 qgn_menu_Testing

      END

#else

PRJ_MMPFILES

#endif

PRJ_MMPFILES

Testing.mmp
```

Exercises

1. Create a S60 application of type "GUI application with UI Designer". Generate a random number in the range of 1 -100 and show the random number in number Editor.
2. A hospital wants to view information regarding its indoor patients in English and Spanish. The information to show include
    - Name of the patient
    - Date of admission

   User wants to view the English information in one view and Spanish in another view. User can switch from one view to other by using options menu. Create a multi-view application to show the information in English in one view and Spanish in the second view.

References

[1] Edwards L. , Barker R. , and Staff of EMCC Software Ltd. (2004) "*Developing Series 60 Applications: A Guide for Symbian OS C++ Developers*", Addison-Wesley.

[1]. Harrison R. and Shackman M. (2007) "*Symbian OS C++ for Mobile Phones: Application Development for Symbian Os* ", Volume 9. John Wiley and Sons

[2] Coulton P. and Edwards R. (2007) "*S60 Programming: A Tutorial Guide*", John Wiley & Sons

[2]. Nokia Forum (2010) Description of the Classes Automatically Created with the Project [online], available: http://wiki.forum.nokia.com/index.php/Description_of_the_classes_automatically_created_with_the_project [accessed 10 Aug 2010].

[3]. Nokia Forum (2010) Application UIDs [online], available: http://library.forum.nokia.com/index.jsp?topic=/S60_5th_Edition_Cpp_Developers_Library/GUID-EA05F9B6-52C7-4BD9-8B9A-4BA3456E70B5.html [accessed 10 Aug 2010].

[4]. Symbian (2010) Capabilities (Symbian Signed) [online], available: http://developer.symbian.org/wiki/index.php/Capabilities_%28Symbian_Signed%29 [accessed 10 Aug 2010].

7 Localized SMS Application

Short Messaging Service (SMS) provides a mechanism to send and receive text messages using mobile phones. When a user sends an SMS to a recipient, it is actually first sent to an SMSC, which stores the message and then forwards the message to the intended recipient. The following figure shows a typical SMS architecture [1].

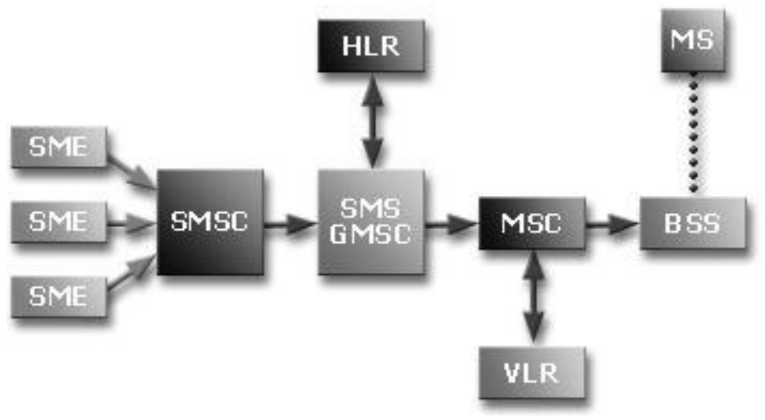


Figure 7.1: SMS Architecture [1]

Figure 7.1 shows a high-level architectural diagram of how short messages are transferred from one mobile device to another.

The SMSC (Short Message Service Center) is responsible for storing and forwarding of messages from one mobile station to another. The SME (Short Message Entity) is typically a mobile phone or a GSM modem, which can be located in the fixed network or a mobile station, and actually sends and receives short messages.

The SMS GMSC (SMS Gateway MSC) is a gateway MSC that can also receive short messages. The gateway MSC is a mobile network point of access to other networks. On receiving the short message from the SMSC, the GMSC queries the current position of the mobile station form the HLR.

HLR (Home Location Register) is the main database that holds information about subscribers in a mobile network e.g. the subscription profile and routing information for the subscriber. The latter is usually determined by the area covered by a MSC where the mobile is currently located. The GMSC is thus able to pass on the message to the correct MSC using the HLR.

An MSC (Mobile Switching Center) is the entity in a GSM network which does the job of switching connections between mobile stations and fixed network.

Each network also maintains a VLR (Visitor's Location Register) which contains temporary information about a mobile station. This includes mobile identification and the cell (or a group of cells) where it is currently located. Using information from VLR, the MSC transfers short messages to the corresponding BSS (Base Station System) which in turn forwards the short message to the mobile. The BSS consists of transceivers, which send and receive information over the air interface to and from the mobile station. This information is passed out-of-band over the signaling channels so the mobile can receive messages even if a voice or data call is going on.

The following sequence of steps takes place when an SMS is sent from one mobile to another.

1. The short message is submitted from the SME to the SMSC.
2. After completing its internal processing, the GSMSC obtains routing information for the mobile subscriber from its HLR.
3. The GSMSC sends the short message to the MSC.
4. The MSC retrieves subscriber information for the mobile device from the VLR.
5. The MSC transfers the short message to the Mobile Station (MS) through the corresponding BSS.
6. The MSC returns the status of this forwarding operation to the SMSC (whether message was delivered successfully or not).
7. SMSC returns a status report indicating delivery of the short message, if requested by the SME.

SMS messaging is used in a number of applications like news feeds, weather reports, chat services, ecommerce transactions, etc. SMS applications can be built by interfacing them to an SMSC. However, each vendor’s SMSC supports its own protocol. Standard protocols are not common and application complexity increases due to varied protocols, platforms and environments.

In addition to person to person text messaging, applications involving content provision to mobile users also use SMS services. The localized SMS provides facility in different local languages other than English. The following sections discuss the features of this application.

7.1 Application Overview

The SMSLocalized application is a Symbian application designed for the languages supported through Pango.

The application is implemented for the Urdu language, chosen for its complexity in contextual shaping and positioning of glyphs. While this application gives examples of Urdu language, it should be noted that it can quickly be customized for other languages. Possible customization mechanisms have been suggested in this document wherever applicable.

7.1.1 Application Features

The localized SMS application provides the following functionalities.

1. Allows typing of text message in complex scripts using open type fonts such as Nafees Nastaleeq in Urdu and Arabic.
2. Provides a customized on-screen keyboard layout for Urdu language.
3. Sends and Receives messages in Urdu language.
4. Integrated with Pangocairo library for text layout and rendering.

7.2 Application Design

The Figure 7.2 shows the class diagram of application:

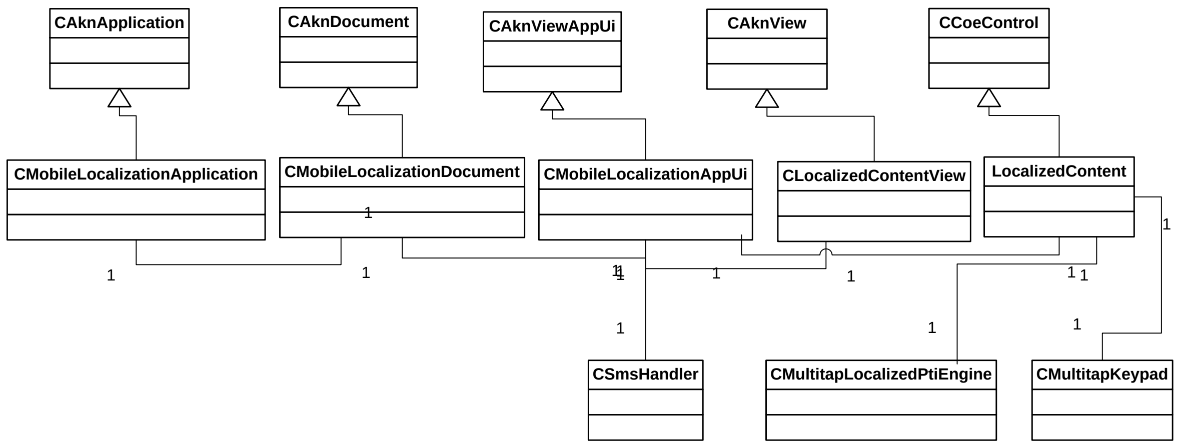


Figure 7.2: SMS Application Design

The application is organized into the following classes.

1. CMobileLocalizationApplication – This class is inherited from class CAknApplication and serves as the main entry point of the application.
2. CMobileLocalizationDocument – Provides a method to instantiate the AppUi class.
3. CMobileLocalizationAppUi–The CMobileLocalizationAppUi class inherits from the CAknViewAppUi, and is responsible for creating and setting default view and container to window when the application starts.
4. CLocalizedContentView– This class inherits from CAknView and is responsible for creating container objects that are part of the view. Moreover, this class also handles commands that are given in the options menu through the HandleCommandL() function. For example to send an SMS, the user selects 'Send' from the options menu. Message sending functionality is implemented in a separate HandleSendMenuItemSelectedL() function which is called from the HandleCommandL() function.
5. LocalizedContent – Inherited from CCoeControl. This class embeds functionality for handling individual key events and loading custom keymap against each key using the PtiEngine object. This class also interfaces with Pango for rendering text in desired script. This class provides following features
    a. Interface with CMultitapKeypad class and draws keypad at application startup.
    b. Holds reference to CPtinEngine class object.
    c. Holds reference to RichText Control in which rendered text is displayed.
    d. Contains the function 'OfferKeyEventL'. All the low-level key events are delivered to this function for processing. This function extracts Unicodes of the keys pressed and passes them to Pango for rendering.
    e. Inserts the text rendered by Pango in RichText component.

6. CMultitapLocalizedPtiEngine – This class contains a reference to Text Input Engine (PtiEngine) object and loads keymap during initialization.
7. CMultitapKeypad – This class is used to draw a custom localized keypad on the mobile screen. This involves measuring screen size and dividing it appropriately to make sufficient space for a numeric keypad consisting of four rows and three columns.
8. CSmsHandler – This class inherits from CActive and registers an active object with the scheduler. It implements methods to handle messages received and send by the application.

7.2.1 Definition of Custom Keyboard

To enable Urdu text input on mobile phones, a custom key map has to be defined so that appropriate characters of Urdu are rendered against each key press. Mobile phones normally support multi-tapped text input, where each key on the keypad represents more than one characters. The desired character/letter is typed by pressing the key repeatedly multiple times. For example, key '2' on the keypad shows sequence 'abc'. To type letter 'c', this key has to be pressed three times in a row. This arrangement of character sequences against each numeric key on the mobile phone is called the Keymap i.e. each numeric key on the device has an associated keymap.

To prevent the operating system from loading the default keymap and enable loading the customized keymap for a local language, a new keymap has to be defined and some mechanism needs to be developed to load this sequence of characters when the application starts up. This involves defining a custom Unicode sequence for each key on the numeric keypad. The PtiEngine API of Symbian S60 framework is used to load customized keymap sequences from the relevant resource file. This API provides text input services in different languages and input modes.

The main client-side class used to define and load a custom localized keymap is CPtiEngine. It is used to select and activate available languages, enable specific input modes and text cases, redefine the keymap and perform actual input functions.

A custom on-screen keyboard needs to be defined to enable text input in the languages not supported on the mobile device. Definition of keyboard requires completion of the following tasks:

- Define and Load Customized Keymap
- Install and Load Fonts
- Layout and Load the Keyboard

7.2.1.1 Install and Load Custom Fonts

Symbian OS gives two classes to handle fonts. *TFontSpec* is a device-independent font specification supporting the name, height and style attributes, usually measured in twips or any device-independent unit. *CFont* provides a device-dependent font used to obtain pixel width of a string of characters. The function GetNearestFontInTwips() is used to map from a TFontSpec to a CFont object through a graphics device.

Carry out following tasks to install and load a font:

1. Place the font file in appropriate folders as mentioned below:
    a. Folder for *S60 Emulator on Windows*
        i. %EPOCROOT%\release\winscw\udeb\z\resource\fonts
    b. Folder for Symbian Mobile Devices
        i. resource\fonts\   (In any directory)

    For instance, place Nafees Nastaleeq.ttf in desired folder. At its Startup, Symbian OS loads all font files placed in above mentioned folders.

2. The following piece of code gives an overview of how to use the above font for drawing on the screen:

```
CFont *iFont;

_LIT(KMyFontName,"Nafees Nastaleeq");

TFontSpec myFontSpec(KMyFontName, fontHeight);

aScreenDevice->GetNearestFontToDesignHeightInTwips(iFont, myFontSpec);

aGc.UseFont(iFont);
```

7.2.1.2 Numeric Keyboard Definition and Installation

Definition of on-screen keyboard/keypad requires following essentials tasks:

1. Definition of keyboard layout in a text file.
2. Measurement of space available on the screen for laying out the keypad
3. Layout the keypad in the local language

In SMSLocalized Application, we have done following for Urdu language; similar can be done for other languages.

- **UrduMultitappingKeyboard.loc** contains layout of keyboard characters for each numeric key. Following code snippets shows character layout for keys 2 and 3.

```
#define STR_EPtiKey2 "ب پ ت ٹ ث ۲"

#define STR_EPtiKey3 " ا آ ء ۳"
```

- **UrduMultitappingKeyboard.rss** contains resource ids for keyboard character strings for each key defined in the file **UrduMultitappingKeyboard.loc**. Using resource Ids, character strings can be directly accessed in the code. Given below code snippets shows character strings resource ids for keys 2 and 3.

```
RESOURCE TBUF local_eptikey_2

{

            buf = STR_EPtiKey2;

}

RESOURCE TBUF local_eptikey_3

{

            buf = STR_EPtiKey3;

}
```

- **CMultitapKeypad** class defines all the functions required to load keyboard character strings and to draw the keyboard on the screen. To draw the keypad, following functions should be used:
  - Call the function SetKeypadCharacteristics to define keypad attributes.

SetKeypadCharacteristics(TBufC<512> lang, TBufC<512> fntName, Tint keypadType, TReal origX, TReal origY, TReal keypadW, TReal keypadH)

Description of parameters:

lang: Name of the language e.g. Arabic, Urdu, Khmer,

origX:

  - Call the function DrawKeypad to display the keypad on the screen.

DrawKeypad(CGraphicsDevice* aScreenDevice, CWindowGc& aGc)

Description of parameters:

aScreenDevice: Name

aGc:

- Following is the code of important functions from the class CMultitapKeypad, along with necessary comments.

```
//Set the keypad characterisitcs. External classes call this function to set keypad attributes before it

// is drawn on the screen.

void CMultitapKeypad::SetKeypadCharacteristics(TBufC<512> lang, TBufC<512> fntName,

                    TInt keypadType, TReal origX, TReal origY, TReal keypadW, TReal keypadH)

          {

          keypadWidth = keypadW;

          rowWidth = keypadW;

          keypadHeight = keypadH;

          keypadOriginX = origX;

          keypadOriginY = origY;

          language = lang;

          fontName = fntName;

          }

//Draws the keypad on the screen

void CMultitapKeypad::DrawKeypad(CGraphicsDevice* aScreenDevice, CWindowGc& aGc)

          {

          //Performs measurements so that keypad can be properly displayed on the screen.

          rowHeight = keypadHeight / numberOfRows;

          if (rowHeight < minRowHeight)

                    {

                    rowHeight = minRowHeight;

                    }


          columnWidth = rowWidth / numberOfColumns;

          TInt verticalMargin = 5;
```

```
	TReal maxFontHeight = rowHeight;
	TReal fontAspectRatio = 0.5;
	TInt maxNumberOfCharOnKeymap = 8;
	//Performs measurements of the font for proper character displays on the keypad.
	TReal maxFontWidth = columnWidth / (maxNumberOfCharOnKeymap + 1);
	TReal expectedFontHeight = maxFontWidth / fontAspectRatio;
	TInt allowedFontHeight =
				(expectedFontHeight > maxFontHeight ? maxFontHeight
							: expectedFontHeight) - verticalMargin;
	TInt fontHeight = aScreenDevice->VerticalPixelsToTwips(allowedFontHeight);
	TFontSpec myFontSpec(fontName, fontHeight);
	aScreenDevice->GetNearestFontToDesignHeightInTwips(iFont, myFontSpec);
	aGc.UseFont(iFont);
	LoadKeypad();
	DrawKeypadLayout(aGc);
	PrintKeypadKeys(aGc);
	aGc.DiscardFont();
	aScreenDevice->ReleaseFont(iFont);
	}
//Loads keypad from the resources (keypad configuration files)
void CMultitapKeypad::LoadKeypad()
	{
	TInt resourceIds[4][3] =
		{
					LOCAL_EPTIKEY_1,
                                        LOCAL_EPTIKEY_2,
                                        LOCAL_EPTIKEY_3,
                                        LOCAL_EPTIKEY_4,
                                        LOCAL_EPTIKEY_5,
                                        LOCAL_EPTIKEY_6,
                                        LOCAL_EPTIKEY_7,
                                        LOCAL_EPTIKEY_8,
                                        LOCAL_EPTIKEY_9,
                                        LOCAL_EPTIKEY_10,
                                        LOCAL_EPTIKEY_0,
                                        LOCAL_EPTIKEY_11
                    };
          for (int i = 0; i < 4; i++)
                    for (int j = 0; j < 3; j++)
                              {
                              keyboardKeys[i][j] = (StringLoader::LoadL(resourceIds[i][j]))->Des();
                              }
          }
//Prints keypad keys i.e. which key should be displayed where is done by this function.
//Characters are drawn using the selected font.
void CMultitapKeypad::PrintKeypadKeys(CWindowGc &aGc)
          {
          TInt baseline = 0;
          TInt margin = 0;
          for (int i = 0; i < 4; i++)
```

```
                    {
                    for (int j = 0; j < 3; j++)
                              {
                              keyboxes[i][j]->SetRect(keypadOriginX + (j) * columnWidth,
                                                  keypadOriginY + (i) * rowHeight, keypadOriginX + (j + 1)
                                                            * columnWidth, keypadOriginY + (i + 1) *
rowHeight);
                              baseline = keyboxes[i][j]->Height() / 2 + iFont->AscentInPixels()
                                                  / 3;
                              aGc.DrawText(keyboardKeys[i][j], *keyboxes[i][j], baseline,
                                                  CGraphicsContext::ECenter, margin);
                              }
                    }
          }
//Draws the keypad layout which includes horizontal and veritical lines that separates characters
// of each key.
void CMultitapKeypad::DrawKeypadLayout(CWindowGc &aGc)
          {
          aGc.DrawRect(TRect(keypadOriginX, keypadOriginY, keypadOriginX
                              + keypadWidth, keypadOriginY + keypadHeight));
          //Draw horizontal lines
          aGc.DrawLine(TPoint(keypadOriginX, keypadOriginY + rowHeight), TPoint(
                              keypadWidth, keypadOriginY + rowHeight));
          aGc.DrawLine(TPoint(keypadOriginX, keypadOriginY + 2 * rowHeight), TPoint(
                              keypadWidth, keypadOriginY + 2 * rowHeight));
```

```
aGc.DrawLine(TPoint(keypadOriginX, keypadOriginY + 3 * rowHeight), TPoint(

                keypadWidth, keypadOriginY + 3 * rowHeight));

//Draw vertical lines

aGc.DrawLine(TPoint(keypadOriginX + columnWidth, keypadOriginY), TPoint(

                keypadOriginX + columnWidth, keypadOriginY + keypadHeight));

aGc.DrawLine(TPoint(keypadOriginX + 2 * columnWidth, keypadOriginY),

                TPoint(keypadOriginX + 2 * columnWidth, keypadOriginY

                                + keypadHeight));

}
```

The final keypad layout of application is shown in Figure 7.3

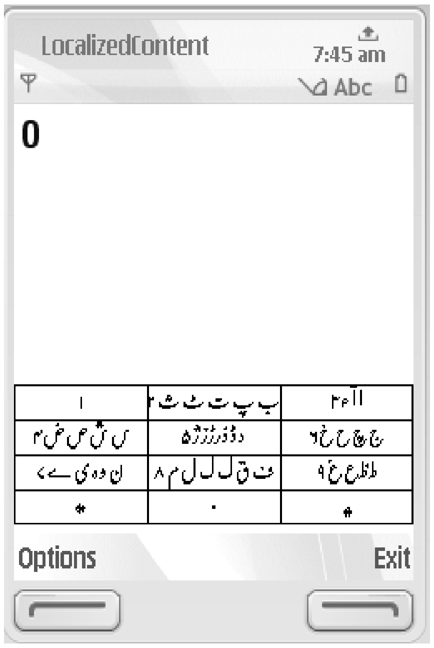
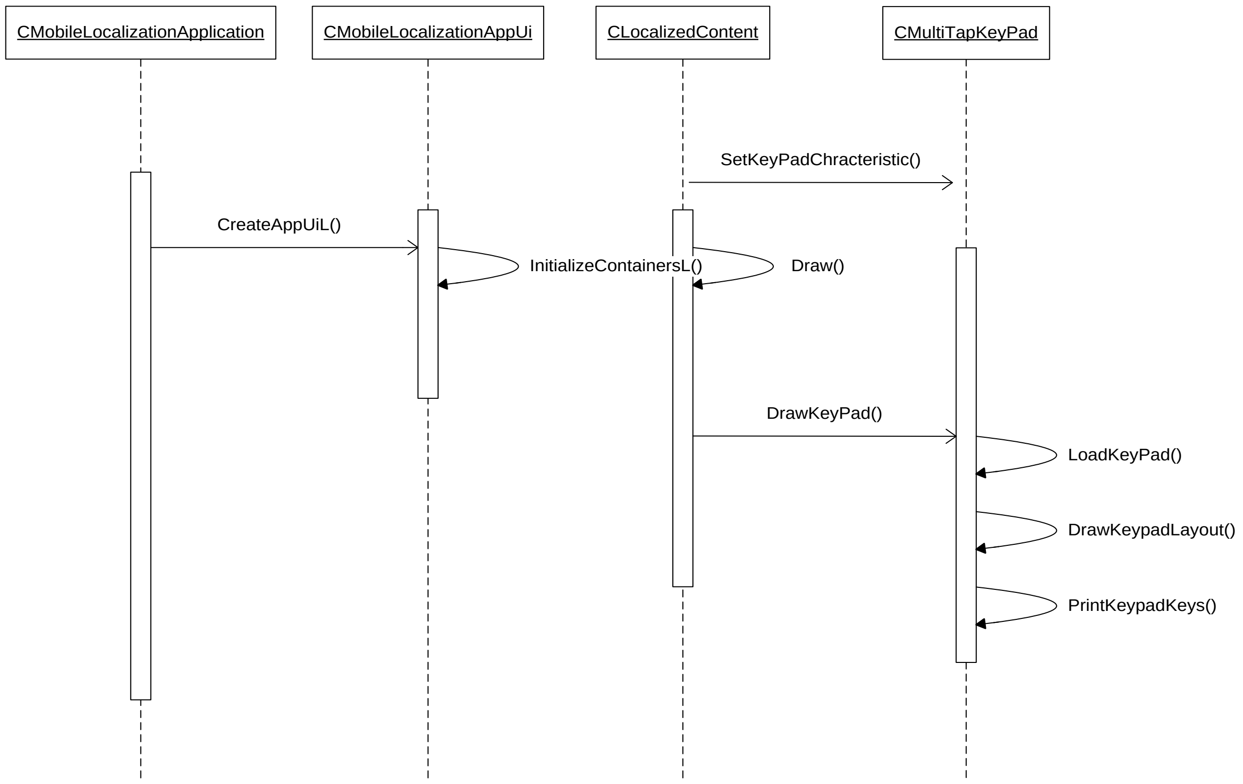


Figure 7.3: Custom keyboard

7.2.1.3 *Sequence Diagram Keyboard Load*



7.2.1.4 *Numeric Keymap Definition and Installation*

In Symbian, PtiEngine APIs provide low level text input functionality. PtiEngine APIs provides methods for querying and activating installed input languages, and performing text input functions. Additionally, PtiEngine API supports predictive text input functionality.

Building on functionality provided by PtiEngine APIs, the following tasks need to be done to define keymap:

1. A keymap is defined in a text file, which contains a Unicode keymap for each of the keys on the numeric keypad. For instance, on a typical numeric keypad phone, there are twelve keys: 0-9, # and *. Each of these keys has an associated keymap and each character in the keymap can be input by pressing relevant keys in a predefined number of times. For instance, on a default English keymap, character ‘C’ can be entered by pressing the numeric key ‘2’ three times in a row.
2. The keymap defined in the step above, is loaded and passed to PtiEngine API. PtiEngine APIs consumes this keymap as a new keymap in place of its default keymap. Therefore, each key press, after replacing default keymap with the new one, generates Unicode according to the new keymap.

In SMSLocalized Application, we have done following for Urdu language; similar can be done for other languages.

- **UrduMultitappingKeyMap.loc** is name of the file that contains keymap for Urdu language in Arabic script. Following code snippets from **UrduMultitappingKeyMap.loc** file, shows definition of keymap for numeric keys 2 and 3.

```
#define STR_EPtiKeyMap2
<0x0628><0x067E><0x062A><0x0679><0x062B><0x06C3><0x06F2>

#define STR_EPtiKeyMap3
<0x0627><0x0622><0x0624><0x0626><0x06D3><0x0621><0x06F3>
```

- **UrduMultitappingKeyboard.rss** is name of the file that refers to UrduMultitappingKeyboard.loc file. Resource Ids defined in this file can directly be used in application code. Following code snippets shows a definition of resource ids for numeric keys 2 and 3.

```
RESOURCE TBUF local_eptikeymap_2

{          buf = STR_EPtiKeyMap2;
}

RESOURCE TBUF local_eptikeymap_3

{

           buf = STR_EPtiKeyMap3;

}
```

- Following code snippets is the piece of code (from the class **CMultitapLocalizedPtiEngine**) that loads keymap the resource file and pass it on to CPtiEngine class object (PtiEngine API) for the keys 2 and 3. Execution of this step ensures that keymap of each key on numeric keyboard has been replaced by new Unicode characters of Urdu language for this specific instance of CPtiEngine.

```
iPtiEngine->SetExternalKeyMapL(EPtiEngineMultitapping, EPtiKey2,
*(StringLoader::LoadL(LOCAL_EPTIKEYMAP_2)), textCase);

                 iPtiEngine->SetExternalKeyMapL(EPtiEngineMultitapping, EPtiKey3,
*(StringLoader::LoadL(LOCAL_EPTIKEYMAP_3)), textCase);
```

- **Capturing Characters:** Each of the keys on the numeric keypad has a unique code. As soon as a key is pressed, the key code is passed to CPtiEngine class object. Following piece of code, from the class **CLocalizedContent, appends the EPtiKey2 (standard name of key '2' of the numeric keypad on Symbian platform) to iEngine (CPtiEngine class object).**

```
        case 50:{
                        iEngine->AppendKeyPress(EPtiKey2);
                        break;
                        }
```

- At any stage, current string of characters can be obtained from CPtiEngine class object as shown in line of code below:

```
TPtrC currentWord1 = iEngine->CurrentWord();
```

In the above steps, we have learnt how to define, load, and install keymap and then how to capture characters and words.

Low Level Key Events Handling

Key events are system events that originate from the keyboard driver. These are generated when the user presses one or more keys on the mobile device. The application framework then delivers this key event to the application that is in focus. More specifically the key board driver passes such events to the window server which in turn offers them to one or more of the controls of the application that is in the foreground. Symbian OS S60 applications respond to key events through the OfferKeyEventL() function.

The OfferKeyEventL() function takes as arguments two parameters: (i) a key event object and (ii) a key event type object. The TKeyEvent structure represents key events in Symbian S60. It encompasses the following details about the event being offered.

```
struct TKeyEvent
        {
                TUint iCode;
                TInt iScanCode;
                TUint iModifiers;
                TInt iRepeats;
        };
```

iCode represents the character code generated by the key event. For example, when numeric key 2 on the mobile phone is pressed, the iCode contains value 50 (corresponding to the ASCII code for decimal 2).

iScanCode contains the scan code of the key that caused the event. Standard scan codes are defined in TStdScanCode.

A key event can be of three types, defined by the TEventCode parameter in the OfferKeyEventL() function. These are *EEventKey*, *EEventKeyUp* and *EEvenetKeyDown*. The sequence of events generated by a single key press is *EEventKeyDown*, *EEventKey* and *EEventKeyUp*. The first event indicates that a key has been pressed down. The second event indicates that a character has been received from the keyboard. The third event is sent when the button has been released. These events are described in the iCode and iScanCode members of TKeyEvent. For example, the iCode value for *EEventKeyDown* and *EEventKeyUp* is zero.

7.2.2 Sending/Receiving Messages

Active objects are the Symbian OS solution to the problem of dealing with asynchronous tasks operating in parallel using only a single thread. Each application executes in its own thread. An active object is responsible for issuing requests and handling the completion of requests. Active objects are handled using the Active Scheduler, which is responsible for devising the order in which events are handled based on the priority of individual active objects (it should be noted that the Active Scheduler does not implement round-robin scheduling of requests at the same priority and so special attention should be given to the allocation of priorities to individual active objects). Each thread can have one active scheduler which may have one or more active objects. The CActiveScheduler class implements the active scheduler. It controls the handling of asynchronous events (active objects), by ordering (scheduling) the active object requests. The scheduler loops through the list looking for active objects that have completed. If it finds an active object that has completed it calls RunL(). CActive implements the active object by encapsulating the issuing of a request to an asynchronous method and handling the completion of that request. Three virtual functions provided by CActive are implemented in the derived class:

- RunL() is called by the active scheduler on completion of a function that has been activated using SetActive(); the SetActive() function indicates that the active object has issued a request
- DoCancel() implements the cancellation of any outstanding requests
- RunError() handles leaves that occur in RunL(), giving the active object the chance to handle its own errors and perform any cleanup.

RunL() and DoCancel() must be implemented before the code can be compiled.

7.2.2.1 Communication Using Sockets

An application can communicate with remote phones over sockets. The APIs are more complex than serial communications but the application can also exercise finer control over the connection, configuring settings that are specific to the particular transport.

The socket server is a standard Symbian OS server that supports a range of protocols via plug-in protocol modules. These DLLs can be identified by the PRT suffix. Both the host and the client create a session to the socket server with RSocketServ. The socket-based communications which are described below are Connection-based. One phone – the host or server – accepts an incoming connection from the second phone – the client. The general process of establishing a socket-to-socket connection is similar to the Berkeley sockets mechanism:

1. The host opens a listening socket and binds it to a local port.
2. The host listens for incoming connections on the listening socket. It calls RSocket::Listen(), passing in a queue size, which specifies the number of connections that it will accept on that port.
3. The host opens a blank socket which is not yet connected to a remote socket and passes it to the listening socket via RSocket::Accept().The host waits for a client to connect.
4. The client opens a connecting socket with the host phone's address, protocol and port number. The format of these values depends on the type of connection. For a Bluetooth phone, the address is the remote phone's 48-bit identifier; the protocol is L2CAP or RFCOMM; and the port is an L2CAP or RFCOMM channel.
5. The client calls RSocket::Connect() and waits for the host to accept the connection.
6. When the host detects an incoming connection, its accept operation completes and it establishes a connection between the connecting client and the blank socket. The blank socket, which may now be referred to as the accept socket or the data socket; can now exchange data with the remote phone.
7. On the client side, the connect operation completes. The connect socket can now exchange data with the accept socket on the host.

7.2.2.2 Messaging Using MTM API

The messaging application on a Symbian OS phone gives the user access to messages that are owned by the message server. SMS, MMS and email are all examples of messages which are managed by the server. The message server is a Symbian OS server, so applications communicate with it via a session.

The messaging framework supports a diverse range of message types with a set of plug-in DLLs called Message Type Modules (MTMs.) Each message type has its own UID. For example, KUidMsgTypeSMS, which is defined in smut.h, identifies short messages, and KUidMsgType-Multimedia, which is defined in mmsconst.h, identifies multimedia messages.

To send a message, an application connects to the Send As server with an instance of RSendAs. It then uses RSendAsMessage to construct an individual message. Once constructed, the message can be sent to a remote phone or saved to the Drafts folder for later processing. Figure 7.4 illustrates the sequence of steps required for sending a message.

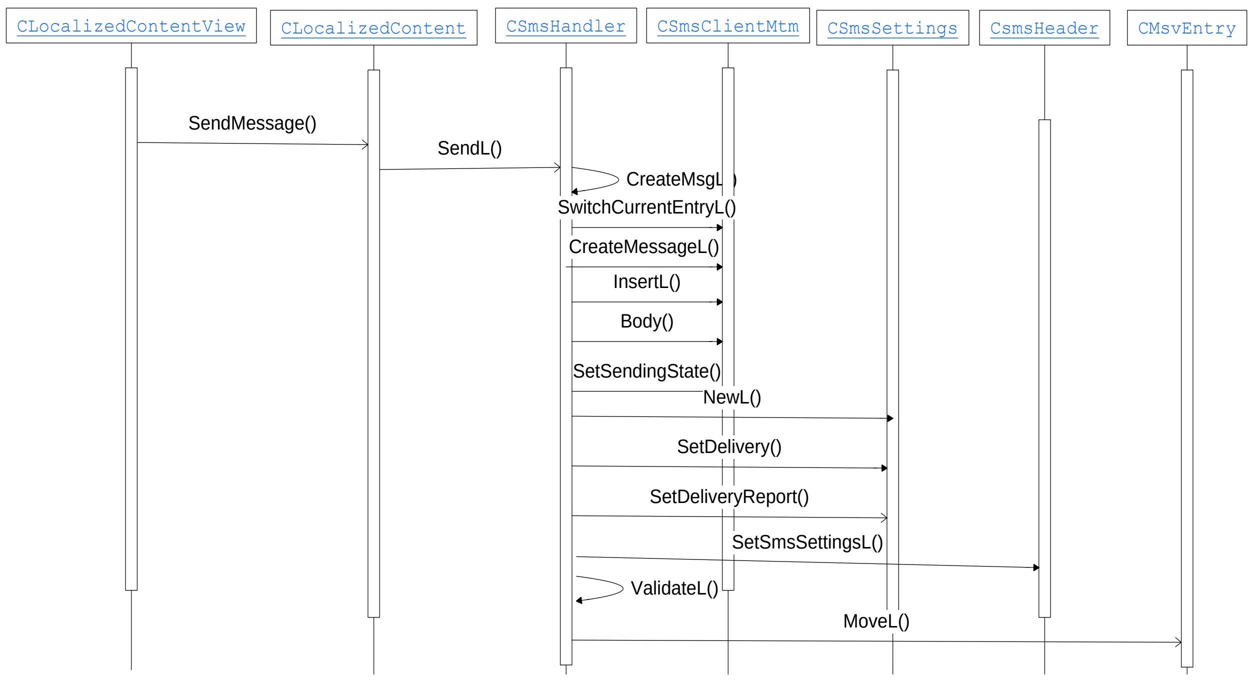


Figure 7.4: SMS send sequence diagram

7.3 Interfacing with Pango

As said earlier, we can capture the Unicode character strings entered by the user. These strings are then passed to pango for rendering in desired script. A class CPangoInterface has been developed in the application to provide interfacing with Pango. CPangoInterface set values of cairo format, display mode, font family name, font size, font weight and language. To draw text a function on top of Pango has been defined which takes as input the unicodeString and returns the rendered text in bitmap format.

The following line of code shows signatures of a function used for interfacing with Pango:

```
void CPangoInterface::draw_text_cairopango(TBuf16<500> *unicodeString, CFbsBitmap* bitmap)
```

The following piece of code extracts recently entered text string Unicodes from PtiEngine and passes them to Pango for rendering in desired script:

```
TPtrC currentWord1 = iEngine->CurrentWord();

            if(iEngine->CurrentWord().Length() == 0){

                  iEngine->AppendKeyPress(EPtiKey0);

                  iEngine->AppendKeyPress(EPtiKey0);

                  //return keyResponse;
```

```
                }

                TBuf16<1000> tempBuf = currentWord1;

                //tempBuf.Append(currentWord1);

                pangoInterface.draw_text_cairopango(tempBuf, pangoBitmap);

                InsertPangoImageL(0, pangoBitmap);

                return keyResponse;

                }
```

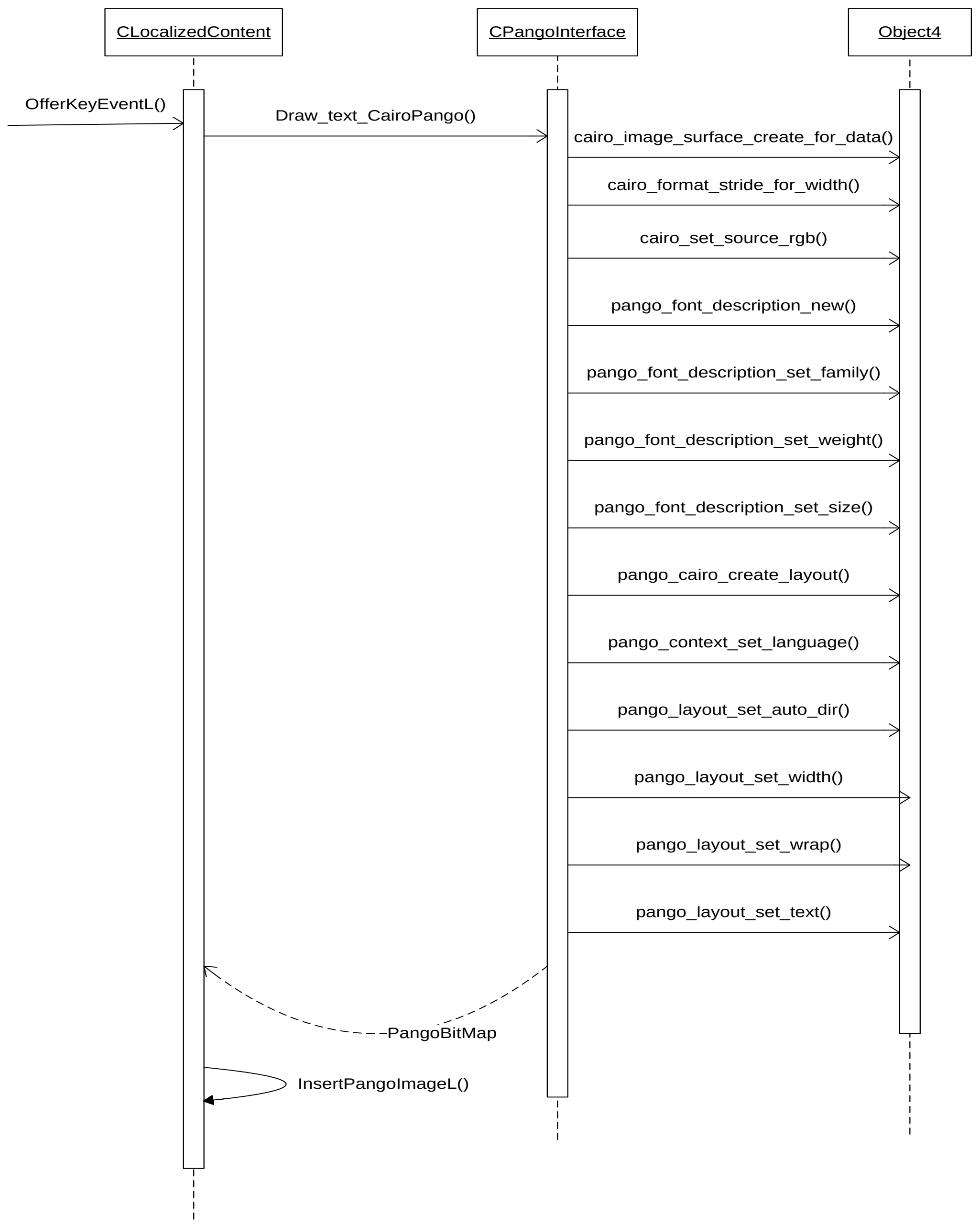
Rendered text returned in bitmap format by Pango is inserted into the rich text editor of the application as can be seen in the following piece of code:

```
void CLocalizedContent::InsertPangoImageL(TInt aPos, CFbsBitmap *aBitmap)

        {

        if (aBitmap != NULL)

                {

                CPangoImage* img;

                img = new (ELeave) CPangoImage(TSize(1800, 2260), *aBitmap);

                CleanupStack::PushL(img);

                TPictureHeader header;

                header.iPicture = TSwizzle<CPicture> (img);

                iRichText1->RichText()->InsertL(0, header);

                CPicture *picture = iRichText1->RichText()->PictureHandleL(1);

                iRichText1->RichText()->DropPictureOwnership(1);

                if (picture != NULL)

                        {

                        delete picture;

                        if (prevBitmap != NULL)
```

```
                    {
                    delete prevBitmap;
                    }
                //iNewMessageRichText->RichText()->Reset();
                //iNewMessageRichText->HandleTextChangedL();
                }
            iRichText1->HandleTextChangedL();
            iRichText1->PictureFormatChangedL();
            CleanupStack::Pop();
            prevBitmap = aBitmap;
            }
        }
```

7.3.1 Character Rendering by Pango

The following sequence diagram shows steps executed during the character rending process by Pango


CLocalizedContent
CPangoInterface
Object4
OfferKeyEventL()
Draw_text_CairoPango()
cairo_image_surface_create_for_data()
cairo_format_stride_for_width()
cairo_set_source_rgb()
pango_font_description_new()
pango_font_description_set_family()
pango_font_description_set_weight()
pango_font_description_set_size()
pango_cairo_create_layout()
pango_context_set_language()
pango_layout_set_auto_dir()
pango_layout_set_width()
pango_layout_set_wrap()
pango_layout_set_text()
PangoBitMap
InsertPangoImageL()

7.4 Steps to Create SMSLocalized Application

Following steps are required to create SMSLocalized application using PangoCairo library.

1. Create new workspace
2. Add and Build PangoCairo Project
    a. Import PangoCairo project by Solution Explorer->Import

b. Select Project Type General ->Existing Projects into Workspace as shown in Figure 7.5.

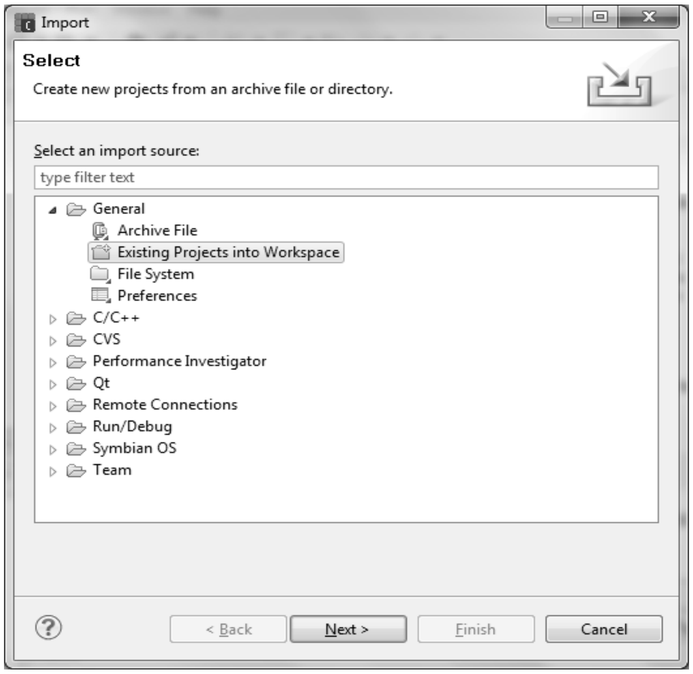


Figure 7.5: Import Project

c. Browse project from containing directory as shown in Figure 7.6

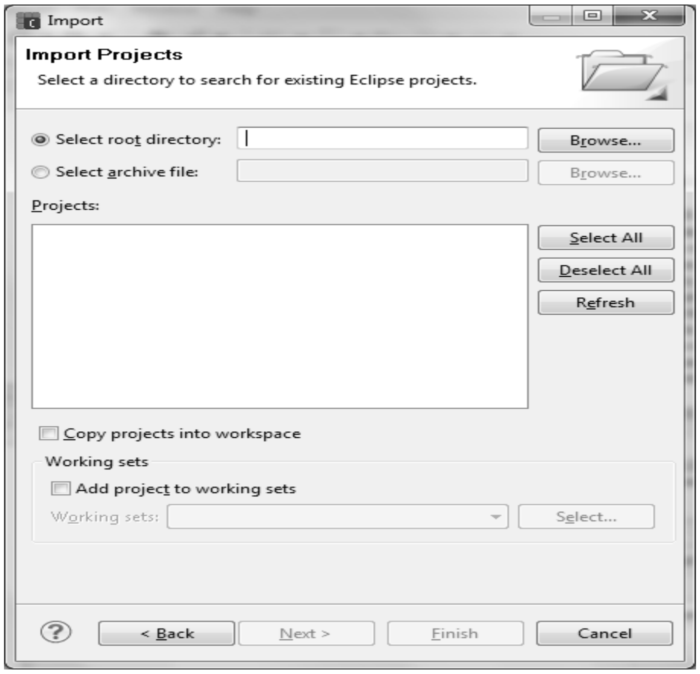
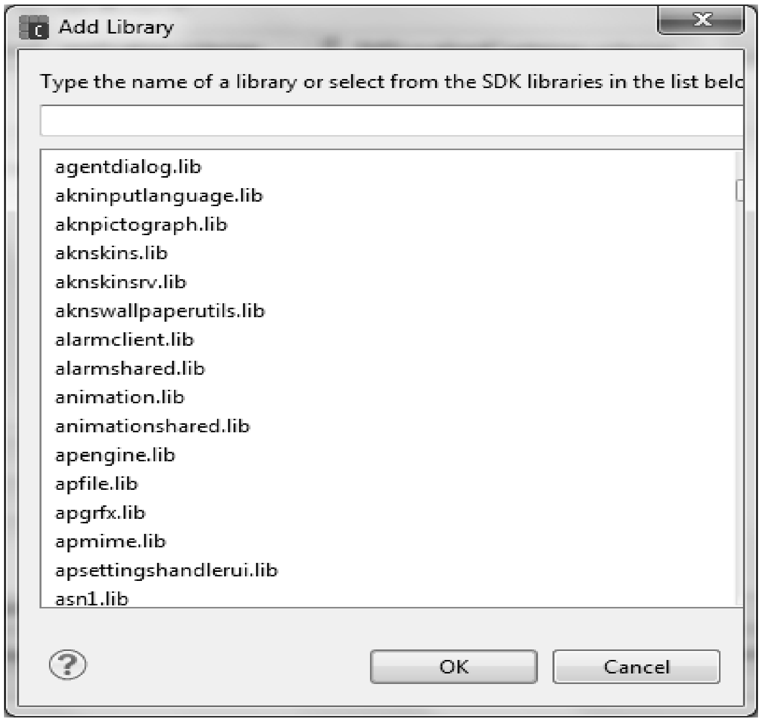


Figure 7.6: Import Project

d. Build project

3. Create New S60 project of type "GUI Application with UI Designer". Project creation wizard and project properties are already discussed in previous chapter.

4. Open mmp file of SMSLocalized project and open Libraries tab
    a. Go to Libraries listbox and click on Add button.
    b. Add library dialogue appears as shown in Figure 7.7.

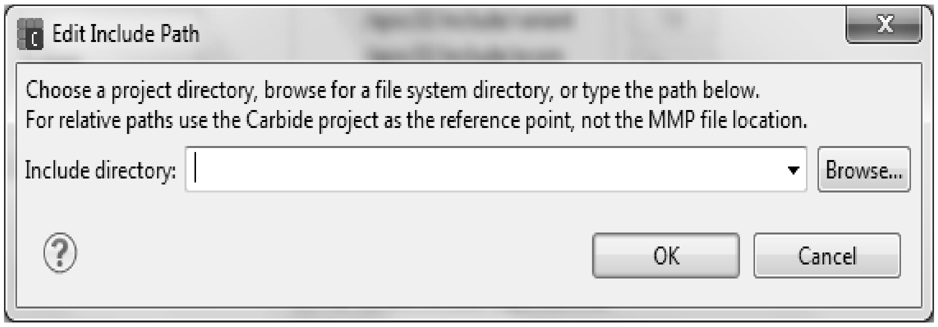


Type the name of a library or select from the SDK libraries in the list belo

agentdialog.lib
akninputlanguage.lib
aknpictograph.lib
aknskins.lib
aknskinsrv.lib
aknswallpaperutils.lib
alarmclient.lib
alarmshared.lib
animation.lib
animationshared.lib
apengine.lib
apfile.lib
apgrfx.lib
apmime.lib
apsettingshandlerui.lib
asn1.lib

Figure 7.7: Add Library

    c. Select and add cairo.lib, pangocairo.lib, glib-missing.lib, fontconfig.lib, freetype.lib, libexpat.lib, libglib.lib, libpng.lib one by one.
5. Open mmp file of SMSLocalized project and click Options tab
    a. Modify compiler settings click on System includes.
    b. Click on add button , Edit include path windows will appear as shown in Figure 7.8



Choose a project directory, browse for a file system directory, or type the path below.
For relative paths use the Carbide project as the reference point, not the MMP file location.
Include directory:

Figure 7.8: Edit Include Path Dialogue

    c. Browse Cairo folder from your S60/epoch32/include.
    d. Similraly add expat, fontconfig, freetype, freetype/config, pango, pixman,png, stdapis, stdapis/glib-2.0
6. Open .pkg file and add up the following lines to include pangocairo on device.

```
"$(EPOCROOT)Epoc32\release\gcce\udeb\glib-missing.dll"  -
"!:\sys\bin\glib-missing.dll"

"$(EPOCROOT)Epoc32\release\gcce\udeb\libexpat.dll"      -
"!:\sys\bin\libexpat.dll"

"$(EPOCROOT)Epoc32\release\gcce\udeb\freetype.dll"      -
"!:\sys\bin\freetype.dll"

"$(EPOCROOT)Epoc32\release\gcce\udeb\fontconfig.dll"    -
"!:\sys\bin\fontconfig.dll"

"$(EPOCROOT)Epoc32\release\gcce\udeb\libpng.dll"        -
"!:\sys\bin\libpng.dll"

"$(EPOCROOT)Epoc32\release\gcce\udeb\pixman-1.dll"      -
"!:\sys\bin\pixman-1.dll"

"$(EPOCROOT)Epoc32\release\gcce\udeb\cairo.dll"         -
"!:\sys\bin\cairo.dll"

"$(EPOCROOT)Epoc32\release\gcce\udeb\pangocairo.dll"    -
"!:\sys\bin\pangocairo.dll"

"$(EPOCROOT)Epoc32\data\c\data\romedalen.png"           -
"!:\data\romedalen.png"

"$(EPOCROOT)Epoc32\data\c\data\fontconfig\fonts.dtd"    -
"!:\data\fontconfig\fonts.dtd"

"$(EPOCROOT)Epoc32\data\c\data\fontconfig\fonts.conf"   -
"!:\data\fontconfig\fonts.conf"

"$(EPOCROOT)Epoc32\release\winscw\udeb\z\resource\fonts\Nafees
Nastaleeq.ttf" - "!:\resource\fonts\Nafees Nastaleeq.ttf"
```

7. After successful integration of PangoCairo in SMSLocalized project our next step is to design keyboard and display keyboard. Numeric keyboard definition and installation is already discussed in section 7.1
8. Open SMSLocalized file SMSLocalizedContainer.uidesign add text area for message and phone number.
9. Add menu item for send message and inbox in SMSLocalizedContainer.uidesign
10. Add Class for interfacing with PangoCairo. This class will be responsible for setting up value for font family, and font size. This class will also include the functionality to take Unicode string and return image. The details of this class are present in previous section.

7.5 SMSLocalized Application Flow

Following sequence diagram shows the core flow of code from text input by the user, through text rendering by the Pangocairo library, to text message sending:

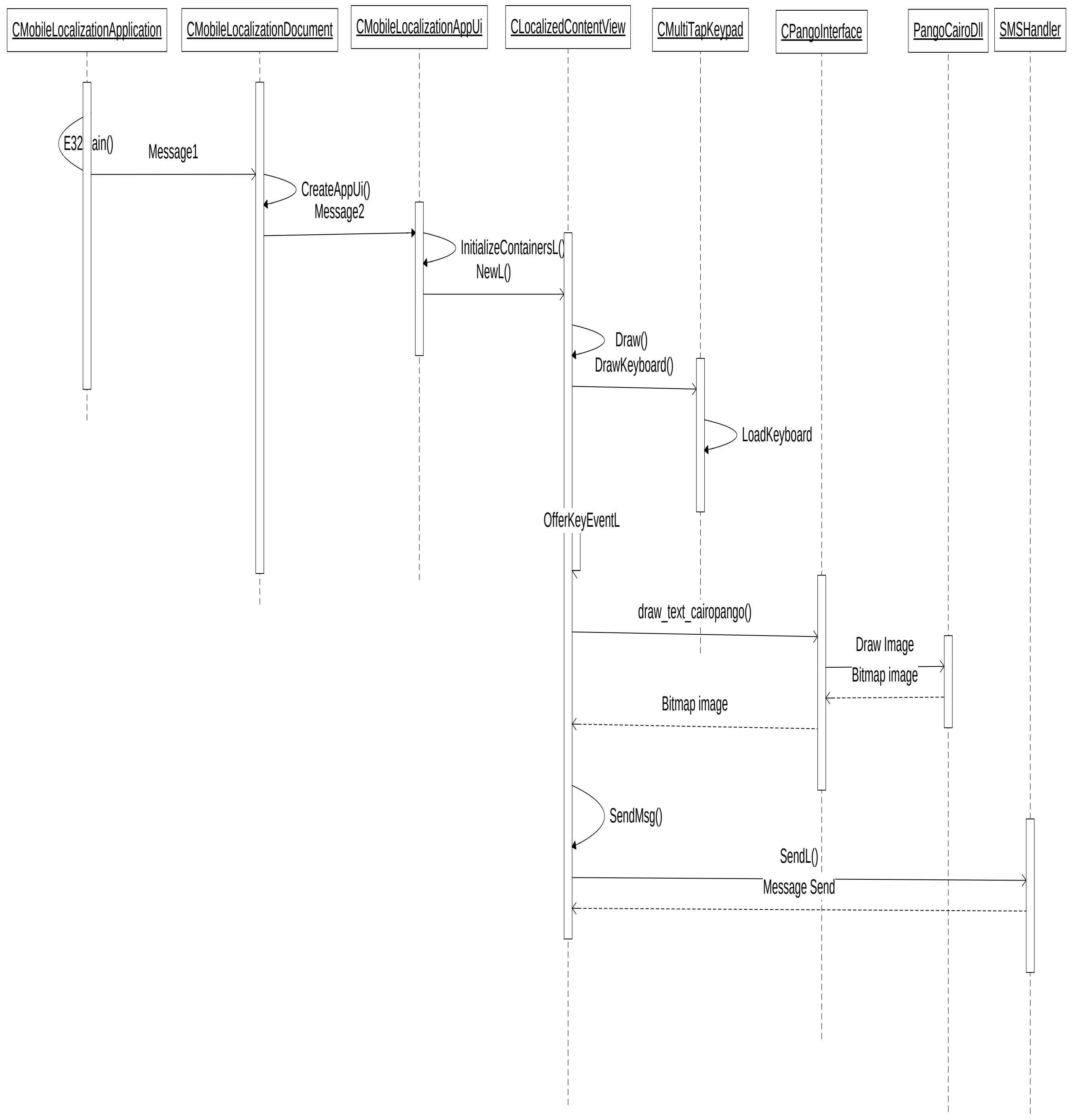


Figure 7.9: SMSLocalized Application Flow

Exercises

1. Create an application to define and load keyboard in your own local language.
2. Modify the application developed in Ex-1 to interact with Pango- Cairo Library and show the first letter of your language in RichTextEditor.
3. Modify the application developed in Ex-2 to display the character pressed from on-screen keypad.

References

[1]. Logix Mobile (2011) How does SMS Work?[online],available : http://www.logixmobile.com/faq/show.asp?catid=1&faqid=3 [accessed 20 Dec 2011].

8 Pango: A Viable Open Source Font Rendering Engine for Smartphone Platforms

Considering that Symbian has substantial market share of mobile phone market and that it is a mature operating system, it was chosen as the first platform for exploring the viability of Pango as a text layout and font rendering engine. Pango has multiple script modules. There is a basic module and then there are others specific to various international scripts such as Arabic, Indic, Khmer, and Tibetan.

There has already been a previous compilation of Pango on Symbian platform (see http://code.google.com/p/cairo-for-symbian/downloads/list). This compilation only covers basic module of Pango. Individual script-specific modules (e.g. Arabic, Khmer etc.) were not compiled. To avoid repeating the previously done work, we used this existing compilation as baseline for our Research &Development. In our R&D, we compiled the script specific modules of Pango and tested them on Symbian platform.

8.1 Pango Overview

Pango as described on the website (http://www.pango.org) : “*Pango is a library for laying out and rendering of text, with an emphasis on internationalization. Pango can be used anywhere that text layout is needed, though most of the work on Pango so far has been done in the context of the GTK+ widget toolkit. Pango forms the core of text and font handling for GTK+-2.x.*”

Pango can work with multiple font back-ends and rendering libraries as describe below (http://www.pango.org):

- Client side fonts using the FreeType and fontconfig libraries. Rendering can be with Cairo or Xft libraries, or directly to an in-memory buffer with no additional libraries.
- Native fonts on Microsoft Windows using Uniscribe for complex-text handling. Rendering can be done via Cairo or directly using the native Win32 API.
- Native fonts on MacOS X using ATSUI for complex-text handling, rendering via Cairo.

The integration of Pango with Cairo provides a complete solution with high quality text handling and graphics rendering. Combination of Pango and Cairo along with their dependencies was compiled as part of this project.

The following libraries are required for complete solution to compile and work:

- **Pango (**http://www.pango.org**)**

  Pango is font rendering and text layout engine.

- **Cairo (**http://cairographics.org **)**

Cairo is 2-D graphics library which supports multiple output devices such as X-Window, Win32, PDF, and SVG.

- **FreeType (http://www.freetype.org/)**

  Freetype is an ANSI C compliant font rasterization library. It provides access to font files of various formats and performs actual font rasterization. Font rasterization feature includes conversion of glyph outline of characters to bitmaps.

- **FontConfig (http://www.freetype.org/)**

  FontConfig allows selection of an appropriate font given certain font characteristics. It supports font configuration and font matching features. It depends on Expat XML parser. Fontconfig has two key modules:

  - *Configuration Module* builds an internal configuration from XML files
  - *Matching Module* accepts font patterns and returns the nearest matching font.

- **Glib (http://library.gnome.org/devel/glib/)**

  Glib is a utility library written in C language.

- **Pixman (http://cgit.freedesktop.org/pixman/)**

  Pixman is a pixel manipulation library for X and Cairo.

- **Expat (http://expat.sourceforge.net/)**

  Expat is an XML parser written in C.

- **libpng (http://www.libpng.org/pub/png/libpng.html)**

  Libpng is png reference library.

8.2 Pango Compilation for Symbian Platform

Pango compilation for Symbian platform can be carried out by using the number of tools and technologies. The details are provided in subsequent sections.

8.2.1 Tools and Technologies

Code Baseline

Code from following website http://code.google.com/p/cairo-for-symbian/downloads/list was taken as baseline for the project. This code covers compilation of only ’Basic’ module of Pango.

Tools

The following tools were used during development:

- Carbide C++ v2.3.0
- Symbian S60 3rd Edition Feature Pack 2 SDK v1.1.2
- GCCE Compiler was used to compile the code for target mobile device.

Compiled Code

The following are details of Pango and its dependencies which were compiled:

Table 8.1: Pango and its Dependencies

Libraries	**Version**
Pango	1.22.2
FontConfig	2.6.0
Caior	1.8.6
Expat	2.0.1
FreeType	2.3.7
Pixman	0.13.2
Lbpng	1.2.34

8.3 Deployment Platform

Compiled solutions were deployed and tested on the following platforms:

WINSCW

WINSCW is Symbian platform simulator included in Symbian S60 3rd Edition Feature Pack 2 SDK v1.1.2 for Windows Platform.

Nokia E51 (A Symbian Phone)

Following are specifications of Nokia E51—a Symbian phone:

- Symbian: v9.2 S60 v3.1 UI
- CPU: ARM 11 369 MHz Processor
- RAM: 96 MB

Changes Done in the Code

Following changes were done in the Pangocairo baseline code to compile it for Symbian platform:

Libray Name: FontConfig

Filename: fonts.conf

Following text was added so that when application is launched on a mobile device, font files located on various drives can be loaded:

```
<dir>a:\resource\fonts</dir>

<dir>b:\resource\fonts</dir>

<dir>c:\resource\fonts</dir>

<dir>d:\resource\fonts</dir>

<dir>z:\resource\fonts</dir>
```

Library Name: Pango

Filename: Pango.mmp

In project pango.mmp file, the following changes were made to include script specific modules (basic, Arabic, khmer, hangul etc.) of Pango in compilation process.

```
sourcepath	../../modules/basic

source  basic-fc.c

sourcepath	../../modules/arabic

source  arabic-fc.c

source  arabic-ot.c

sourcepath	../../modules/khmer

source  khmer-fc.c
```

```
sourcepath      ../../modules/hangul

source  hangul-fc.c

sourcepath      ../../modules/hebrew

source  hebrew-fc.c

source  hebrew-shaper.c

sourcepath      ../../modules/indic

source  indic-fc.c

source  indic-ot.c

source  indic-ot-class-tables.c

source  mprefixups.c

sourcepath      ../../modules/syriac

source  syriac-fc.c

source  syriac-ot.c

sourcepath      ../../modules/thai

source  thai-fc.c

source  thai-shaper.c

source  thai-charprop.c

sourcepath      ../../modules/tibetan

source  tibetan-fc.c
```

Filename: modules-defs-fc.c

Following changes were made in this file. These changes declare interfaces of language specific modules of Pango:

```
void _pango_arabic_ft2_script_engine_list (PangoEngineInfo **engines, gint *n_engines);

void _pango_arabic_ft2_script_engine_init (GTypeModule *module);
void _pango_arabic_ft2_script_engine_exit (void);
PangoEngine *_pango_arabic_ft2_script_engine_create (const char *id);


void _pango_khmer_ft2_script_engine_list (PangoEngineInfo **engines, gint *n_engines);
void _pango_khmer_ft2_script_engine_init (GTypeModule *module);
void _pango_khmer_ft2_script_engine_exit (void);
PangoEngine *_pango_khmer_ft2_script_engine_create (const char *id);


void _pango_hangul_ft2_script_engine_list (PangoEngineInfo **engines, gint *n_engines);
void _pango_hangul_ft2_script_engine_init (GTypeModule *module);
void _pango_hangul_ft2_script_engine_exit (void);
PangoEngine *_pango_hangul_ft2_script_engine_create (const char *id);


void _pango_hebrew_ft2_script_engine_list (PangoEngineInfo **engines, gint *n_engines);
void _pango_hebrew_ft2_script_engine_init (GTypeModule *module);
void _pango_hebrew_ft2_script_engine_exit (void);
PangoEngine *_pango_hebrew_ft2_script_engine_create (const char *id);


void _pango_indic_ft2_script_engine_list (PangoEngineInfo **engines, gint *n_engines);
void _pango_indic_ft2_script_engine_init (GTypeModule *module);
void _pango_indic_ft2_script_engine_exit (void);
PangoEngine *_pango_indic_ft2_script_engine_create (const char *id);


void _pango_syriac_ft2_script_engine_list (PangoEngineInfo **engines, gint *n_engines);
void _pango_syriac_ft2_script_engine_init (GTypeModule *module);
void _pango_syriac_ft2_script_engine_exit (void);

PangoEngine *_pango_syriac_ft2_script_engine_create (const char *id);


void _pango_thai_ft2_script_engine_list (PangoEngineInfo **engines, gint *n_engines);

void _pango_thai_ft2_script_engine_init (GTypeModule *module);

void _pango_thai_ft2_script_engine_exit (void);

PangoEngine *_pango_thai_ft2_script_engine_create (const char *id);


void _pango_tibetan_ft2_script_engine_list (PangoEngineInfo **engines, gint *n_engines);

void _pango_tibetan_ft2_script_engine_init (GTypeModule *module);

void _pango_tibetan_ft2_script_engine_exit (void);

PangoEngine *_pango_tibetan_ft2_script_engine_create (const char *id);
```

Following additional changes were made for registering the language specific modules so that they can be loaded at runtime:

```
{

   _pango_arabic_ft2_script_engine_list,

   _pango_arabic_ft2_script_engine_init,

   _pango_arabic_ft2_script_engine_exit,

   _pango_arabic_ft2_script_engine_create

},

{

   _pango_khmer_ft2_script_engine_list,

   _pango_khmer_ft2_script_engine_init,
   _pango_khmer_ft2_script_engine_exit,
   _pango_khmer_ft2_script_engine_create
},
{
   _pango_hangul_ft2_script_engine_list,
   _pango_hangul_ft2_script_engine_init,
   _pango_hangul_ft2_script_engine_exit,
   _pango_hangul_ft2_script_engine_create
},
{
   _pango_hebrew_ft2_script_engine_list,
   _pango_hebrew_ft2_script_engine_init,
   _pango_hebrew_ft2_script_engine_exit,
   _pango_hebrew_ft2_script_engine_create
},

{
   _pango_indic_ft2_script_engine_list,
   _pango_indic_ft2_script_engine_init,
   _pango_indic_ft2_script_engine_exit,
   _pango_indic_ft2_script_engine_create
},
{
   _pango_syriac_ft2_script_engine_list,
   _pango_syriac_ft2_script_engine_init,
   _pango_syriac_ft2_script_engine_exit,

   _pango_syriac_ft2_script_engine_create

},

{

   _pango_thai_ft2_script_engine_list,

   _pango_thai_ft2_script_engine_init,

   _pango_thai_ft2_script_engine_exit,

   _pango_thai_ft2_script_engine_create

},

{

   _pango_tibetan_ft2_script_engine_list,

   _pango_tibetan_ft2_script_engine_init,

   _pango_tibetan_ft2_script_engine_exit,

   _pango_tibetan_ft2_script_engine_create

},
```

Filename: Pango-engine.h

Following macro declarations were added for language specific modules:

```
#define PANGO_MODULE_PREFIX_ARABIC        _pango_arabic_ft2

#define PANGO_MODULE_PREFIX_KHMER         _pango_khmer_ft2



#define PANGO_MODULE_PREFIX_HANGUL        _pango_hangul_ft2

#define PANGO_MODULE_PREFIX_HEBREW        _pango_hebrew_ft2

#define PANGO_MODULE_PREFIX_INDIC         _pango_indic_ft2

#define PANGO_MODULE_PREFIX_SYRIAC        _pango_syriac_ft2

#define PANGO_MODULE_PREFIX_THAI_pango_thai_ft2
```

```
#define PANGO_MODULE_PREFIX_TIBETAN        _pango_tibetan_ft2
```

Following are macro definitions for various language modules:

```
#ifdef PANGO_MODULE_PREFIX_ARABIC

#define PANGO_MODULE_ENTRY_ARABIC(func)
_PANGO_MODULE_ENTRY_ARABIC2(PANGO_MODULE_PREFIX_ARABIC,func)

#define _PANGO_MODULE_ENTRY_ARABIC2(prefix,func)
_PANGO_MODULE_ENTRY_ARABIC3(prefix,func)

#define _PANGO_MODULE_ENTRY_ARABIC3(prefix,func) prefix##_script_engine_##func

#endif

#ifdef PANGO_MODULE_PREFIX_HANGUL

#define PANGO_MODULE_ENTRY_HANGUL(func)
_PANGO_MODULE_ENTRY_HANGUL2(PANGO_MODULE_PREFIX_HANGUL,func)

#define _PANGO_MODULE_ENTRY_HANGUL2(prefix,func)
_PANGO_MODULE_ENTRY_HANGUL3(prefix,func)

#define _PANGO_MODULE_ENTRY_HANGUL3(prefix,func) prefix##_script_engine_##func

#endif

#ifdef PANGO_MODULE_PREFIX_HEBREW

#define PANGO_MODULE_ENTRY_HEBREW(func)
_PANGO_MODULE_ENTRY_HEBREW2(PANGO_MODULE_PREFIX_HEBREW,func)

#define _PANGO_MODULE_ENTRY_HEBREW2(prefix,func)
_PANGO_MODULE_ENTRY_HEBREW3(prefix,func)

#define _PANGO_MODULE_ENTRY_HEBREW3(prefix,func) prefix##_script_engine_##func

#endif

#ifdef PANGO_MODULE_PREFIX_INDIC

#define PANGO_MODULE_ENTRY_INDIC(func)
_PANGO_MODULE_ENTRY_INDIC2(PANGO_MODULE_PREFIX_INDIC,func)

#define _PANGO_MODULE_ENTRY_INDIC2(prefix,func) _PANGO_MODULE_ENTRY_INDIC3(prefix,func)
```

```
#define _PANGO_MODULE_ENTRY_INDIC3(prefix,func) prefix##_script_engine_##func

#endif

#ifdef PANGO_MODULE_PREFIX_KHMER

#define PANGO_MODULE_ENTRY_KHMER(func)
_PANGO_MODULE_ENTRY_KHMER2(PANGO_MODULE_PREFIX_KHMER,func)

#define _PANGO_MODULE_ENTRY_KHMER2(prefix,func)
_PANGO_MODULE_ENTRY_KHMER3(prefix,func)

#define _PANGO_MODULE_ENTRY_KHMER3(prefix,func) prefix##_script_engine_##func

#endif

#ifdef PANGO_MODULE_PREFIX_SYRIAC

#define PANGO_MODULE_ENTRY_SYRIAC(func)
_PANGO_MODULE_ENTRY_SYRIAC2(PANGO_MODULE_PREFIX_SYRIAC,func)

#define _PANGO_MODULE_ENTRY_SYRIAC2(prefix,func)
_PANGO_MODULE_ENTRY_SYRIAC3(prefix,func)

#define _PANGO_MODULE_ENTRY_SYRIAC3(prefix,func) prefix##_script_engine_##func

#endif

#ifdef PANGO_MODULE_PREFIX_THAI

#define PANGO_MODULE_ENTRY_THAI(func)
_PANGO_MODULE_ENTRY_THAI2(PANGO_MODULE_PREFIX_THAI,func)

#define _PANGO_MODULE_ENTRY_THAI2(prefix,func) _PANGO_MODULE_ENTRY_THAI3(prefix,func)

#define _PANGO_MODULE_ENTRY_THAI3(prefix,func) prefix##_script_engine_##func

#endif

#ifdef PANGO_MODULE_PREFIX_TIBETAN

#define PANGO_MODULE_ENTRY_TIBETAN(func)
_PANGO_MODULE_ENTRY_TIBETAN2(PANGO_MODULE_PREFIX_TIBETAN,func)

#define _PANGO_MODULE_ENTRY_TIBETAN2(prefix,func)
_PANGO_MODULE_ENTRY_TIBETAN3(prefix,func)

#define _PANGO_MODULE_ENTRY_TIBETAN3(prefix,func) prefix##_script_engine_##func
```

```
#endif
```

Filename: Pango-language.h

Following function has been declared as PangoApi to make it accessible externally:

```
PangoApi PangoLanguage *pango_language_from_string (const char *language);
```

Filename: Pango-language.c

Following function has been declared as PangoApi to make it accessible externally

```
PangoApi PangoLanguage *pango_language_from_string (const char *language);
```

Filename: PangocairoU.def

Following changes were made in this file

```
pango_language_from_string @ 416 NONAME
```

Pango Interface API

Following API was developed to make Pango accessible from external applications. Class containing this API lets the programmers define additional parameters such as font family name, font size, font weight, language etc.

```
void PangoSymbianInterface::draw_text_cairopango(TBuf16<500> unicodeString, CFbsBitmap&
bitmap){

        bitmap.LockHeap();

        int stride;

        unsigned char *data;

        TInt width = 200;
```

```
    TInt height = 200;

    TSize size(width, height);

    /*provides a stride value that will respect alignment requirements of the image-rendering code
    within cairo*/

    stride = cairo_format_stride_for_width(format, width);

  /* Creates an image surface for the provided pixel data*/

            cairo_surface_t* surface = cairo_image_surface_create_for_data(

                            (unsigned char*) bitmap.DataAddress(),

                            format, size.iWidth, size.iHeight, stride);

            /* contains the current state of the rendering device, including coordinates of yet to be
            drawn shapes*/

            cairo_t* cr = cairo_create(surface);


            cairo_status_t err = cairo_status(cr);

            if (err == CAIRO_STATUS_NO_MEMORY)

                            {

                            User::Leave(KErrNoMemory);

                            }

            /* Makes a copy of the current state of cr and saves it on an internal stack of saved
            states */

            cairo_save(cr);


    /*creates a new font description structure*/

    PangoFontDescription* fontDesc = pango_font_description_new();


    pango_font_description_set_family(fontDesc, fontFamilyName);
    /*specifies how light or bold the font should be*/

        pango_font_description_set_weight(fontDesc, fontWeight);

    /*sets the size of a font*/

        pango_font_description_set_size(fontDesc, fontSize * PANGO_SCALE);

        /*Creates a layout object set up to match the current transformation        and target surface of
        the Cairo context*/

        PangoLayout *layout = pango_cairo_create_layout(cr);


        PangoContext* pangoContext = pango_layout_get_context(layout);


        /* Converts RFC-3066 format language tag string to a PangoLanguage pointer */

    PangoLanguage *pangoLanguage = pango_language_from_string(language);

        pango_context_set_language(pangoContext, pangoLanguage);


    /*set the base direction for the context*/

        pango_context_set_base_dir(pangoContext, PANGO_DIRECTION_RTL);

        pango_context_set_font_description(pangoContext, fontDesc);

        pango_layout_set_font_description(layout, fontDesc);

        pango_context_load_font(pangoContext, fontDesc);

        /* Forces recomputation of any state in the PangoLayout that might depend on the layout's
context. This function should be called if you make changes to the context subsequent to creating the
layout. */

        pango_layout_context_changed(layout);

        TBuf8<5000> textUtf8;

        char* charText = "";
        CnvUtfConverter::ConvertFromUnicodeToUtf8(textUtf8, unicodeString);

        charText = (char *) User::Alloc(textUtf8.Length()+1);

        TInt index = 0;

        for(index =0 ; index < textUtf8.Length(); index++)

        {

          charText [index] = textUtf8[index];

        }

        charText[index] = 0;

        /*sets the text of layout in UTF-8 format*/

        pango_layout_set_text(layout, charText, -1);

        cairo_save(cr);

        cairo_move_to(cr, 120.0, 20.0);

        cairo_set_source_rgba (cr, 0, 0,0, 1);

        pango_cairo_update_layout(cr, layout);

    /* Draws a PangoLayoutLine in the specified cairo context.*/

        pango_cairo_show_layout_line(cr, pango_layout_get_line(layout, 0));

         cairo_restore(cr);

        g_object_unref(layout);

        pango_font_description_free(desc);


}
```

9 Conclusion and Future Research

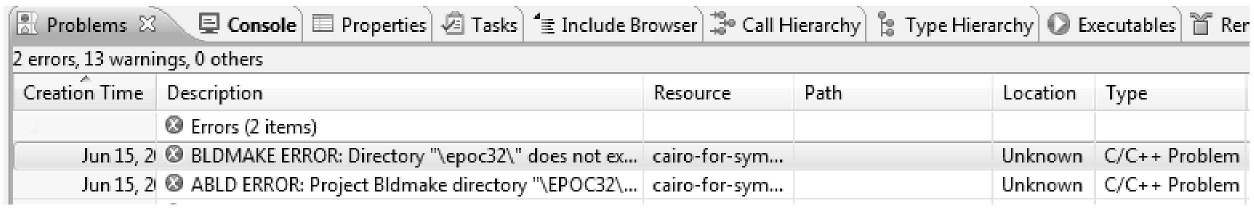
The global penetration of smart-phones is making local language support for them both urgent and significant, as an increasing number of mobile users want the devices to access local language content. However, we have learnt that smart-phones are still far from current desktops in their support for the local scripts of developing Asia. The Symbian platform, among the oldest and mature mobile platforms, does not provide complete Open Type Font (OTF) support. However, the porting of Pango script-specific modules can add OTF support to Symbian. This has been successfully achieved through our project. All of the Pango language script modules have been ported to the Symbian OS, with extensive testing carried out for Urdu and initial level of testing performed for Khmer. Through this process, we have learnt that the Arabic, Indic and Khmer language modules of Pango work well on Symbian platform. We believe that given the extensive support for international languages by Pango, is a good choice for serving as a text layout and rendering engine for smart-phone devices.

Currently, the work is underway to port Pango to Google Android, which is an open source platform and is increasingly becoming popular.

10 Troubleshooting

BLDMAKE and ABLD Error

When importing existing application and trying to build due to selection of SDK, BLDMAKE and ABLD error occurs as shown in the figure below.



2 errors, 13 warnings, 0 others

Creation Time	Description	Resource	Path	Location	Type
	Errors (2 items)				
Jun 15, 2	BLDMAKE ERROR: Directory "\epoc32\" does not ex...	cairo-for-sym...		Unknown	C/C++ Problem
Jun 15, 2	ABLD ERROR: Project Bldmake directory "\EPOC32\...	cairo-for-sym...		Unknown	C/C++ Problem

This error can be resolved by selecting the SDK and its version for project. SolutionExplorer->Properties->Carbide.c++ ->Build Configurations ->Configurations Drop down.

Appendix A: Directions for Solving Exercises

Exercise 6.1

1. Create new application named "Random" by following the project creation wizard (New -> Symbian OS C++ Project -> S60 -> GUI application with UI Designer)
2. Click on RandomContainer.uidesign
3. Drag and drop Number Editor from right hand side panel.
4. Open RandomContainer.cpp
5. Generate Random Number by following statement

   ```
   Math::Random()%(100-1+1)+1;
   ```
6. Assign generated number to number text editor by using SetNumber() function.

Exercise 6.2

1. Create new application named "MultiView" by following the project creation wizard (New -> Symbian OS C++ Project -> S60 -> GUI application with UI Designer)
2. Add a new view for Spanish by using the Left hand panel .Right click on project node(MultiView), click on New ->S60 UI Design
3. Now in your project you have two different views you can use one for English and second for Spanish.
4. Open application.uidesign and go to Languages Tab, Add Spanish into the list
5. Open MultiviewContainer.uidesign, drag TextEditor for name and DateEditor for Admission Date.
6. Repeat step 5 for Spanish view as well.
7. Open MultiViewContainer.uidesign and click on optionsMenu add new option for Spanish.
8. Open your Spanish view uidesign and click on optionsMenu add new option for English.
9. For Spanish View, Go to your data folder there will be file for Spanish string with "??" Add your Spanish strings here e.g Name can be written as nombre.
10. Go to your MultiviewContainer.uidesign optionsMenu,right click on Spanish option and select option "Handle selected Event".
11. Repeat Step 10 for your Spanish view as well.
12. Open your code of Spanish option generated by MultiviewContainer.uidesign and write the following code to activate Spanish view:

    ```
    ActivateViewL (TVwsViewId(KUidMultiViewApplication,TUid::Uid(ELocalizedSpanishViewId)));
    ```
13. Repeat Step 12 for activating English view.