

Symbian C++

ПРОГРАММИРОВАНИЕ
ДЛЯ МОБИЛЬНЫХ
ТЕЛЕФОНОВ

А.Н. Труфанов

Symbian C++

ПРОГРАММИРОВАНИЕ
ДЛЯ МОБИЛЬНЫХ
ТЕЛЕФОНОВ



Москва • Санкт-Петербург • Киев
2010

ББК 32.973.26-018.2.75

T80

УДК 681.3.07

Издательский дом “Вильямс”

Зав. редакцией *А.В. Слепцов*

По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу:
info@williamspublishing.com, <http://www.williamspublishing.com>

Труфанов, А.Н.

T80 Symbian C++. Программирование для мобильных телефонов. — М. :
ООО “И.Д. Вильямс”, 2010. — 464 с. : ил.

ISBN 978-5-8459-1629-7 (рус.)

ББК32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения Издательского дома “Вильямс”.

Copyright © 2010 by WilliamsPublishing House.

All rights reserved including the right of reproduction in whole or in part in any form.

Научно-популярное издание

Александр Николаевич Труфанов

Symbian C++

Программирование для мобильных телефонов

Литературный редактор *Е.П. Перестюк*

Верстка *Л.В. Чернокозинская*

Художественный редактор *В.Г. Павлютин*

Корректор *Л.А. Гордиенко*

Подписано в печать 17.02.2010. Формат 70х100/16.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 37,41. Уч.-изд. л. 24,0.

Тираж 1000 экз. Заказ № 0000.

Отпечатано по технологии StP

в ОАО “Печатный двор” им. А. М. Горького

197110, Санкт-Петербург, Чкаловский пр., 15.

ООО “И. Д. Вильямс”, 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1629-7 (рус.)

© Издательский дом “Вильямс”, 2010

Оглавление

Введение	11
Глава 1. Основы операционной системы Symbian	19
Глава 2. Структура проекта на Symbian C++	41
Глава 3. Работа с SDK	63
Глава 4. Интегрированная среда разработки Carbide.c++	95
Глава 5. Основы Symbian C++	124
Глава 6. Разработка приложений	211
Глава 7. Сертификация приложений	437
Приложение А. Акронимы и сокращения	450
Приложение Б. Справочные материалы	452
Предметный указатель	458

Содержание

Введение	11
О чем эта книга	11
Чего нет в этой книге	12
Инструменты, которые вам потребуются	13
Знания, необходимые для изучения Symbian C++	14
Как работать с книгой	15
Ресурсы для разработчика	16
Об авторе	17
Обратная связь	17
Благодарности	17
От издательского дома “Вильямс”	18
Глава 1. Основы операционной системы Symbian	19
Краткая история Symbian OS	19
Операционная система и платформа	21
Архитектура Symbian OS	23
Ядро EKA2	24
Службы операционной системы	25
Общие вспомогательные службы	27
Прикладные службы	29
Исполняемые файлы в ROM и RAM	29
Уникальные идентификаторы в Symbian OS	30
Платформа безопасности Symbian OS	30
Защищенные возможности в библиотеках	34
Идентификаторы VID и SID	34
Экранирование данных	36
Установка приложений и сертификаты	38
Глава 2. Структура проекта на Symbian C++	41
Файл bld.inf	41
ММР-файлы	44
Файлы ресурсов и локализация проекта	50
Объявление структуры ресурса	52
Объявление ресурса	54
Идентификаторы ресурсов	56
Перечисления в файлах ресурсов	57
Прочие выражения файлов ресурсов	58
Локализация и компиляция файла ресурса	58
Прочие файлы проекта	62

Глава 3. Работа с SDK	63
Выбор SDK	63
Установка SDK	64
Состав SDK	65
Выбор текущего SDK	67
Компиляторы, платформы и режимы компиляции	68
Сборка проекта	70
Заморозка проекта, def-файлы	73
Работа с эмулятором	73
Очистка проекта	80
Создание дистрибутива приложения	80
Файл PKG	81
Создание SIS-файла	91
Проблемы, часто возникающие при установке	93
Глава 4. Интегрированная среда разработки Carbide.c++	95
Немного истории	95
Инсталляция Carbide.c++ 2.x	96
Запуск, интерфейс и рабочее пространство	97
Создание и импорт существующих проектов	98
Создание нового проекта	98
Импорт существующего проекта	101
Работа с проектом	103
Навигация	103
Работа с файлами проекта	105
Панель Console	105
Работа с исходным кодом	106
Очистка и заморозка проекта	110
Сборка проекта	110
Запуск приложения в эмуляторе	111
Отладка в эмуляторе	112
Сборка SIS-пакета	115
Отладка на устройстве	116
Обновление Carbide.c++	122
Глава 5. Основы Symbian C++	124
Фундаментальные типы данных	124
Соглашение об именовании	126
Общее правило	126
Константы и макросы	126
Классы и члены классов	127
Структуры	128
Перечисления и их значения	128
Функции и аргументы	129
Обработка ошибок и исключений	130
Сбросы	130
Ловушки	131

8 Symbian C++. Программирование для мобильных телефонов

Паника	134
Макросы __ASSERT_XXX (утверждения)	136
Управление памятью: куча и стек	137
Стек очистки: CleanupStack	140
T-классы	147
C-классы, двухфазное конструирование	149
R-классы	152
M-классы, наследование	155
Дескрипторы, работа со строками	158
Классы дескрипторов. Изменяемые и неизменяемые дескрипторы	158
Базовые дескрипторы	159
Символьные дескрипторы	160
Дескрипторы-буферы TBuf и TBufC	162
Дескрипторы-указатели	167
Хранение строк в памяти кучи. Дескрипторы HBufC и RBuf	169
8- и 16-битовые дескрипторы. Кодировки	173
Выбор подходящего класса дескриптора	176
Дескрипторы как аргументы и результат функций	177
Дескрипторы-пакеты TPkg, TPkgC и TPkgBuf	179
Лексический анализатор TLex	180
L-классы	181
Динамические массивы	183
Массивы фиксированного размера	191
Активные объекты	192
Глава 6. Разработка приложений	211
Приложение Hello World на Symbian C++	211
Консоль	219
Регистрация программы в меню приложений	221
Изменение подписи пиктограммы	224
Изменение пиктограммы	226
Именованье исполняемых файлов, смена идентификаторов	230
Автостарт при запуске системы	232
Создание библиотек	234
Статически связываемые библиотеки (LIB)	234
Разделяемые динамические библиотеки (DLL)	235
Изменяемые глобальные данные в DLL	245
Работа с процессами и потоками	246
Синхронизация потоков	254
Межпоточное взаимодействие	260
Межпроцессное взаимодействие	260
Разделяемые области памяти	260
Очереди сообщений	262
Механизм уведомлений Publish & Subscribe	263
Клиент-серверная архитектура приложений	265
Общие сведения	266

Сервер	267
Представление сессии на стороне сервера	269
Представление сессии на стороне клиента	270
Запуск сервера при установке сессии	272
Остановка сервера	274
Команды, сообщения и передача данных	276
Механизм ECom	286
Общие сведения	287
Интерфейс	288
Реализация интерфейса, ECom DLL	291
Регистрация ECom DLL	294
Выбор реализаций	296
Resolver	298
Распознаватели	298
Работа со временем	300
Таймеры	303
Работа с файловой системой	312
Именованние файлов и папок	313
Сессия файлового сервера	316
Текущий каталог сессии, работа с именами файлов и каталогов	317
Приватный каталог процесса	319
Получение информации о доступных дисках и разделах	319
Создание каталогов, переименование и удаление файлов и каталогов	321
Операции с атрибутами каталогов и файлов	323
Получение списка подкаталогов и файлов в каталоге	325
Прочие полезные функции файловой сессии	328
Файловый менеджер CFileMan	329
Файлы, чтение и запись данных	338
Открытие файла	338
Режимы доступа к файлу	340
Чтение и запись данных	342
Перемещение текущей позиции	348
Прочие методы класса RFile	349
Потоки данных	350
Базовые классы потоков чтения и записи	351
Потоки чтения и записи	353
Операторы << и >>	357
Хранилища данных	359
Организация файлового хранилища	360
Создание хранилища	361
Открытие хранилища и чтение данных	364
Сжатие хранилища	365
Класс CDictionaryFileStore	365
Класс TSwizzle	366
Базы данных	367
Доступ к базе данных	367
Классы баз данных	368

10	Symbian C++. Программирование для мобильных телефонов	
	Таблицы	370
	Индексы	374
	Чтение и запись данных	375
	SQL-запросы	381
	Использование сессии сервера СУБД	387
	Транзакции	388
	Поэтапное выполнение операций	390
	Сокеты	390
	Сервер сокетов	391
	Протоколы	392
	Адреса, класс RHostResolver	396
	Работа с сокетами	398
	Подключения	403
	Сервер окон	404
	Получение уведомлений о нажатиях клавиш	407
	Рисование на экране	421
	Отображение текста	428
	Работа с изображениями	432
	Другие возможности сервера окон	435
	Глава 7. Сертификация приложений	437
	Способы сертификации	437
	Open Signed Online	441
	Покупка идентификатора издателя	442
	Open Signed Offline	443
	Резервирование идентификаторов	445
	Критерии тестирования Symbian Signed	445
	Программа сертификации Express Signed	447
	Программа сертификации Certified Signed	448
	Приложение А. Акронимы и сокращения	450
	Приложение Б. Справочные материалы	452
	Документация	452
	Ссылки	457
	Предметный указатель	458

Введение

За последние годы мир портативных компьютеров стремительно развивался: наряду с уже ставшими привычными ноутбуками, навигаторами, карманными компьютерами и коммуникаторами появились новые классы устройств: нетбуки, субноутбуки и интернет-планшеты (internet tablets, или MID — Mobile Internet Device). Очевидно, что эволюция этих устройств в самом разгаре. В конце 2008 года впервые совокупные продажи портативных компьютеров превысили продажи персональных. Успехи исследователей в таких областях науки и техники, как элементы питания, сенсорные и гибкие экраны сулят нам фантастические возможности уже в ближайшем будущем. Но незаменимыми помощниками портативные компьютеры делают не достижения в аппаратной архитектуре, а поддерживаемое ими программное обеспечение.

Смартфоны и коммуникаторы — один из наиболее распространенных видов портативных компьютеров. Эти устройства функционируют под управлением операционных систем, признанным лидером среди которых является Symbian OS. Смартфоны на базе Symbian OS производят компании Nokia, Samsung и Sony Ericsson. На начало 2009 года Symbian OS занимала 46% мирового рынка смартфонов. Свою популярность, а также любовь пользователей и разработчиков всего мира, Symbian OS заслужила благодаря ее широчайшим возможностям. Разработка приложений для платформы S60, основанной на операционной системе Symbian, может вестись на таких языках программирования, как Symbian C++, C, C++ и Python, а также при помощи технологий Qt, Java 2 Micro Edition, .Net Compact Framework, Flash Lite и WRT-виджетов. Она позволяет создавать программы, которые просто невозможно разработать для других платформ.

О чем эта книга

С помощью этой книги вы познакомитесь с архитектурой Symbian 9.x, изучите интегрированную среду разработки Carbide.c++ и язык программирования Symbian C++ — язык, на котором написана сама операционная система. Symbian C++ — наиболее мощный инструмент разработчика приложений для Symbian OS. С его помощью вы сможете получить доступ даже к тем подсистемам, которые недоступны для прочих языков программирования и технологий. Именно на Symbian C++ пишутся модули для Python и Qt, расширяются возможности J2ME, WRT-виджетов и Flash Lite. К сожалению, Symbian C++ также является и наиболее сложным средством разработки приложений для

Symbian OS. Поэтому прежде чем приступить к его освоению, я советую прочесть раздел “Знания, необходимые для изучения Symbian C++”.

Подробно рассмотрены вопросы разработки различных типов приложений и библиотек, а также использование базовых API Symbian OS. Освещены основные положения сертификации приложений для Symbian OS.

Данная книга также может использоваться для подготовки к сдаче экзамена Accredited Symbian Developer (ASD). В конце разделов, темы которых входят в учебный план ASD v2.1 (2008–2009), приводятся список тезисов для запоминания и предъявляемых к экзаменуемым требований.

Чего нет в этой книге

В этой книге рассматриваются вопросы программирования для операционной системы Symbian 9-й версии. Эта ОС используется в платформах UIQ 3, S60 3-й и 5-й редакции. Создание приложений для Symbian 9.x несколько отличается от версий 7.x и 8.x (UIQ 2 и S60 2-й редакции): в частности, в них не используется платформа безопасности, по-другому реализован автозапуск приложений, клиент-серверная архитектура, нет ряда API и т.д. Подобные отличия я постараюсь отмечать, но подробно останавливаться на них мы не будем. Это связано с тем, что устройства под управлением Symbian 7.x-8.x уже довольно давно не выпускаются.

Symbian OS была изначально спроектирована таким образом, чтобы служить ядром для платформ, разрабатываемых различными производителями устройств. В свою очередь именно платформы предоставляют разработчику подсистемы для создания пользовательского интерфейса, исходя из особенностей реализации аппаратной архитектуры конкретного производителя. Например, наиболее известными платформами на базе Symbian OS 9-й версии являются UIQ 3 и S60 3-й редакции. Приложения, использующие в своей работе подсистемы разных платформ, несовместимы. Поэтому программа для UIQ 3 не будет работать на устройстве под управлением S60 3-й или 5-й редакции без некоторых (подчас существенных) изменений. В этой книге мы изучим вопросы программирования на уровне операционной системы, не касаясь использования подсистем различных платформ. В частности, здесь не рассматриваются системы Qikon и Avkon, а значит, из этой книги вы не узнаете, как создавать всевозможные кнопки, списки и прочие элементы пользовательского интерфейса — это тема для отдельной книги. С другой стороны, нельзя переходить к изучению сервисов платформы, не усвоив материал этой книги, — он является базовым при программировании на Symbian C++ для любых платформ на основе Symbian OS. Кроме того, даже не зная особенностей Avkon или Qikon, вы сможете создавать полноценные приложения со стандартным пользовательским интерфейсом при помощи инструмента UI Designer, входящим в состав IDE Carbide.c++ 2.x. UI Designer является средством визуального проектирования пользовательского интерфейса приложений и поддерживается как в SDK

для платформы UIQ, так и SDK для S60 3-й/5-й редакций. Здесь мы не будем рассматривать работу с UI Designer, но его использование достаточно просто освоить самостоятельно.

Значительная часть API Symbian OS в данной книге не освещена. Их так много, что рассмотреть их все в одной книге просто невозможно. Кроме того, подробное описание возможностей системы является задачей справочника, входящего в состав SDK, но никак не учебника. Однако приведенного в книге материала достаточно, чтобы обеспечить формирование у читателя базовых знаний и навыков, необходимых и достаточных для дальнейшего освоения любого системного API.

В книге не рассматриваются такие интегрированные среды разработки, как Metrowerks CodeWarrior, Borland C++ Mobile Studio и Carbide.vs (последняя является дополнением к Microsoft Visual Studio 2003/2005) — развитие и поддержка этих сред прекращена авторами. Наиболее актуальной и функциональной IDE для программирования на Symbian C++ в настоящее время является Carbide.c++ от Forum Nokia. Использование именно этой среды подробно освещается в данной книге.

Инструменты, которые вам потребуются

Для того чтобы начать программировать для Symbian OS, вам потребуется следующее.

1. Компьютер с установленной на нем ОС Windows XP или Vista (XP предпочтительнее).
2. SDK для платформы, использующей Symbian 9-й версии. Рекомендую All-in-One S60 3rd Edition, Feature Pack 2 v1.1 (455 Mb)¹.
3. ActivePerl v 6.5.1.x² Необходима именно эта версия, более новые или более старые не подойдут.
4. Бесплатная интегрированная среда разработки Carbide.c++ 2.x³.
5. Приведенные выше компоненты являются обязательными. Желательно также иметь следующее.
 - Программа Nokia PC Suite⁴ для установки приложений в смартфонах.
 - Смартфон или коммуникатор под управлением S60 3-й или 5-й редакции для тестирования программ. Если у вас есть смартфон, то выберите SDK согласно версии его платформы. Если вам по какой-то причине не удастся получить подходящее устройство для тестирования, то вы сможете запускать создаваемые приложения в имеющемся в составе SDK эмуляторе.

¹ www.forum.nokia.com/Resources_and_Information/Tools/Platforms/S60_Platform_SDKs/

² <http://www.oldapps.com/Perl.php>

³ www.forum.nokia.com/Resources_and_Information/Tools/IDEs/Carbide.c++/

⁴ www.nokia.ru/support/software/nokia-pc-suite/download

Знания, необходимые для изучения Symbian C++

Данная книга рассчитана на читателя, знакомого с принципами объектно-ориентированного программирования и основами языка C++. Предполагается, что вы уже знаете:

- что такое процесс и поток;
- что такое сервис, сервер и клиент-серверная архитектура приложения;
- типы, выражения, операторы и функции языка C++;
- область видимости объектов в C++;
- объявление классов, конструкторы и деструкторы в C++;
- наследование и полиморфизм классов;
- виртуальные методы классов;
- что такое интерфейсы;
- механизм динамического выделения памяти и различия между кучей и стеком.

Все это — базовая программа изучения языка программирования C++ в любом ВУЗе. Вам совершенно не потребуются знания и опыт программирования на C++ для операционных систем Windows или Linux — Symbian OS имеет с ними довольно мало общего.

Зачастую человеку, не имеющему большого опыта в создании программ для Windows или Linux, оказывается легче освоить программирование для Symbian, чем опытному разработчику. Это во многом происходит от того, что “гуру” острее переживают свои неудачи и возможные затруднения в устранении возникающих проблем. Им сложно смириться с тем, что их знания и опыт применительно к новой системе обесцениваются. Будьте открыты чему-то новому. Если сама перспектива необходимости сменить среду разработки уже вводит вас в ступор — Symbian C++ не для вас. Запомните: программирование для мобильных устройств во многом *сложнее* разработки приложений для персональных компьютеров. Вы неминуемо будете сталкиваться с различными проблемами, в том числе и по независящим от вас причинам. Если это способно вывести вас из себя — поищите себе более легкое занятие. Разработка приложения, которое вы могли создать для ПК за несколько минут в Delphi или Visual Studio, для мобильного устройства может затянуться на недели или вообще оказаться невозможной. Только с опытом вы научитесь реально оценивать сложность и реализуемость проектов, определять, пройдут ли они сертификацию и видеть прочие подводные камни на своем пути. И даже тогда вы будете отнюдь не застрахованы от “сюрпризов” в виде неверно функционирующих API или отсутствия необходимой документации. Вам придется брать в расчет такие параметры работы приложения, как энергопотребление и ресурсоемкость, скорость и отказоустойчивость — все то, о чем вы просто не задумывались при программировании для ПК.

Помимо C++, вам также пригодится опыт работы с какой-либо интегрированной средой разработки (IDE), желательно с Eclipse или NetBeans. Как минимум, необходимо понимание принципов отладки приложений и умение работать с точками останова (breakpoints).

Предполагается также, что читатель умеет работать со своим смартфоном. Здесь мы не будем тратить время на описание тысячи и одного способа установки или удаления приложения с телефона, или его подключения к компьютеру.

Как работать с книгой

В книге содержится семь глав, разбитых на разделы. Ссылки на отдельные главы выделены курсивом. Вот так: *глава N*.

Чаще всего при упоминании функций в тексте, их аргументы и тип возвращаемого значения не указывается. Например: `NewLC()` или `ConstructL()`. Такое упрощение принято из соображений краткости изложения. Отличить имя функции от имени класса можно по наличию скобок `()`.

Новые термины выделяются в тексте **полужирным шрифтом**. Обычно сразу после термина в скобках указывается его английское название.

Для большинства впервые появившихся в тексте сокращений и акронимов в скобках дается расшифровка. Если впоследствии вы забудете их значения — освежить свою память вам поможет приложение A “Акронимы и сокращения” в конце книги.



В тексте вы также встретите комментарии, оформленные в виде врезок, отмеченных данной пиктограммой. Эта информация необязательна для ознакомления, но я советую обращать на нее внимание — для достижения более глубокого понимания обсуждаемых тем.

Многие темы, рассматриваемые в данной книге, входят в курс подготовки к экзамену Accredited Symbian Developer. В конце каждой из них приводятся список тезисов, которые необходимо запомнить, и требований, которым необходимо соответствовать. Приведем пример.

Подготовка к сертификации ASD

- Знание того, что такое A.
 - Умение отличать A от B.
 - Понимание необходимости использования C.
-

В конце книги приводится список, содержащий ссылки на документацию и дополнительные материалы по освещаемым темам. В основном это книги издательства Symbian Press, а также документы сообществ разработчиков Forum Nokia и Symbian Foundation. Многие из них выложены для открытого доступа в Интернете и в книге приводятся соответствующие ссылки, работоспособность которых, к сожалению, я гарантировать не могу.

Ресурсы для разработчика

Практически каждый производитель мобильных устройств, использующих Symbian OS, имеет собственные сообщества разработчиков. Безоговорочным лидером среди них является Forum Nokia. Это подразделение Nokia, в задачи которого входит взаимодействие с бизнес-партнерами и разработчиками приложений для различных устройств, выпускаемых этой компанией. Портал Forum Nokia⁵ содержит огромный форум (Discussion Boards), собственную Википедию и блоги специалистов. Там же публикуются SDK, документация и инструменты разработчика (в частности, IDE серии Carbide). Сайт сообщества Nokia предлагает информацию и доступ к форумам по всем языкам программирования и технологиям, поддерживаемым платформой S60, в том числе и по Symbian C++. Форум Discussion Boards и Википедия имеют русскоязычные разделы, а совсем недавно появилась и русскоязычная версия самого портала Forum Nokia⁶.

Еще один важный ресурс — портал Symbian Developer Community⁷. Он был открыт относительно недавно, вследствие преобразования компании Symbian Ltd в открытое сообщество Symbian Foundation. На данный момент туда переносятся все материалы Symbian Developer Network⁸. Этот ресурс позволяет получить доступ к исходному коду текущей версии Symbian OS и поучаствовать в ее разработке. Там же хранятся различные пакеты инструментов и документация для разработчиков и производителей устройств. Кроме того, на форуме можно обратиться за помощью к специалистам, участвующим в разработке Symbian OS.

Разработчикам, создающим приложения для смартфонов компании Samsung под управлением платформы S60, стоит посетить сайт Samsung Mobile Innovator⁹, а если вы пишете программы для платформы UIQ, то обязательно загляните на сайт Sony Ericsson Developers World¹⁰.

Из ресурсов, не инспирированных производителями устройств и самим сообществом Symbian Foundation, стоит отметить англоязычные порталы NewLC¹¹ и SymbianResources¹².

Еще один веб-ресурс, который я настоятельно рекомендую посетить всем разработчикам программного обеспечения для мобильных устройств, — сайт DevMobile¹³. В отличие от Forum Nokia, это сообщество русскоязычное, и посвящено всем мобильным платформам: от Symbian и Windows Mobile до Brew и Palm. На сайте представлена большая коллекция документов, статей и видеоуроков на русском языке о программировании под различные мобильные платформы.

⁵ forum.nokia.com

⁶ russia.forum.nokia.com

⁷ developer.symbian.org

⁸ developer.symbian.com

⁹ innovator.samsungmobile.com

¹⁰ developer.sonyericsson.com

¹¹ www.newlc.com

¹² www.symbianresources.com

¹³ devmobile.ru

Костяк сообщества формируют специалисты по Symbian C++, поэтому там вы всегда получите квалифицированные ответы на возникшие у вас вопросы.

Об авторе

Труфанов Александр Николаевич — выпускник Самарского государственного университета, независимый специалист в области разработки программного обеспечения для мобильных устройств, аккредитованный разработчик для Symbian OS (Accredited Symbian Developer); член экспертного совета FRUCT¹⁴, участник программ Forum Nokia Champion и Forum Nokia Advisory Council; руководитель российского сообщества разработчиков приложений для мобильных устройств DevMobile; автор многочисленных статей о программировании на Symbian C++ и переводов технической документации Symbian Press.

Обратная связь

Я старался сделать книгу максимально полезной и актуальной и готов работать над ее дальнейшим улучшением. Если у вас есть идеи, вы нашли неточность, считаете, что я что-то упустил или недостаточно подробно раскрыл, — всегда можете обсудить это со мной на форумах портала DevMobile.

Благодарности

Я благодарю сотрудников Forum Nokia за помощь, оказанную мне в подготовке книги. Без их многолетней поддержки всех разработчиков ПО для мобильных устройств и особого внимания, оказываемого развитию компетенций и бизнеса в России, было бы просто невозможно представить не только сам факт написания этой книги, но даже то, что я достигну необходимой для этого квалификации.

Я также выражаю благодарность специалистам, принимавшим участие в рецензировании книги:

- Димитрову Вячеславу Михайловичу, преподавателю кафедры ИМО Петрозаводского государственного университета;
- Михайлову Антону Александровичу, Forum Nokia Champion.

¹⁴ fruct.org

От издательского дома “Вильямс”

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@williamspublishing.com

WWW: <http://www.williamspublishing.com>

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

в Украине: 03150, Киев, а/я 152

ГЛАВА 1

Основы операционной системы Symbian

Краткая история Symbian OS

В отличие от многих операционных систем, Symbian не была портирована на мобильные устройства с персональных компьютеров, а изначально создавалась для них. Это обуславливает некоторые преимущества Symbian OS — она на уровне ядра оптимизирована для работы на критичных к энергопотреблению устройствах с минимальным объемом памяти и небольшой процессорной мощностью. Ниже приводятся основные факты и вехи в развитии Symbian OS.

- Свою родословную Symbian OS ведет от 16-битовой однопользовательской многозадачной операционной системы EPOC, разработанной компанией Psion для своих портативных компьютеров семейства SIBO (Sixteen Bit Organiser) в 1989 году. Название EPOC ничего не означает, но распространены легенды, что EPOC является укороченным “Epoch” (Эпоха), или аббревиатурой от Electronic Piece Of Cheese. Операционная система EPOC была написана на ассемблере (Intel 8086) и C, поддерживала разработку приложений на C и OPL при помощи IDE OVAL (Object-based Visual Application Language), а также обладала графическим интерфейсом (тем самым опередив Microsoft Windows 3.0). В 1991 году появился КПК Psion Series 3 под управлением ОС EPOC, оснащенный 128 Кбайт RAM и процессором, совместимым с Intel 8086.
- Впоследствии ОС EPOC была полностью переработана, и в середине 1997 года свет увидела ОС EPOC/32, впервые примененная на КПК Psion Series 5 с 4–8 Мбайт RAM. Новая операционная система разрабатывалась для процессоров с ARM-архитектурой и позволяла создавать приложения на C++. 16-битовая версия EPOC была переименована в EPOC/16 (спустя некоторое время ее стали именовать SIBO), а EPOC/32 в EPOC. Впоследствии компания Psion разделилась на Psion Computers, Psion Enterprise и Psion Software. Разработкой операционной системы занималась Psion Software. ОС EPOC постоянно совершенствовалась: на рынке появлялись устройства под управлением EPOC Release 2, EPOC Release 3 (часто упоминаются под сокращениями ER2 и ER3). Однако EPOC Release 4 не существовало.

- В июне 1998 года Psion Software, Nokia и Ericsson создали компанию Symbian Ltd. В ее задачи входила разработка новой операционной системы мирового уровня для конвергентных устройств на основе КПК и телефонов.
- В мае 1999 года к акционерам Symbian Ltd. присоединилась компания Panasonic, а вскоре была анонсирована ОС EPOC Release 5 (сейчас ее неофициально называют Symbian 5.0), содержащая виртуальную машину Java ME. Через год ее улучшенная версия EPOC 5u (ER5u или Symbian 5.1) была применена в устройстве Ericsson R380. Именно с версии 5.1 в Symbian по умолчанию используются Unicode-строки. Позднее Symbian Ltd. заключает соглашение с Sybase об использовании их технологий доступа к базам данных в мобильных устройствах.
- В 2000 году лицензиатами Symbian становятся Sanyo и Sony. Анонсированы Symbian 6.0 (первый официальный релиз) и Symbian 6.1.
- В 2001 году основана компания Symbian Press, Nokia выпускает коммуникатор 9200 на платформе Series 80 под управлением Symbian 6.0 и смартфон 7650 на платформе Series 60 под управлением Symbian 6.1. Symbian OS лицензируется Siemens и Fujitsu.
- В 2002 году совладельцами Symbian Ltd. становятся Samsung, Siemens и Sony Ericsson. Symbian OS лицензируется Sendo. Анонсируется платформа UIQ на базе Symbian 7.0, ориентированная на использование сенсорных экранов.
- В 2003 году появляются устройства под управлением Symbian 7.0 на базе платформ UIQ, Series 80, Series 90 и Series 60.
- В 2004 году Symbian OS лицензируется NTT DoCoMo, Lenovo и Sharp, анонсируются Symbian 8.1a и Symbian 8.1b (с новым ядром EKA2). Акционеры Symbian Ltd. выкупают долю Psion. Внутри компании ведется работа над Symbian 9.0.
- В 2005 году Symbian Ltd. лицензирует использование протокола Microsoft Exchange Server Active Sync. Релиз Symbian OS 9.1 был выпущен с модулем безопасности, требующим обязательной сертификации устанавливаемых приложений. В этом же году появляются устройства с новой операционной системой (под платформой UIQ3).
- В 2006 году появляются Symbian 9.2 и Symbian 9.3 с улучшенным механизмом управления памятью (demand paging), встроенной поддержкой протоколов WiFi 802.11 и HSDPA. Создаются программа сертификации разработчиков Symbian Accredited Developer (ASD) и программа взаимодействия с ВУЗами в области подготовки специалистов Symbian Academy. Продан 100-миллионный смартфон под управлением Symbian OS.
- В 2007 году на Symbian портируются библиотеки формата POSIX и SQLite. Анонсируется поддержка многоядерных процессоров, а также цифрового телевидения в форматах DVB-H и ISDB-T.
- В июне 2008 года Symbian Ltd. празднует свое десятилетие. Объявляется о создании Symbian Foundation — организации, призванной разработать новую открытую единую платформу на базе Symbian OS. В течение года ком-

пания Nokia выкупает все акции Symbian Ltd. и передает Symbian OS в ведение Symbian Foundation. Вслед за этим Nokia, Sony Ericsson, NTT DoCoMo и Samsung передают Symbian Foundation ресурсы и исходные коды платформ S60, UIQ и MOAP(s). Продано 200-миллионное устройство под управлением Symbian OS.

- В начале 2009 года объявляется о прекращении поддержки платформы UIQ. Появились устройства под управлением платформы S60 5-й редакции на базе Symbian 9.4. Позднее Symbian 9.4 и S60 5-й редакции усилиями Symbian Foundation были объединены под названием Symbian^1 и выбраны в качестве отправной точки для дальнейшей эволюции Symbian OS в качестве открытой системы. Опубликован план подготовки Symbian^2 и Symbian^3.
- В октябре 2009 года опубликован исходный код микроядра EKA2 Symbian OS.

За всю историю существования Symbian OS было выпущено более 250 моделей устройств под ее управлением от полутора десятка производителей общим количеством свыше 250 миллионов.

Операционная система и платформа

Вероятно, вы уже обратили внимание на термин **платформа** (platform), встречающийся в предыдущем разделе. Разделение на ОС и платформу существовало до недавнего времени. Дело в том, что до середины 2008 года разработкой Symbian OS занималась коммерческая компания Symbian Ltd., и использовать ее в своих устройствах могли лишь компании, приобретшие соответствующую лицензию. Однако эти компании являлись конкурентами на рынке смартфонов и коммуникаторов, и, естественно, их продукты не должны были походить один на другой. Для того чтобы завоевать большую долю на рынке, производители должны предложить покупателю нечто новое: например, поддержку элементов оборудования с выдающимися характеристиками или даже совершенно новые и зачастую патентованные сенсоры. Помимо этого, модель смартфона могла обладать совершенно особым графическим интерфейсом или быть нацелена на несколько иную нишу рынка. Наконец, каждый производитель стремился иметь собственный пакет предустановленного программного обеспечения, более удобный и функциональный чем его аналоги конкурентов. Поэтому Symbian OS являлась ядром, службы которого обеспечивали работу с памятью, файловой системой, многозадачность, виртуализацию и многое другое. Но она содержала лишь минимальный базовый графический интерфейс и поддерживала исключительно стандартное аппаратное обеспечение. Реализация же дополнительных сервисов, красочного пользовательского интерфейса, набора предустановленных служб, прикладных программ, дополнительных библиотек и средств взаимодействия виртуальной машины Java с операционной системой целиком ложилась на плечи производителей мобильных устройств. Совокупность этих перечисленных выше дополнительных компонентов и образует так называемую *платформу*.

На базе Symbian OS было создано несколько платформ: Series 60 (впоследствии переименованная в S60), Series 80 и Series 90 компании Nokia, платформа UIQ от Sony Ericsson и Motorola, а также MOAP(s), используемая японскими производителями мобильных устройств. Приложения, созданные для одной платформы, в большинстве случаев были несовместимы с другими и требовали существенных изменений при портировании.

Помимо популярной в Японии платформы MOAP(s) на базе Symbian OS существует и платформа MOAP(l) на базе Linux. Обе являются закрытыми — они не допускают установку дополнительного программного обеспечения, кроме как мидлетов для виртуальной машины Java с профилем DoJa (определенным на основе Java ME CLDC). Поэтому в дальнейшем мы не будем рассматривать платформу MOAP(s).

Не только операционная система, но и сами платформы непрерывно развивались и совершенствовались. Различные производители устройств по-разному указывают версии своих платформ. В частности, Nokia называет каждое новое поколение платформы S60 ее **редакциями** (editions), а различные версии в рамках одной редакции — **пакетами обновлений** (Feature Packs, или FP). Реже применяется числовое обозначение. В табл. 1.1 продемонстрирована взаимосвязь версий Symbian OS и использующих ее платформ.

Таблица 1.1. Версии Symbian OS и различные использующие ее платформы

Платформа		Версии Symbian OS											
		6.0	6.1	7.0	7.0s	8.0	8.1a	8.1b	9.0	9.1	9.2	9.3	9.4
S60	вер. 0.9		●										
	1 ред. (вер. 1.2)		●										
	2 ред. (вер. 2.0)				●								
	2 ред. Feature Pack 1 (вер. 2.1)				●								
	2 ред. Feature Pack 2 (вер. 2.6)					●							
	2 ред. Feature Pack 3 (вер. 2.8)						●						
	3 ред. (вер. 3.0)									●			
	3 ред. Feature Pack 1 (вер. 3.1)										●		
	3 ред. Feature Pack 2 (вер. 3.2)											●	
	5 ред. (вер. 5.0)												●
Series 80	1 ред. (вер. 1.0)	●											
	2 ред. (вер. 2.0)				●								
Series 90					●								
UIQ	2.0			●									
	2.1			●									

Окончание табл. 1.1

Платформа	Версии Symbian OS											
	6.0	6.1	7.0	7.0s	8.0	8.1a	8.1b	9.0	9.1	9.2	9.3	9.4
3.0									●			
3.1										●		
3.2										●		

С течением времени изменения на рынке мобильных устройств привели к осознанию акционерами Symbian Ltd. необходимости реформирования принципов развития Symbian OS. Компания Symbian Ltd. была преобразована в сообщество Symbian Foundation, а сама Symbian OS стала открытой (opensource) и бесплатно распространяемой. Ее версии теперь обозначаются как Symbian^1, Symbian^2 и т.д. Кроме того, было принято решение об объединении функций операционной системы и платформы. Это позволяет минимизировать затраты производителей на интеграцию операционной системы в новые устройства, а также решает проблему совместимости приложений с ними. В качестве Symbian^1 была выбрана S60 5-й редакции компании Nokia на базе Symbian 9.4. Таким образом, термин *платформа* утрачивает свой первоначальный смысл. Так как на сегодняшний день устройств под управлением Symbian^2 еще не производится, то я буду продолжать придерживаться старой терминологии.

Архитектура Symbian OS

Архитектура операционной системы чрезвычайно сложна, в этом разделе я привожу лишь поверхностное ее описание с перечислением компонентов, которые наиболее важны для программиста или будут описываться в следующих главах книги.

Symbian OS является однопользовательской многозадачной операционной системой реального времени с микроядерной архитектурой. Она содержит множество подсистем, которые в свою очередь строятся из компонентов. Большинство служб позволяет расширить свою функциональность при помощи динамически подключаемых библиотек. Подобная структура позволяет разработчикам платформ для мобильных устройств заменять или даже исключать некоторые компоненты операционной системы. Symbian OS работает как с выполняющейся на устройстве DOS, так и напрямую с аппаратным обеспечением. В свою очередь ее службы используются подсистемами платформы. Положение Symbian OS в мобильном устройстве упрощенно представлено на рис. 1.1.

Symbian OS содержит следующие подсистемы.

- Ядро и интерфейсы взаимодействия с аппаратным обеспечением (Kernel Services, HAL).
- Службы операционной системы (OS Services):
 - базовые сервисы (Base Services);
 - коммуникационные сервисы (Comms Services);

- службы мультимедиа (Multimedia);
- службы работы с графикой (Graphics);
- сервисы определения местоположения (Location Based Services, LBS).
- Общие вспомогательные службы (Generic Middleware):
 - службы поддержки приложений (Generic Application Support);
 - сервисы безопасности (Security Management);
 - прикладные протоколы (Application Protocols);
 - вспомогательные службы мультимедиа (Multimedia Middleware);
 - подсистема графического интерфейса (System GUI Framework).
- Прикладные службы (Application Services):
 - службы управления персональной информацией (Personal Information Management, PIM);
 - службы сообщений (Messaging);
 - службы синхронизации и удаленного управления (Remote Management);
 - Java 2 ME.

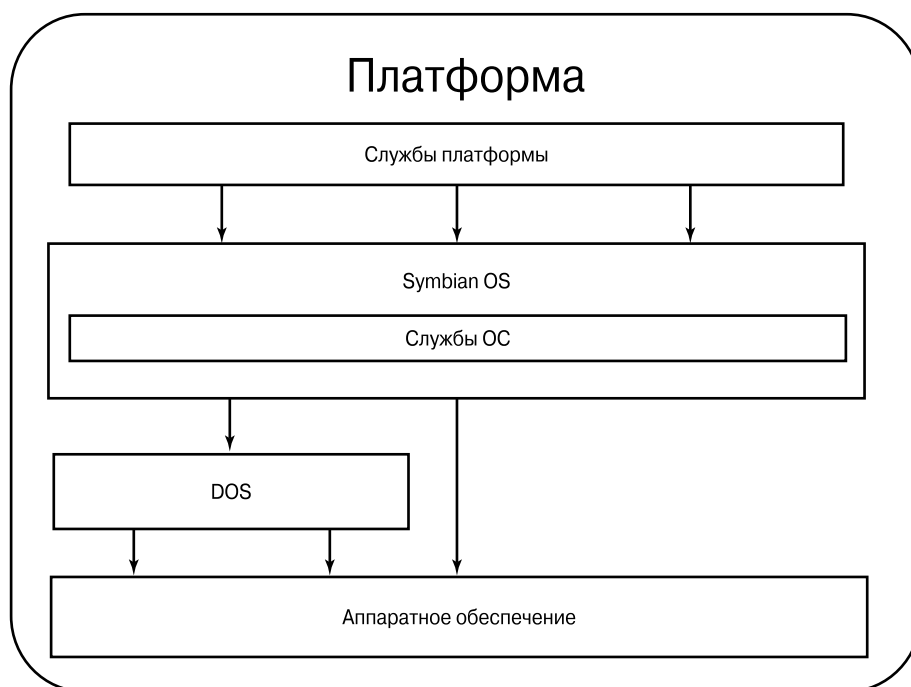


Рис. 1.1. Место Symbian OS в программном обеспечении мобильного устройства

Ядро ЕКА2

В Symbian 9.x применяется второе поколение архитектуры ядра операционной системы — ЕКА2 (ЕРОС Kernel Architecture 2). Эта архитектура разра-

батывалась для Symbian 7.x, но впервые появилась лишь в Symbian 8.1b. Ядро архитектуры ЕКА2 содержит микроядро, службы, обеспечивающие загрузку и исполнение ядра, а также интерфейсы взаимодействия с аппаратным обеспечением. Ядро и часть сопутствующих служб выполняются в привилегированном режиме. В отличие от функционирующего в одном потоке ядра ЕКА1, в ЕКА2 применяется вытесняемая многопоточность. Микроядро обеспечивает работу в реальном времени с сигнальным стеком GSM, wCDMA и CDMA. В функции ядра входит: создание, управление, планирование и прерывание процессов и потоков; организация работы с памятью; обработка прерываний; реализация таймеров, мьютексов и семафоров; работа с аппаратным обеспечением. Помимо этого ядро ЕКА2 содержит двухуровневую модель логических/физических драйверов устройств и средства поддержки локализации.

Ядро ЕКА2 включает механизм межпроцессного взаимодействия (Inter-process communication, IPC) и содержит компоненты для обеспечения работы платформы безопасности. В ядре не поддерживаются сбросы и их обработка. Большинство компонентов ЕКА2 аппаратно-зависимы, поэтому это ядро организовано так, чтобы максимально облегчить портирование имеющегося программного обеспечения на новое устройство.

Службы операционной системы

Базовые службы

Базовые службы Symbian OS дополняют функциональность ядра, но выполняются в непривилегированном пользовательском режиме. Наиболее значимыми компонентами, входящими в эту категорию, являются библиотека User (User Library), файловый сервер (File Server), службы СУБД (DBMS), центральный репозиторий (Central Repository), система ECom, библиотека BAFL (Basic Application Framework Library).

Библиотека User является основным интерфейсом для доступа к таким объектам ядра, как процессы, потоки, мьютексы и очереди сообщений. В ней также определены базовые типы данных, константы и классы, реализованы инструменты для работы с датой и временем, таймеры, массивы, деревья и списки, наиболее важные идиомы Symbian: активные объекты, стек очистки, дескрипторы, UID, средства поддержки клиент-серверной архитектуры приложений и механизм публикаций и уведомлений (Publish & Subscribe). Знакомство с этими классами и идиомами мы продолжим в следующих главах книги. Библиотека User используется практически всеми сервисами и приложениями Symbian OS, кроме служб ядра.

Файловый сервер — служба, стартующая первой сразу после загрузки и запуска ядра. Файловый сервер использует подключаемые модули, обеспечивающие работу с различными файловыми системами (FAT, ROFS). В нем также реализованы некоторые вспомогательные функции для работы с файлами. Подробнее о файловом сервере рассказывается в *главе 6*, раздел “Работа с файловой системой”.

Службы поддержки СУБД позволяют создавать реляционные базы данных, хранящиеся в одном файле или файловом потоке. Они определяют API для обращения к подобным хранилищам напрямую и с помощью SQL-запросов для одного или нескольких клиентов. Подробнее о СУБД Symbian OS будет сказано в *главе 6*, раздел “Базы данных”.

Центральный репозиторий представляет собой универсальное хранилище настроек системы, а также механизм уведомлений об их изменении.

Система ECom предоставляет функции для регистрации, поиска и загрузки различных реализаций интерфейсов в полиморфных DLL. При помощи ECom в Symbian OS осуществляется подключение разнообразных плагинов. Подробнее эта система будет рассмотрена в *главе 6*, раздел “Механизм ECom”.

Библиотека BAFL (Basic Application Framework Library) предоставляет набор вспомогательных классов и методов, наиболее широко используемыми из которых являются динамические массивы дескрипторов и функции для работы с файловой системой.

Коммуникационные службы

Коммуникационные службы имеют довольно сложную организацию и зависят от аппаратной архитектуры. Они содержат реализацию протоколов и служб RS232, IrDA, USB, Bluetooth, Wi-Fi, SMS, MMS, WAP, OBEX, SIP, стека TCP/IP, SyncML и все необходимое для выполнения звонков и передачи данных в 2G-, 2.5G- и 3G-сетях. В состав коммуникационных сервисов входят последовательный сервер C32 (C32 Serial Server), сервер телефонии ETel (ETel Telephony Server), сервер сокетов ESock (ESock Socket Server), факс-сервер (Fax Server), а также запускающий и останавливающий их корневой сервер коммуникаций (Comms Root Server).

Сервер C32 предоставляет API для последовательного доступа к коммуникационным ресурсам (последовательный порт, Bluetooth, IrDA, USB и т.п.) и поддерживает CSY-плагины, реализующие или эмулирующие последовательный порт для различных каналов связи.

Сервер телефонии ETel оперирует такими объектами, как телефон (phone), линия (line) и звонок (call), доступ и управление которыми можно осуществить при помощи специального API. Аналогично серверу C32, он использует схожие по назначению подключаемые модули TSY. В состав Symbian OS входят MultiMode TSY, CDMA TSY и SIM TSY. MultiMode TSY предоставляет функциональность GSM и GPRS, связываясь с телефоном или модемом посредством AT-команд через последовательный или инфракрасный канал. CDMA TSY является аналогом MultiMode TSY, а SIM TSY эмулирует аппаратное обеспечение устройства на основе заданной конфигурации. В SDK входит лишь ETel API для сторонних лиц (ETel Third-Party API), предоставляющее несколько ограниченный (в целях безопасности) доступ к серверу.

Сервер ESock позволяет осуществлять передачу данных при помощи сокетов, реализующих различные протоколы. Сами протоколы подгружаются из PRT-плагинов. На транспортном уровне сервер ESock пользуется сервисами

серверов ETel и C32. Именно сервер ESock ответственен за отправку сообщений (при помощи модуля SMS PRT). Более подробно работа с сокетами рассмотрена в *главе 6*, раздел “Сокеты”.

Службы мультимедиа

Symbian OS содержит единую подсистему для управления аудио, видео, MIDI, камерой, системой распознавания речи и другими ресурсами мультимедиа. Она построена на использовании Ecom-плагинов, позволяющих расширить список поддерживаемых форматов аудио и видео.

Службы работы с графикой

Операционная система обеспечивает весьма богатые возможности работы с графикой. Symbian OS поддерживает как растровые, так и векторные шрифты FreeType, в том числе нелатинские шрифты и шрифты, позволяющие читать справа налево. Ключевыми компонентами этой подсистемы являются сервер окон (Window Server), а также компоненты Bit GDI и GDI. Более подробно они будут рассмотрены в *главе 6*, раздел “Сервер окон”. Для работы с 3D-графикой в Symbian OS используется OpenGL ES.

Службы определения местоположения

Symbian OS поддерживает определение местоположения устройства при помощи технологий GPS, A-GPS и на основе информации, предоставляемой сетью связи. Для этого в системе имеется сервер позиционирования (Location Server), использующий систему PSY-плагинов для доступа к различным методам определения местоположения. По мере необходимости сервер позиционирования выполняет загрузку и выгрузку модулей плагинов, а также сбор и обработку полученных ими данных. Помимо этого он предоставляет клиентским приложениям API для определения и уведомления об изменении местоположения. В состав сервисов определения местоположения также входит механизм организации и управления базами геометок (landmarks).

Общие вспомогательные службы

Службы поддержки приложений

Содержат системы конвертирования файлов и HTML-данных, форматирования текста, а также сервер часовых поясов (Timezone Server), сервер напоминаний (Alarm Server) и механизм резервного копирования.

Службы безопасности

К этой категории относятся службы криптографии и установщики пакетов приложений SIS и JAR. Сервисы безопасности играют значительную роль в платформе безопасности Symbian OS, подробно обсуждаемой в соответствующем разделе этой главы.

Прикладные протоколы

К ним относятся компоненты, реализующие протоколы HTTP, WAP, WAPPush, а также профили Bluetooth и различные утилиты для работы с HTTP.

Вспомогательные службы мультимедиа

К этой категории относятся реализации протоколов MTP, RTP и SIP, применяемых в IP-телефонии.

Подсистема графического интерфейса

Как уже говорилось ранее, Symbian OS содержит лишь подсистемы базового графического интерфейса, которые затем надстраиваются службами уровня платформы. Главную роль в ней играют системы Uikon, CONE и FEP.

Служба FEP (Front-end processor) занимается предобработкой вводимой пользователем информации, в частности, распознаванием рисуемых на сенсорном экране символов и определением вводимого символа при многократном нажатии на кнопку (T9). Для этих целей FEP использует подключаемые плагины, реализующие подобную предобработку, а также определяет базовые классы для создания таких плагинов.

Библиотека CONE (Control Environment, окружение элементов интерфейса) предоставляет абстрактные и базовые классы для создания элементов интерфейса. Она может использоваться как напрямую, так и из Uikon или служб пользовательского интерфейса платформы. Помимо этого, CONE осуществляет трансляцию различных событий (в том числе от сервера окон) в элементы интерфейса.

Фреймворк Uikon пришел на смену Eikon, как только Symbian OS стала поддерживать исключительно Unicode, и является основой для служб пользовательского интерфейса на уровне платформы (таких, как Avkon в S60 и Qikon в UIQ). Uikon предоставляет базовые классы для архитектуры приложений (модель Application-View-AppUi), а также фабрику классов для кнопок, списков, диалогов и других элементов интерфейса. При необходимости Uikon запускает требуемые для работы приложения сервера операционной системы. В состав Uikon входят серверы для обеспечения резервного копирования приложения и для завершения работы приложений перед отключением устройства. Для того чтобы определить поведение элементов графического интерфейса (внешний вид кнопок, поведение окон, шрифты по умолчанию), Uikon загружает статическую библиотеку LAF (Look-and-Feel), подготавливаемую разработчиками платформы.

Помимо вышеперечисленных компонентов в состав подсистемы графического интерфейса Symbian OS входят библиотеки, предоставляющие различные графические утилиты и эффекты, а также механизм воспроизведения анимации с помощью растровых изображений или спрайтов.

Прикладные службы

Службы управления персональной информацией

К этой категории относятся компоненты, реализующие спецификации vCal и vCard консорциума Versit, база данных, содержащая контактную книгу, и API для работы с ней, а также служба календаря с поддержкой формата iCalendar.

Службы сообщений

Стандартные типы сообщений (например, E-mail, SMS или MMS) в Symbian OS обрабатываются соответствующим сервером (Messaging Server) и помещаются в хранилище (Messaging Store). Поддержка различных типов сообщений обеспечивается системой подключаемых хранилищем MTM-плагинов. Symbian OS содержит плагины для отправки, получения и редактирования различных сообщений: SMS, MMS, Email (POP3, IMAP4, SMTP) и OBEX. Помимо этого, в системе существуют служба отправки сообщений по расписанию, службы обработки сообщений BIO (Bearer Independent Objects) и механизм уведомления приложений об их поступлении.

Службы синхронизации и удаленного управления

Symbian OS поддерживает стандарт SyncML и содержит компоненты и службы для синхронизации контактной книги, календаря, записок, закладок и сообщений.

Java 2 ME

В состав Symbian OS входят профиль MIDP 2.0, пакеты MIDP 2.0, конфигурация CLDC 1.1 и виртуальная машина Java. Реализация спецификаций JSR ложится на плечи разработчиков платформы.

Исполняемые файлы в ROM и RAM

Устройства под управлением Symbian OS обычно имеют два типа накопителей: ROM-память (диск Z :), содержимое которой записывается при производстве, и RAM-память (встроенная или внешняя). Файлы EXE и DLL могут выполняться непосредственно с ROM-накопителя, при этом в ОЗУ загружаются только секции изменяемых данных приложения. В этом случае в библиотеках в целях экономии места удаляется вся информация, позволяющая выполнять настройку адресации (relocation — перемещение). Поэтому нельзя просто скопировать DLL из ROM-памяти и использовать ее в RAM.

Исполняемые файлы, устанавливаемые на устройство пользователем, хранятся в RAM и полностью загружаются в ОЗУ перед выполнением. При этом при выполнении нескольких копий одного приложения или использовании одной разделяемой DLL их секции кода и неизменяемых данных используются

повторно, в памяти создаются лишь копии изменяемых секций. Такие исполняемые файлы полностью выгружаются из памяти только после того, как завершит свою работу последняя копия программы (или все программы, использующие разделяемую DLL).

Подготовка к сертификации ASD

- Умение определять правильность утверждений о выполнении EXE и DLL в ROM и RAM Symbian OS.
-

Уникальные идентификаторы в Symbian OS

Уникальный идентификатор (UID) в Symbian OS представляет собой 32-рядное беззнаковое число, чаще всего записываемое в шестнадцатеричном виде и принимающее значение от 0x00000000 до 0xFFFFFFFF. UID в Symbian OS достаточно широко используются:

- для идентификации различных объектов;
- для декларации совместимости с чем-либо, принадлежности к чему-либо или реализации какого-либо интерфейса;
- для ассоциации приложений с типами документов.

Все исполняемые файлы и многие типы файлов данных в Symbian OS содержат в своем заголовке (первые 12 байт) три идентификатора: UID1, UID2 и UID3.

- UID1 задает тип файла (EXE, DLL, файловое хранилище, файл ресурсов или что-либо иное).
- Значение UID2 зависит от типа файла. Например, в исполняемых файлах оно игнорируется, а в динамических полиморфных библиотеках указывает на реализуемые в них интерфейсы.
- UID3 служит для различения файлов с одинаковыми значениями UID1 и UID2.

Помимо трех числовых значений UID, в исполняемых файлах могут быть указаны два идентификатора специального назначения: SID (Secure ID, или идентификатор безопасности) и VID (Vendor ID, или идентификатор производителя). Более подробно о них будет рассказано в следующем разделе.

Платформа безопасности Symbian OS

Платформа безопасности (Platform Security или PlatSec) — концептуальное решение, в которое входят как отдельные компоненты Symbian OS, так и некоторые элементы служб, напрямую не связанных с безопасностью (поддержка платформы безопасности присутствует даже в ядре Symbian OS). Платформа

безопасности впервые появилась в Symbian 9.x и направлена на борьбу со следующими явлениями:

- вредоносным программным обеспечением;
- неправомерным доступом к данным пользователя или устройства;
- неправомерным использованием платных служб (отправка SMS-сообщений, выполнение звонков);
- пиратским использованием контента.

В основу платформы безопасности положено понятие “**защищенная возможность**” (capability). В Symbian OS эти элементы играют роль меток и используются для определения легитимности доступа к тому или иному ресурсу или API системы. Около 40% всех API Symbian OS 9.x защищены подобным образом.

Всего в платформе безопасности объявлено двадцать защищенных возможностей, разбитых на четыре категории (перечислены по возрастанию важности).

- **Пользовательские возможности** (User Capabilities) — защищают наиболее часто используемые пользователем функции:
 - **LocalServices** — доступ к локальным сервисам (Bluetooth, IrDA);
 - **Location** — доступ к системам определения местоположения (GPS, A-GPS);
 - **NetworkServices** — высокоуровневый доступ к сетевым сервисам, таким как WLAN и GPRS;
 - **ReadUserData** — чтение конфиденциальной информации пользователя (например, из контактной книги или SIM-карты);
 - **UserEnvironment** — доступ к сервисам пользовательского окружения (камера, запись аудио);
 - **WriteUserData** — изменение конфиденциальной информации пользователя.
- **Системные возможности** (System Capabilities) — защищают сервисы системы, настройки устройства и некоторые функции аппаратного обеспечения:
 - **PowerMgmt** — завершение выполнения процессов, подготовка и выключение устройства;
 - **ProtServ** — использование защищенных имен для серверов, присвоение процессам и потокам защищенного статуса;
 - **ReadDeviceData** — получение информации об устройстве и сотовой сети;
 - **SurroundingsDD** — доступ к драйверам логических устройств, предоставляющих информацию об окружении устройства;
 - **SwEvent** — перехват и отправка событий, связанных с использованием клавиатуры;

- **TrustedUI** — позволяет отображать диалоги в безопасном UI-окружении для предотвращения их подделки;
- **WriteDeviceData** — изменение настроек устройства и сервисов системы.
- **Ограниченные возможности** (Restricted Capabilities) — защищают файловую систему, коммуникационные и сервисы мультимедиа устройства:
 - **CommDD** — прямой доступ к драйверам коммуникационных служб (USB, WiFi, последовательный порт);
 - **DiskAdmin** — управление логическими дисками (проверка, форматирование, установка пароля), монтирование файловых систем;
 - **NetworkControl** — прямой доступ к API сервера телефонии (звонки, линии, работа с SMS, USSD) и корневого сервера коммуникаций;
 - **MultimediaDD** — более широкий доступ к мультимедийным API (обработка аудио, видео).
- **Возможности производителя устройств** (Manufacturer Capabilities) — защищают наиболее важные сервисы системы:
 - **AllFiles** — доступ к некоторым защищенным каталогам файловой системы, получение уведомлений об изменениях в файловой системе;
 - **DRM** — контроль доступа к цифровому контенту (видео, музыка) и его использования;
 - **ТСВ** — доступ на запись к исполняемым и разделяемым ресурсам, которые отмечены как “только для чтения”.

Каждое приложение или библиотека содержит список используемых ими защищенных возможностей. Он задается разработчиком и кодируется в заголовке файла во время компиляции. После запуска приложения процесс наследует список доступа, породившего его исполняемого файла. Этот список также разделяется и подгружаемыми процессом библиотеками. Таким образом, именно процесс является “**единицей доверия**” (unit of trust) в Symbian OS.

Кроме того, процесс является единицей защиты памяти. Прямой доступ к виртуальному адресному пространству одного процесса другим невозможен и предотвращается на аппаратном уровне.

Использование ряда функций API и ресурсов системы требует наличия у процесса доступа к тем или иным защищенным возможностям. Например, метод `GetSubscriberId()` класса `CTelephony` позволяет получить IMSI-номер SIM-карты устройства, но для того, чтобы им воспользоваться, процесс должен декларировать доступ к защищенной возможности **ReadDeviceData**. В обратном случае результатом работы метода будет `-21` (код ошибки `KErrAccessDenied`). Некоторые функции API могут требовать декларирования сразу нескольких защищенных возможностей. Более того, в редких случаях эти требования могут меняться, в зависимости от того, к каким ресурсам обращается метод или какие плагины необходимо загрузить для его выполнения. Информацию о том, ограничено ли использование той или иной функции API с помощью защищенных возможностей можно получить в справочнике SDK.

Платформа безопасности предполагает три абстрактных уровня доверия.

- **Высоконадежная вычислительная база** (Trusted Computing Base, TCB) — представляет собой наиболее доверенный уровень. Входящие в состав TCB компоненты являются неотъемлемой частью платформы безопасности и реализуют основные ее функции. Компонентами высоконадежной вычислительной базы являются элементы аппаратного обеспечения, ядро Symbian OS, некоторые драйверы, файловый сервер и установщик приложений. Службы уровня TCB имеют доступ ко всем защищенным возможностям и обеспечивают полный контроль над работой всех приложений в системе: их установку, загрузку в память исполняемых файлов, создание и управление процессами. Поэтому в целях безопасности в TCB включается минимум компонентов.
- **Высоконадежное вычислительное окружение** (Trusted Computing Environment, TCE) — данный уровень составляют многочисленные системные серверы. Эти компоненты обычно предоставляются создателями операционной системы, разработчиками использующей ее платформы, производителем устройства и другими надежными источниками. Они менее привилегированны, чем компоненты уровня TCB — во многом потому, что такой уровень доверия не является необходимым для их работы. Службы TCE имеют доступ лишь к некоторым используемым ими защищенным возможностям системы и контролируются компонентами уровня TCB.
- **Уровень приложений** — третий уровень доверия. Его составляют как предустановленное ПО устройства, так и устанавливаемые пользователем программы. Приложения в своей работе обращаются к серверам уровня TCE и (иногда) к компонентам TCB. Они делятся на **надежные** (trusted) и **ненадежные** (untrusted). Процессы ненадежных приложений могут обращаться только к пользовательской категории защищенных возможностей. Существует также ряд иных накладываемых на них ограничений. Например, ненадежные приложения не могут автоматически запускаться при старте системы. Надежные приложения могут получить доступ к различным защищенным возможностям, но чем более они важны для системы, тем сложнее это сделать. Например, если доступ к категории системных возможностей для своего приложения может обеспечить практически любой разработчик, то при использовании возможностей производителя устройств потребуется специальное разрешение. Более подробно данный вопрос будет освещен несколько позднее.

Подготовка к сертификации ASD

- Понимание аксиомы “процесс — единица доверия” и как ее реализует Symbian OS.
- Понимание назначения высоконадежной вычислительной базы (TCB) и причин ее важности.
- Понимание того, что большинство API Symbian OS не защищено и не требует проверок уровня доступа перед использованием.

- Знание различных категорий защищенных возможностей и их назначение.
 - Понимание взаимосвязи между защищенными возможностями и TCB.
-

Защищенные возможности в библиотеках

Как уже говорилось ранее, список доступа к защищенным возможностям есть не только у исполняемых файлов, но и у динамически подключаемых библиотек DLL. При подключении библиотеки к процессу она разделяет с ним и доступ к защищенным возможностям. Если учесть то, что концепция платформы безопасности распространяется не только на пользовательские приложения, но и на системные службы и серверы, а также то, насколько широко применяются плагины (чаще всего в форме полиморфных DLL) в Symbian OS, то мы увидим множество возможностей для нелегитимного доступа к защищенным ресурсам. Чтобы этого не произошло, платформа безопасности включает следующее правило.

Библиотека может быть загружена процессом только в том случае, если она декларирует доступ к такому же или более широкому набору защищенных возможностей.

Это же правило распространяется на загрузку всех библиотек, используемых подключаемой библиотекой. Если загрузить библиотеку не удастся, программа просто не запустится.

На практике, для того чтобы расширить возможность использования DLL в сторонних программах, в них декларируют доступ даже к тем защищенным возможностям, которые не используются.

Идентификаторы VID и SID

В платформе безопасности определены два специальных вида идентификатора: **идентификатор безопасности** (SID или Secure ID) и **идентификатор производителя** (VID или Vendor Id).

Каждый исполняемый файл в Symbian 9.x имеет идентификатор безопасности SID. Он может быть задан при компиляции, в противном случае в качестве SID система использует UID3. При запуске приложения процесс получает SID исполняемого файла. Значение SID используется как уникальный идентификатор приложения — если вы попытаетесь установить программу с дублирующим значением SID, инсталлятор прервет работу, сообщив об ошибке. Помимо этого, SID используется системой для определения доступа процесса к защищенным возможностям, в частности — при определении приватного каталога (см. раздел “Экранирование данных” ниже в этой главе). В клиент-серверных решениях сервер может в целях безопасности ограничивать доступ к своим сервисам по значению SID клиента. Значение SID в библиотеках игнорируется. На практи-

ке разработчики часто не задают SID, позволяя ему принимать значение UID3 исполняемого файла.

Идентификатор VID может задаваться как в исполняемых файлах, так и в библиотеках (по умолчанию он имеет нулевое значение). Значение VID для библиотек игнорируется. VID служит для уникальной идентификации автора файла. Например, VID компании Nokia равен 0x101FB657. При использовании ненулевого значения VID приложение должно обязательно пройти сертификацию на Symbian Signed. Подробнее вопросы сертификации рассматриваются в главе 7. На практике VID сторонними разработчиками не используется.

Существует несколько диапазонов значений для уникальных идентификаторов, разбитых на две области (табл. 1.2).

Таблица 1.2. Диапазоны значений уникальных идентификаторов

Область значений	Класс	Диапазон значений	Назначение
Защищенная область значений	0	0x00000000 – 0x0FFFFFFF	Только для разработки
	1	0x10000000 – 0x1FFFFFFF	Унаследованные значения UID (использованные до Symbian 9.x)
	2	0x20000000 – 0x2FFFFFFF	Защищенные значения
	3	0x30000000 – 0x3FFFFFFF	Зарезервировано
	4	0x40000000 – 0x4FFFFFFF	Зарезервировано
	5	0x50000000 – 0x5FFFFFFF	Зарезервировано
	6	0x60000000 – 0x6FFFFFFF	Зарезервировано
Незащищенная область значений	7	0x70000000 – 0x7FFFFFFF	Значения для VID
	8	0x80000000 – 8x0FFFFFFF	Зарезервировано
	9	0x90000000 – 0x9FFFFFFF	Зарезервировано
	A	0xA0000000 – 0x2AFFFFFFF	Незащищенные значения
	B	0xB0000000 – 0xBFFFFFFF	Зарезервировано
	C	0xC0000000 – 0xCFFFFFFF	Зарезервировано
	D	0xD0000000 – 0xDFFFFFFF	Зарезервировано
	E	0xE0000000 – 0xEFFFFFFF	Только для разработки
	F	0xF0000000 – 0xFFFFFFFF	Для совместимости с унаследованными UID

Принципиальное отличие этих областей значений заключается в том, что значения из защищенной области получают разработчиком с сайта Symbian Signed¹, где хранится база соответствия выданных UID и запросивших их разработчиков. Использование UID3 (SID) из защищенной области значений является гарантией того, что на устройстве не окажется приложения с таким же идентификатором. С другой стороны, хранение назначенных идентификаторов в базе позволяет быстро определить автора программы.

Значения UID3 из незащищенного диапазона выбираются случайным образом и используются в приложениях, не предназначенных для широкого распространения, например, в демонстрационных примерах SDK.

Подготовка к сертификации ASD

- Умение объяснить, что такое идентификатор безопасности SID, где он задается и как используется.
 - Понимание сходства и различий идентификатора безопасности (SID), идентификатора производителя (VID) и уникальных идентификаторов файлов (UID1, UID2, UID3).
 - Знание того, что SID и VID могут быть заданы в DLL, но игнорируются системой.
 - Знание правил, по которым идентифицируется приложение в соответствии с SID, VID и UID.
 - Понимание того, что значения UID делятся на две категории (из защищенного и незащищенного диапазона) и их влияние на тестирование и сертификацию.
-

Экранирование данных

Экранирование данных (Data caging) — одна из концепций платформы безопасности Symbian OS, основная идея которой заключается в сокрытии части файловой системы от процессов, не имеющих доступа к необходимым защищенным возможностям. Все операции с файловой системой выполняются файловым сервером Symbian OS (являющегося компонентом TCB), который в состоянии проверить список доступа к защищенным возможностям подключившегося к нему процесса и в зависимости от его состава корректировать возвращаемый результат. Таким образом, процесс, не имеющий доступа к определенным возможностям, получит отказ в доступе к определенным файлам и каталогам и даже не обнаружит их существования. Экранирование данных используется для защиты как системных данных, так и данных процессов. При этом даже большинство системных служб не имеет доступа к ним. Давайте более подробно рассмотрим экранируемые каталоги и их назначение.

¹ www.symbiansigned.com

- `\sys\` — каталог, для чтения содержимого которого необходима декларация защищенной возможности **AllFiles**, а для изменения данных — **TCB**. В каталоге `\sys\bin\` хранятся исполняемые файлы и библиотеки различных приложений. Это надежная защита от изменения исполняемых файлов вредоносными программами.
- `\resource\` — здесь хранятся файлы ресурсов приложений. Этот каталог со всем содержимым доступен для чтения без ограничений. Модификация находящейся в нем информации требует наличия доступа к возможности **TCB**.
- `\private\` — каталог, содержащий конфиденциальные данные приложений. При установке программы для каждого исполняемого файла в нем отводится подкаталог `\private\<SID>\`, где `<SID>` — значение соответствующего идентификатора в шестнадцатеричном виде и без префикса “0x”. Доступ к нему может получить лишь приложение с этим SID либо процесс, декларировавший доступ к возможностям **AllFiles**. Таким образом, каждый исполняемый файл имеет собственный, закрытый для прочих программ каталог для хранения конфиденциальной информации (настроек, ресурсов и пр.).

В приватном каталоге процесса существует особый подкаталог: `\private\<SID>\import\`. В него могут помещаться файлы сторонних приложений во время их инсталляции (эту работу выполняет системный установщик SIS-пакетов). Этот механизм может служить для регистрации сторонних программ в системах вашего приложения или установки плагинов. Например, все устанавливаемые в Symbian OS программы, желающие отображаться в системном списке приложений, должны поместить специальный файл-ресурс с регистрационной информацией в каталог `\private\10003a3f\import\apps\` (SID `0x10003A3F` принадлежит серверу AppArcServer, файл `ApsExe.exe`). А для автозапуска программы при загрузке смартфона необходимо во время ее установки скопировать специальный файл в каталог `\private\101f875a\import\` (SID `0x101F875A` принадлежит приложению App Installer, файл `SWInstSvrUi.exe`).

При деинсталляции приложения его приватный каталог и все его содержимое уничтожается автоматически.

Так как Symbian OS допускает установку приложений не только в память телефона, но и на съемные носители, система экранирования данных справедлива и для них. Разумеется, съемный носитель можно извлечь из устройства и, подключив его к ПК, изменить содержимое защищенных каталогов. Для предотвращения подобной возможности во время установки приложения вычисляется хеш (контрольная сумма) исполняемых файлов. Хеш помещается в специальное хранилище в памяти устройства. Помимо этого, разработчик может запретить установку своего приложения на съемные носители в настройках SIS-пакета.

Подготовка к сертификации ASD

- Понимание того, что экранирование данных предназначено для защиты всех типов файлов в трех специальных каталогах (`\sys\`, `\resource\` и `\private\`).

В частности, экранирование данных используется для изоляции всех исполняемых файлов, поэтому, раз прошедшие проверку, они защищены от изменения.

- Понимание влияния экранирования данных на именование исполняемых файлов.
 - Понимание того, что экранирование данных может быть использовано для выделения защищенной области на диске для данных приложения.
 - Знание того, декларация каких защищенных возможностей необходима для доступа на чтение и запись к защищенным каталогам и их подкаталогам.
 - Знание того, что библиотеки не имеют собственного приватного каталога и используют приватный каталог загрузившего их процесса.
-

Установка приложений и сертификаты

Внимательный читатель, видимо, уже задал себе вопрос: что мешает разработчику объявить в исполняемом файле доступ ко всем защищенным возможностям и получить полную свободу действий? Ответ кроется в процедуре инсталляции SIS-пакетов.

Платформа безопасности Symbian 9.x не позволяет пользователю получить прямой доступ к каталогу `\sys\bin\`, хранящему исполняемые файлы. Поэтому единственный способ инсталлировать приложение на устройство — воспользоваться системным установщиком (являющимся компонентом TCB). Для этого программа, сопутствующие файлы и ресурсы упаковываются в файл с расширением `.sis` или `.sisx`. SIS-пакет хранит информацию о каталогах, в которые следует скопировать хранящиеся в нем файлы, а также ряд настроек для установщика. Более подробно инструменты для создания SIS файлов рассматриваются в *главе 3*, “Работа с SDK”.

Перед установкой в SIS-файл внедряется **цифровой сертификат** (digital certificate), содержащий следующую информацию:

- кем, когда и кому он выдан;
- срок его действия;
- список IMEI-номеров устройств, на которые разрешена установка данного SIS-файла;
- список защищенных возможностей, доступ к которым он удостоверяет.

Процедура прикрепления цифрового сертификата называется **подписыванием** (signing), а сам SIS-файл становится **подписанным** (signed). Подписать SIS-пакет можно сразу несколькими сертификатами. SIS-файл без сертификата является **неподписанным** (unsigned). В Symbian 9.x установка неподписанных SIS-файлов запрещена.

Системный установщик играет роль своеобразного шлюза, проверяя сертификаты инсталлируемых SIS-файлов. В том случае, когда сертификат отсутствует либо срок его действия истек, приложение установлено не будет. После инсталляции программа может использоваться неограниченно долго — к ее работе срок действия сертификата отношения не имеет. При наличии в системе опреде-

ленных настроек установщик также может выполнить онлайн-проверку сертификата и отказать в инсталляции приложения, чей сертификат был отозван.

Установщик также проверяет, что декларируемый в приложении доступ ко всем защищенным возможностям удостоверен сертификатом. Для этого он сканирует все содержащиеся в SIS-пакете исполняемые файлы и формирует общий список декларируемых в них защищенных возможностей. Такой же список формируется на основании всех прикрепленных к пакету сертификатов, после чего оба списка сравниваются. Если во множестве декларируемых исполняемыми файлами защищенных возможностей есть хоть одна, не содержащаяся во множестве возможностей, удостоверенных сертификатами, то установка приложения прерывается. Исключением является группа пользовательских возможностей — их использование может не заверяться сертификатом.

Здесь можно отметить некоторое противоречие: с одной стороны, установка неподписанных приложений в Symbian 9.x запрещена, а с другой, — программа, не использующая защищенных возможностей или ограничивающаяся доступом лишь к категории пользовательских возможностей, не нуждается в сертификате. Такое противоречие разрешается подписанием SIS-пакета **сертификатом-пустышкой**, не удостоверяющим каких-либо возможностей. Такой сертификат может сгенерировать сам разработчик с помощью утилит SDK (подробнее об этом рассказывается в *главе 3*). Сертификат-пустышка в англоязычной документации обозначается как “self-generated certificate”, а подписанный с его помощью файл — “self-signed”. Так как получить сертификат self-generated может любой, приложения, устанавливаемые при помощи пакетов self-signed, относятся к категории ненадежных.

Список защищенных возможностей, не требующих заверения сертификатом при установке, в действительности задается в файле `\system\data\swipolicy.ini` и может изменяться производителем устройств. Предполагается, что в него входят все возможности из категории пользовательских, и я не встречал устройства, где это было бы не так. Но вот сам состав группы пользовательских возможностей не так давно изменился — в него вошла возможность **Location**, ранее относившаяся к системным. На устройствах с платформой S60 и S60 FP1 возможность **Location** в файле `swipolicy.ini` не упомянута, и объявивший ее self-signed пакет не будет установлен.

Помимо типа self-generated, есть еще один специальный тип сертификата — **сертификат разработчика** (DevCert, Developer Certificate). Получить его относительно легко, но подписанный сертификатом разработчика SIS-пакет может быть установлен лишь на те устройства, номер IMEI которых закодирован в сертификате. Поэтому он используется только в процессе разработки и тестирования.

Вопросами сертификации приложений для Symbian OS занимается организация под названием Symbian Signed. На их сайте www.symbiansigned.com есть сервис Open Signed Online, где вы сможете подписать свой SIS-файл сертификатом разработчика для одного устройства (указав его IMEI). При этом вы сможете сформировать список заверяемых им защищенных возможностей

из числа пользовательских и системных. Заполнив все формы и отправив SIS-файл на сервер, вы через некоторое время получите подписанный SIS-пакет на указанный почтовый адрес E-mail. Сам файл сертификата разработчика вы получить не сможете.

Использование сервиса Open Signed Online накладывает ряд ограничений на содержащиеся в SIS-пакете приложения. Главным из них является использование значения UID3 (SID) из незащищенного диапазона (0xE0000000...0xFFFFFFFF).

Владельцы **идентификатора издателя** (Publisher ID) имеют возможность при помощи сервиса Open Signed Offline заказать сертификат разработчика с любыми защищенными возможностями, кроме группы возможностей производителя. Такой сертификат может позволять установку на несколько устройств (указывается список IMEI — до 1000 номеров), а также получить сам файл сертификата и подписывать им SIS-файлы самостоятельно. Более подробно процедура получения идентификатора издателя и сервис Open Signed Offline описаны в *главе 7* “Сертификация приложений”.

Обычный сертификат, не имеющий ограничений по IMEI и открывающий доступ ко всем необходимым защищенным возможностям, на руки не выдается. Подписать им ваше приложение можно будет, только воспользовавшись одной из программ сертификации Symbian Signed. Для этого также потребуются идентификатор издателя. Кроме того, чем более важные защищенные возможности декларируются приложением, тем более требовательную и дорогостоящую программу сертификации придется пройти. Подробнее об этом рассказано также в *главе 7* “Сертификация приложений”.

Приложения, устанавливаемые при помощи SIS-пакетов, подписанных сертификатом разработчика или обычным сертификатом, относятся к категории надежных.

Подготовка к сертификации ASD

- Знание того, что ПО, подписанное сертификатом-пустышкой и не использующее защищенных служб системы, не является доверенным (untrusted). Оно может устанавливаться и исполняться на устройстве с рядом ограничений (sandboxed).
 - Понимание взаимосвязи между TCB/TCE, декларацией доступа к защищенным возможностям, установщика приложений как “шлюза” и сертификации приложений.
 - Понимание общих принципов процесса тестирования и публикации сертифицированных приложений для Symbian 9.x.
-

ГЛАВА 2

Структура проекта на Symbian C++

Проект приложения на Symbian C++ содержит исходный код, заголовочные файлы, различные ресурсы и данные, необходимые для работы программы, а также файлы настроек. Все вышеперечисленное может находиться в одном каталоге, но на практике используется следующая структура подкаталогов:

- `\data\` — файлы ресурсов (`“.rss”`, `“.rssi”`, `“.loc”`);
- `\inc\` — заголовочные файлы исходного кода (`“.h”`);
- `\src\` — исходный код (`“.cpp”`);
- `\gfx\` — изображения (`“.bmp”`, `“.svg”`);
- `\group\` — настройки проекта (`bld.inf`, `“.mmp”`, `“.mk”`);
- `\sis\` — скрипты для сборки установочного SIS-пакета (`“.pkg”`).

Рекомендуется всегда придерживаться подобной схемы расположения файлов проекта. Именно такую структуру каталогов используют при создании нового проекта и IDE Carbide.c++.

Файл `bld.inf`

В каталоге `\group\` хранятся файл `bld.inf`, определяющий формирующие проект компоненты, и набор файлов с расширением `“.mmp”`, описывающих каждый из компонентов проекта. Компонентами проекта могут быть исполняемые файлы либо библиотеки. В проекте может быть только один файл `bld.inf`. На основании информации из файла `bld.inf` средствами SDK создается командный файл `abld.bat`, содержащий инструкции для сборки проекта.

Файл `bld.inf` является текстовым ANSI-файлом, содержащим следующие секции: `prj_platforms`, `prj_mmpfiles`, `prj_testmmpfiles`, `prj_exports` и `prj_testexports`. Последние три из них довольно редко используются на практике. Порядок объявления секций в файле произвольный, некоторые секции могут отсутствовать или объявляться несколько раз. Вот пример стандартного файла `bld.inf`.

```
PRJ_PLATFORMS
DEFAULT
```

```
PRJ_MMPFILES
gnumakefile icons_aif_scalable_dc.mk
helloworld.mmp
```

Секция **PRJ_PLATFORMS** содержит список **целевых платформ** (target platforms), для которых может собираться проект. Формат использования такой.

```
prj_platforms list_of_platforms
```

Здесь параметр *list_of_platforms* — перечень целевых платформ, разделенных пробелами или начинающихся с новой строки. Чаще всего целевыми платформами являются **WINSCE**, **ARMV5** и **GCCE**. Более подробно о назначении целевых платформ будет рассказано в *главе 3*, “Работа с SDK”. Секция **PRJ_PLATFORMS** также допускает использование псевдоплатформы **DEFAULT** для обозначения любой допустимой платформы и необязательного перечня платформ с префиксом “-” перед именем — для их исключения из этого множества. Этот прием может быть довольно полезен в том случае, если проект не может быть собран под определенные платформы (например, ввиду отсутствия необходимых для сборки библиотек). В данном примере исключена сборка под эмулятор Windows.

```
PRJ_PLATFORMS DEFAULT -WINSCE
```

В секции **prj_mmpfiles** объявляются компоненты, формирующие проект. Эта секция имеет следующий формат.

```
prj_mmpfiles
mmp_file_1 [build_as_arm] [tidy]
mmp_file_n [build_as_arm] [tidy]
makefile makefile_1 [build_as_arm] [tidy]
makefile makefile_n [build_as_arm] [tidy]
nmakefile makefile_1 [build_as_arm] [tidy]
nmakefile makefile_n [build_as_arm] [tidy]
gnumakefile makefile_1 [build_as_arm] [tidy]
gnumakefile makefile_n [build_as_arm] [tidy]
```

Элементами секции являются ссылки на содержащиеся в проекте ММР-файлы и дополнительные **сборочные скрипты** (extension makefiles). Ссылки могут содержать относительный путь к файлу, если он не находится в одном каталоге с файлом `bld.inf`. Ключевые слова `makefile` (`nmakefile`) и `gnumakefile` определяют, какой утилитой SDK (`nmake` или `make`) будет исполняться скрипт `makefile`. Чаще всего подобные скрипты находятся в каталоге `\group\`, имеют расширение “.mk” и используются для автоматической конвертации изображений в формат MIF при сборке проекта. Отличительной особенностью скрипта `makefile` от командного файла Windows является возможность выполнения той или иной его части в зависимости от параметра, с которым вызвана утилита `make` на данном этапе сборки. Более подробно синтаксис сборочных скриптов описан в справочниках SDK.



Обычно файл `bld.inf` содержит либо один MMP-файл (определяющий исполняемый файл или библиотеку), либо пару: MMP-файл и МК-скрипт для создания пиктограммы. Но в ряде проектов компонентов может быть больше. Например, в проектах клиент-серверных приложений в файле `bld.inf` можно встретить и сервер, и экспортирующую методы доступа к серверу клиентскую библиотеку, и само клиентское приложение. В таких случаях важна последовательность объявления компонентов в секции `prj_mmpfiles`, так как именно она определяет порядок их сборки.

Необязательный атрибут `tidy` помечает компонент как не используемый в релизе проекта. Результат сборки такого компонента будет автоматически удален после сборки всего проекта. Атрибут `build_as_arm` используется только при сборке проекта для целевой платформы ARMV5 и указывает на необходимость компиляции программы или библиотеки для набора инструкций ARM вместо THUMB. Более подробно платформы ARM и THUMB рассматриваются в главе 3.

Также в секции `prj_mmpfiles` возможно использование следующего условного блока.

```
#if defined(<target_platform>)
<mmp_file>
#endif
```

Это позволяет *исключать* некоторые компоненты проекта при сборке под определенные целевые платформы.

Синтаксис секции **`prj_testmmpfiles`** совпадает с синтаксисом секции `prj_mmpfiles` за исключением необязательных атрибутов: в секции отсутствует атрибут `build_as_arm` и введены два новых необязательных атрибута — `manual` и `support`. Компоненты, объявленные в `prj_testmmpfiles`, не должны включаться в `prj_mmpfiles`. Различие между этими секциями заключается в том, что для компонентов, объявленных в секции `prj_testmmpfiles`, после сборки создаются командные файлы для автоматического или ручного тестирования. Дополнительную информацию об этой секции можно получить в справочнике SDK.

Секция **`prj_exports`** позволяет после сборки проекта автоматически копировать файлы из каталога проекта куда-либо. Она имеет следующий формат.

```
prj_exports
source_file_1 [destination_file]
source_file_n [destination_file]
:zip zip_file [destination_path]
```

Исходное и новое имена копируемого файла могут содержать относительные пути. При этом для исходного имени этот путь будет определяться от каталога, в котором находится файл `bld.inf`, а для нового имени — от каталога `\epoc32\include\` в SDK. Это позволяет копировать заголовочные файлы проекта в соответствующий каталог текущей SDK и использовать их как системные

в исходном коде других проектов. В случае, если новое имя файла содержит букву диска, то путь будет определяться от каталога `\epoc32\data\ SDK`, например:

```
PRJ_EXPORTS
Mydata.dat e:\data\appdata.dat
```

В результате файл `mydata.dat` после сборки проекта будет скопирован в каталог `\epoc32\data\e\appdata\` — на диск E: эмулятора SDK.

Директива `:zip` указывает на необходимость распаковать архив *zip_file* в каталог *destination_path* (он должен существовать).

Секция **prj_testexports** не поддерживает директиву `:zip` и отличается от секции `prj_exports` тем, что относительные пути и для исходного, и для нового имени файла определяются от каталога, в котором находится файл `bld.inf`.

ММР-файлы

ММР-файл содержит служебную информацию для сборки компонента: имя файла-результата, его идентификаторы, список используемых системных и пользовательских библиотек, ссылки на необходимые файлы исходного кода и ресурсов. Помимо этого, в ММР-файле декларируется использование тех или иных защищенных возможностей. Проект может содержать несколько ММР-файлов. Во время сборки на основе ММР-файлов и выбранной целевой платформы генерируются скрипты для компиляции компонентов.

ММР-файл, как и файл `bld.inf`, является текстовым и состоит из ключевых выражений с параметрами. Каждое ключевое выражение должно начинаться с новой строки. Часть из них обязательна и может появляться в файле лишь один раз, другая часть необязательна и может использоваться сколько угодно раз. Порядок ключевых выражений в ММР-файле не важен. Параметры ключевых выражений записываются в той же строке через пробел. В случае, если используется несколько параметров, все они разделяются пробелами. Вот пример небольшого ММР-файла.

```
TARGET                helloworld.exe
TARGETTYPE            exe
UID                   0 0xE21F6D60

USERINCLUDE           ..\inc
SYSTEMINCLUDE         \epoc32\include

SOURCEPATH            ..\src
SOURCE                helloworld.cpp

LIBRARY               euser.lib
```

Формат ММР-файла довольно сложен, и вам нет необходимости запоминать все ключевые выражения. В IDE Carbide.c++, которую мы будем рассматривать в главе 4, имеется визуальный редактор для файлов `bld.inf` и ММР-файлов. Тем не менее именно в терминах ключевых выражений обсуждаются вопросы, связанные с ММР-файлами, как в документации, так и на многочисленных форумах. В табл. 2.1 приводится описание основных ключевых выражений, используемых в ММР-файлах.

Таблица 2.1. Основные ключевые выражения, используемые в ММР-файлах

Ключевое выражение	Описание
<code>TARGET filename.ext</code>	Задаёт имя скомпилированного файла
<code>TARGETTYPE target-type</code>	Указывает на тип компилируемого файла. По сути, это способ задания значения UID1 при помощи псевдонима. Задать UID1 явно нельзя. Например, <code>TARGETTYPE EXE</code> устанавливает значение UID1 равным <code>0x1000007A</code> , а <code>TARGETTYPE DLL</code> — равным <code>0x10000079</code> . Полный список всех возможных значений параметра <code>target-type</code> (больше двух десятков) вы можете найти в справочнике SDK
<code>TARGETPATH target-path</code>	Позволяет изменить каталог, в который помещается результат компиляции компонента проекта. В случае отсутствия этого ключевого выражения используется каталог <code>SDK \epoc32\release\platform\variant\</code> , где <code>platform</code> — целевая платформа, <code>variant</code> — режим компиляции. Если параметр <code>target-path</code> задан, то результат будет помещён в подкаталог <code>\epoc32\release\platform\variant\z\target-path\</code>
<code>UID [uid2] [uid3]</code>	Позволяет задать идентификаторы UID2 и UID3. Если это ключевое выражение опущено, то оба идентификатора будут нулевыми. Пример использования: <code>UID 0 0xE21F6D60</code>
<code>SECUREID secureid</code>	Позволяет задать SID приложения. Пример использования: <code>SECUREID 0xE21F6D60</code> . В случае если SID не задан, в качестве него используется значение UID3. Задавать SID, отличный от UID3, не рекомендуется
<code>VENDORID vendorid</code>	Устанавливает идентификатор производителя (VID) файла. По умолчанию используется нулевое значение VID
<code>CAPABILITY capability-names</code>	Является декларацией списка защищённых возможностей, доступ к которым осуществляется данным исполняемым файлом или библиотекой. Защищённые возможности перечисляются через пробел, например: <code>CAPABILITY Location ProtServ SwEvent</code> . В качестве значения параметра <code>capability-names</code> также можно использовать слово <code>ALL</code> (все защищённые возможности платформы безопасности) и <code>NONE</code> (пустое множество). Вместе с <code>ALL</code> можно перечислить несколько

Продолжение табл. 2.1

Ключевое выражение	Описание
	<p>возможностей с символом “-” в качестве префикса для исключения их из списка. Пример декларации всех защищенных возможностей, кроме группы возможностей производителя.</p> <p>CAPABILITY ALL -TCB -AllFiles -DRM</p> <p>В случае, если ключевое выражение CAPABILITY в MMP-файле не встречается, при компиляции используется значение CAPABILITY NONE</p>
LANG <i>language-list</i>	<p>Позволяет задать список поддерживаемых проектом локализаций. Язык локализации задается двузначным числом. Пример:</p> <p>LANG 01 16</p> <p>Код 01 — UK English, 16 — Russian. По умолчанию ключевое выражение LANG имеет значение SC, что обозначает отсутствие необходимости в локализации проекта. Более подробно о локализации приложений будет рассказано ниже в этой главе в разделе “Локализация и компиляция файла ресурса” и в главе 3, раздел “Создание дистрибутива приложения”</p>
LIBRARY <i>filename-list</i>	<p>Эти выражения служат для подключения используемых библиотек. Здесь ключевое выражение LIBRARY объявляет список динамически подключаемых библиотек, STATICLIBRARY — список статически подключаемых библиотек, а DEBUGLIBRARY отличается от LIBRARY лишь тем, что она игнорируется во всех режимах компиляции, кроме DEBUG. Параметр <i>filename-list</i> может содержать как одну, так и несколько разделенных пробелами имен файлов библиотек (порядок не важен). Пример:</p> <p>LIBRARY euser.lib efsrv.lib</p> <p>Все три ключевых выражения могут использоваться многократно и в любом порядке</p>
STATICLIBRARY <i>filename-list</i>	
DEBUGLIBRARY <i>filename-list</i>	
SYSTEMINCLUDE <i>directory-list</i>	<p>Эти ключевые выражения задают каталоги, в которых выполняется поиск подключенных к исходному коду заголовочных файлов.</p>
USERINCLUDE <i>directory-list</i>	<p>Ключевое выражение SYSTEMINCLUDE определяет каталог, в котором хранятся заголовочные файлы системных API. В подавляющем большинстве случаев таким каталогом является \epoc32\include\ SDK. Ключевое выражение USERINCLUDE указывает на каталоги, содержащие пользовательские заголовочные файлы. Чаще всего их хранят в каталоге \inc\, в соответствии с описанной ранее структурой каталогов проекта. Учитывая то, что сам MMP-файл находится в каталоге \group\, ссылка на каталог \inc\ будет иметь следующий вид.</p>

Продолжение табл. 2.1

Ключевое выражение	Описание
	<p><code>USERINCLUDE ..\inc\</code></p> <p>Примечательно, что пользовательские заголовочные файлы ищутся сначала в том же каталоге, в котором находится исходный код, и лишь затем в каталогах, заданных ключевым выражением <code>USERINCLUDE</code>. Если и там нужных файлов не оказалось, то поиск проводится в каталогах, указанных в ключевом выражении <code>SYSTEMINCLUDE</code>.</p> <p>Подключение заголовочных файлов в коде осуществляется с помощью директивы <code>#include</code>. При этом имя системных файлов заключается в угловые скобки, а пользовательских — в кавычки. Пример.</p> <p>Системный файл:</p> <pre>#include <e32def.h></pre> <p>Пользовательский файл:</p> <pre>#include "myclass.h"</pre>
<code>SOURCEPATH directory</code>	<p>Задаёт каталог, в котором находятся исходный код, ресурсы или изображения. Значение этого ключевого выражения влияет на пути поиска всех последующих объявлений этих данных, вплоть до следующего объявления <code>SOURCEPATH</code>. Единственный параметр <i>directory</i> может быть как относительным путем (определяется от каталога, в котором находится MMP-файл), так и полным (определяется от каталога, заданного в переменной окружения <code>EPOCROOT</code>)</p>
<code>SOURCE source-file-list</code>	<p>Задаёт файлы с исходным кодом, используемые данным компонентом проекта. Список <i>source-file-list</i> представляет собой перечисление имен файлов с расширениями <code>".cpp"</code>, разделенных пробелами. Файлы ищутся в каталоге, заданном предшествующим ключевым выражением <code>SOURCEPATH</code>. Такой подход позволяет двум компонентам проекта хранить файлы с исходным кодом в одном каталоге. Будьте внимательны: не забывайте подключать CPP-файлы и не подключайте один и тот же файл дважды — это приведет к ошибкам компиляции</p>
<code>START RESOURCE source-file</code> <code>[target target-file-name]</code> <code>[targetpath targetpath]</code> <code>[header]</code> <code>[lang languages]</code> <code>[uid uid-value-1 [uid-value-2]]</code> <code>END</code>	<p>Структура, описывающая правила компиляции файла ресурса (RSS). Параметр <i>source-file</i> задает имя файла и может включать относительный путь к нему. Файл ищется в каталоге, указанном предшествующим ключевым выражением <code>SOURCEPATH</code>.</p> <p>Значение параметра <i>target-file-name</i> задает имя файла после компиляции, не содержит расширение (оно добавляется автоматически) и по умолчанию совпадает с именем, задаваемым параметром <i>source-file</i>.</p>

Ключевое выражение	Описание
	<p>Директива <code>targetpath</code> задает каталог, в который помещается скомпилированный ресурс, причем путь указывается относительно диска <code>z</code> эмулятора SDK (каталог <code>\epoc32\data\z\</code>). В случае если он опущен, используется значение ключевого выражения <code>TARGETPATH</code> для всего MMP-файла.</p> <p>Присутствие директивы <code>header</code> сигнализирует о необходимости создания так называемого “заголовочного файла ресурса” (resource header file). Это файл с расширением <code>.rsg</code>, содержащий объявление идентификаторов-индексов, для находящихся в файле ресурса структур.</p> <p>Директива <code>UID</code> позволяет задать значения <code>UID2</code> и <code>UID3</code> скомпилированного ресурса. По умолчанию <code>UID2</code> нулевой, а значением <code>UID3</code> является 20-битовый хеш имени файла. Значения <code>UID2</code> и <code>UID3</code> могут быть также переопределены в самом файле ресурса. <code>UID1</code> ресурса всегда равен <code>0x101F4A6B</code>.</p> <p>Директива <code>LANG</code> позволяет переопределить настройки локализации MMP-файла для данного файла ресурсов. Расширение скомпилированного файла ресурса имеет вид <code>.rXX</code>, где <code>XX</code> — двузначный код языка локализации либо <code>sc</code>. Файл ресурсов компилируется столько раз, сколько локализаций он имеет</p>
<pre>START BITMAP target-file [targetpath targetpath] [header] [sourcepath sourcepath] source color-depth source-bitmap-list END</pre>	<p>Структура <code>BITMAP</code> описывает правила компиляции изображений формата BMP и SVG в формат MBM (multi-bitmap file). В отличие от структуры файла ресурсов, в ее заголовке указывается не файл-источник, а имя файла результата с расширением <code>.mbm</code>. Компилируемые в него изображения задаются с помощью директив <code>sourcepath</code> и <code>source</code>, подобно файлам исходного кода. При этом значение директивы <code>sourcepath</code> имеет влияние лишь в пределах этой структуры. Список <code>bitmap-list</code> может содержать одно или несколько имен файлов изображений через пробел. Изображения на четных позициях этого списка играют роль маски для изображений, стоящих на нечетных позициях. Все изображения и маски в одном списке должны иметь одинаковую глубину цвета (color depth), задающуюся обязательным параметром <code>color-depth</code> в формате <code>[c] digit, digit</code>. Присутствие префикса <code>c</code> свидетельствует о том, что изображение цветное, а следующие за ним два числа являются глубиной цвета изображения и его маски соответственно. В качестве глубины цвета маски рекомендуется использовать значение <code>1</code> (занимает меньше памяти). Вот небольшой пример объявления</p>

Окончание табл. 2.1

Ключевое выражение	Описание
	<p>MBM-файла, содержащего три изображения, одно из которых является маской первого:</p> <pre>START BITMAP icons.mbm HEADER TARGETPATH \resource\apps SOURCEPATH ..\gfx SOURCE c12,1 icon.bmp icon_mask.bmp SOURCE c12 just_image.bmp END</pre>
<p>EPOCHEAPSIZE <i>minimum</i> <i>maximum</i></p> <p>EPOCSTACKSIZE <i>stacksize</i></p>	<p>Ключевое выражение EPOCHEAPSIZE задает минимальный и максимальный размер памяти, отводимой под кучу процесса. По умолчанию это 4 Кбайт и 1 Мбайт соответственно. При старте программы куча имеет минимальный размер. Затем, при необходимости, ее размер увеличивается. Так как увеличивается он за счет выделения новой страницы памяти, то каждое значение в ключевом выражении EPOCHEAPSIZE округляется до кратного размеру страницы (4 Кбайт). Ключевое выражение EPOCSTACKSIZE задает размер стека главного потока процесса в байтах. По умолчанию он равен 8 Кбайт и может быть увеличен до 80 Кбайт. Будьте внимательны, для работы GUI приложения требуется стек размером не менее 20 Кбайт. Если он окажется меньше, программа завершится аварийно. Значения ключевых выражений EPOCHEAPSIZE и EPOCSTACKSIZE могут быть заданы как в десятичной, так и в шестнадцатеричной системе счисления (последнее предпочтительнее). Пример:</p> <pre>EPOCSTACKSIZE 0x5000</pre> <p>Оба ключевых выражения игнорируются при сборке для целевых платформ WINS/WINSCW</p>
EPOCPROCESSPRIORITY <i>priority</i>	<p>Приоритет процесса приложения. Может принимать значения low, background, foreground, high, windowserver, filesaver, realtimeserver или supervisor. По умолчанию — foreground. Используется только в исполняемых файлах. Игнорируется при сборке для целевых платформ WINS/WINSCW</p>
COMPRESSTARGET INFLATECOMPRESSTARGET BYTEPAIRCOMPRESSTARGET NOCOMPRESSTARGET	<p>Эти ключевые выражения определяют метод, которым будут сжаты секции кода и данных в скомпилированном файле. Лишь одно из них должно присутствовать в MMP-файле. По умолчанию это COMPRESSTARGET (deflate-вариант алгоритма Хаффмана +LZ77)</p>

В табл. 2.1 описаны лишь основные ключевые выражения, используемые в MMP-файлах. Полный их список можно найти в справочнике SDK.

Как и в случае файла `bld.inf`, формат MMP-файла допускает использование условного блока `#if defined`, позволяющего изменять настройки компонента проекта в зависимости от целевой платформы сборки.

Подготовка к сертификации ASD

- Знание синтаксиса и основных ключевых выражений MMP-файла.
 - Знание того, как задаются VID и SID в MMP-файле.
 - Знание того, как декларируется доступ к защищенным возможностям в MMP-файле.
-

Файлы ресурсов и локализация проекта

Файлы ресурсов в Symbian OS являются хранилищем данных и содержат записи ресурсов различной структуры. Ресурсы в приложении выполняют три функции:

- разделение кода и данных;
- регистрация различной информации в системе;
- локализация данных.

Многие элементы пользовательского интерфейса и диалоговые окна имеют методы, позволяющие им быть инициализированными при помощи ресурсов. Естественно, их структура при этом должна соответствовать определенному формату. Такие методы принимают идентификатор ресурса в качестве аргумента, разбирают его поля и инициализируют объект полученными значениями. Например, в листинге 2.1 приведен текст ресурса, позволяющий создать диалоговое окно с двумя озаглавленными полями редактирования для ввода текста.

Листинг 2.1. Ресурс создания диалогового окна с двумя полями ввода

```
RESOURCE_DIALOG r_gui_container_multi_query1
{
    flags = EAknGeneralQueryFlags;
    buttons = R_AVKON_SOFTKEYS_OK_CANCEL;
    items =
    {
        DLG_LINE
        {
            type = EAknCtMultilineQuery;
            id = 1;
            control = AVKON_DATA_QUERY
            {
                layout = EMultiDataFirstEdwin;
                label = "Enter data1:";
            }
        }
    }
}
```

```

        control = EDWIN
        {
            maxlength = 50;
            default_case = EAknEditorTextCase;
        };
    },
    DLG_LINE
    {
        type = EAknCtMultilineQuery;
        id = 2;
        control = AVKON_DATA_QUERY
        {
            layout = EMultiDataSecondEdwin;
            label = "Enter data2:";
            control = EDWIN
            {
                maxlength = 50;
                default_case = EAknEditorTextCase;
            };
        };
    }
};
}

```

Выглядит устрашающе, не правда ли? А ведь этот пример сильно упрощен. В исходном коде использование этого ресурса для инициализации объекта выглядит следующим образом.

```

// Объявление двух дескрипторов для хранения
// введенных пользователем значений
TBuf<50> data1; TBuf<50> data2;
// Создание экземпляра диалога
CAknMultiLineDataQueryDialog* queryDialog =
    CAknMultiLineDataQueryDialog::NewL( data1, data2 );
// Инициализация диалога из ресурса и его отображение
queryDialog->ExecuteLD( R_GUI_CONTAINER_MULTI_QUERY1 );

```

Это гораздо проще, чем вызов более десятка методов для настройки внешнего вида экземпляра диалогового окна. Такой подход позволяет разделить код и данные и применять различные инструменты разработки для их редактирования. Например, среда разработки Carbide.c++ имеет специальный визуальный проектировщик графического интерфейса (UI Designer), позволяющий настроить все параметры внешнего вида элементов и генерирующий объявления ресурсов необходимой структуры для их инициализации.

Некоторые файлы ресурсов программы используются самой Symbian OS, например, для регистрации приложения в определенных подсистемах (отображение в главном меню, автозапуск и пр.).

Файл ресурсов также может хранить текстовые записи, которые впоследствии могут быть использованы в элементах управления и диалоговых окнах. Это позволяет создавать несколько локализованных файлов ресурсов и переключать их в зависимости от выбранного в программе или системе языка.

Ресурсы проекта хранятся в каталоге `\data\` и имеют расширение `".rss"`. Файлы RSSI являются частями RSS-файла и подключаются к нему с помощью директивы `#include "filename.rssi"`. Во время компиляции ресурса содержимое RSSI-файлов вставляется в файл RSS. Будьте внимательны: в ряде случаев важно соблюсти порядок объявления ресурсов в файле, поэтому подключение RSSI-компонентов рекомендуется выполнять в конце RSS-файла.

Объявление структуры ресурса

Файл RSS содержит записи ресурсов различной структуры. Прежде чем использовать структуру для объявления ресурса, она сама должна быть объявлена. Структуры ресурсов объявляются следующим образом.

```
STRUCT <struct-name> [ BYTE | WORD ]
{
    <struct-member-list>
}
```

Параметр `<struct-name>` является именем структуры. Оно должно состоять только из заглавных букв, не может начинаться с цифры и совпадать с названием одного из 13 возможных типов данных полей структуры. При наличии в объявлении структуры ключевых слов `BYTE` или `WORD` к записи ресурса такой структуры при компиляции будет добавлен префикс (байт или машинное слово соответственно) с информацией о его размере.

Параметр `<struct-member-list>` является списком полей структуры. Описание каждого поля должно оканчиваться точкой с запятой и иметь следующий формат.

```
[ LEN [ BYTE ] ] <type-name> <member-name>
[ ( <length-limit> ) ] [ [ [ <array-size> ] ] ]
```

В простейшем случае объявление поля структуры имеет только тип данных (параметр `<type-name>`) и имя поля (параметр `<member-name>`). Допустимо использование следующих типов данных (обязательно заглавными буквами).

- `BYTE` — байт.
- `WORD` — машинное слово (два байта).
- `LONG` — двойное слово (четыре байта).
- `DOUBLE` — 8-байтовое число с плавающей точкой.
- `BUF` — UNICODE-строка.
- `BUF<n>` — UNICODE-строка максимальной длины *n*.
- `BUF8` — строка 8-битовых символов.

- TEXT — оканчивающаяся нулевым байтом (null-terminated) строка. Тип не рекомендуется к использованию.
- LTEXT — UNICODE-строка, содержащая байт-префикс с ее размером.
- SRLINK — 32-битовый идентификатор ресурса. Не должен инициализироваться, так как значение присваивается компилятором ресурсов.
- LINK — 16-битовый идентификатор. Используется как ссылка на другой ресурс.
- LLINK — 32-битовый идентификатор. Используется как ссылка на другой ресурс.
- STRUCT — ресурс произвольной структуры. Позволяет использовать вложенные записи ресурсов. К сожалению, при использовании одного ресурса в качестве члена другого ресурса проверка его структуры не выполняется. Поэтому разработчики в комментариях указывают, ресурсы какой структуры могут стать членом объявляемой структуры.

Типы BYTE, WORD и LONG могут принимать как знаковые, так и беззнаковые значения.

Параметр *<length-limit>* является способом задания максимальной длины строки для BUF, TEXT и LTEXT.

Квадратные скобки после имени поля обозначают массив данного типа. Число *<array-size>* в таких скобках задает массив фиксированной длины.

Пример объявления простой структуры, содержащей информацию о пользователе.

```
STRUCT USER_INFO
{
    BUF<20> name;
    BYTE age;
    STRUCT friends[]; // USER_INFO
}
```

Предполагается, что поле name будет хранить имя пользователя, age — возраст, а friends является массивом записей произвольной структуры и будет содержать список друзей пользователя. В комментариях я указал, какой именно структуры должны быть элементы массива friends. Если разработчик ошибется и поместит в него записи другой структуры, файл ресурса все равно успешно скомпилируется, но это может привести к ошибке во время использования ресурса.

Вы также можете задать значения по умолчанию для полей структуры. Например, зададим имя пользователя по умолчанию.

```
STRUCT USER_INFO
{
    BUF<20> name = "Unknown";
    BYTE age;
    STRUCT friends[]; // USER_INFO
}
```

Если значения по умолчанию в объявлении структуры не указаны, то будут использованы пустые строки и нули для числовых полей.

Объявления структур ресурсов обычно содержатся в специальных заголовочных файлах ресурсов с расширением “.rh”. Они подключаются к ресурсу типа “.rss” или “.rssi” так же, как и заголовочные файлы исходного кода — с помощью выражения `#include`. Например:

```
#include <eikon.rh>
#include "users.rh"
```

В первом случае подключается системный заголовочный файл ресурса, во втором — пользовательский. Принципиальная разница между системными и пользовательскими заголовочными файлами лишь в определении их местоположения¹.

Объявление ресурса

После того как в RSS или RSSI подключены необходимые заголовочные файлы, можно объявлять в них ресурсы. Запись ресурса имеет следующий формат.

```
RESOURCE <struct-name> [<resource-name>]
{
    <resource-initialiser-list>
}
```

Здесь параметр `<struct-name>` — название структуры, формату которой соответствует ресурс. Если вы забудете подключить заголовочный файл с объявлением этой структуры, то получите ошибку во время компиляции RSS-файла. Параметр `<resource-name>` — название ресурса, **обязательно должно быть записано в нижнем регистре**. Название ресурса служит для его использования в качестве элемента другого ресурса или инициализации полей `LINK` и `LLINK`. Помимо этого, на основании имен ресурсов генерируется файл RSG, назначение которого мы обсудим несколько позднее. Параметр `<resource-initialiser-list>` — блок инициализации элементов ресурса.

Итак, простейшая запись ресурса объявленной нами структуры `USER_INFO` будет выглядеть следующим образом.

```
RESOURCE USER_INFO myresource1
{
}
```

В этом случае поля записи инициализируются значениями по умолчанию, и фактически она будет хранить следующую информацию.

```
name = unknown, age = 0, friends = NULL
```

¹ См. описание директив `SYSTEMINCLUDE` и `USERINCLUDE` MMP-файла в табл. 2.1.

Теперь в этом же файле запишем ресурс той же структуры, но с явным заданием значений полей.

```
RESOURCE USER_INFO myresource2
{
    name = "Aleksandr";
    age = 25;
    friends =
        {myresource1, myresource1};
}
```

В данном случае запись соответствует пользователю Aleksandr, 25-ти лет, имеющего двух друзей (на самом деле одного, но для простоты я использовал имя первой записи дважды). Если вы попытаетесь инициализировать в ресурсе поля, не объявленные в его структуре, это приведет к ошибке во время компиляции ресурса.

Теперь немного усложним пример, изменив структуру USER_INFO. Добавим новое поле loc, содержащее географические координаты местонахождения пользователя. Для этого нам потребуется массив действительных чисел фиксированной длины. Пусть он имеет значение по умолчанию, отличное от нулевого.

```
STRUCT USER_INFO
{
    BUF<20> name = "Unknown";
    BYTE age;
    STRUCT friends[]; // USER_INFO
    DOUBLE loc[2] =
        {53.2091, 50.2854}; // user default location
}
```

В случае если элементы массива структур нигде больше не используются, мы можем объявить их непосредственно в самом массиве.

```
RESOURCE USER_INFO myresource2
{
    name = "Aleksandr";
    age = 25;
    friends =
    {
        USER_INFO
        {
            name = "Viktor";
            age = 18;
            friends =
                {myresource2};
            loc =
                {56.3127, 44.0174};
        },
        USER_INFO
```

```

    {
        name = "Oleg";
        age = 27;
        friends =
            {myresource2};
        loc =
            {55.0293, 82.9677};
    }
};
}

```

Таким же образом задаются и значения по умолчанию для массивов структур, но продемонстрировать это не удастся, так как мы условились, что массив хранит структуры `USER_INFO`, а мы не можем использовать имя этой структуры в собственном объявлении.

В файле ресурса можно использовать директиву `#define` для задания псевдонимов часто используемым константам. Такие псевдонимы можно использовать как в объявлении структур, так и в определении ресурсов.

```

#define max_username 20

STRUCT USER_INFO
{
    BUF<max_username> name = "Unknown";
    BYTE age;
    STRUCT friends[]; // USER_INFO
    DOUBLE loc[2] = {53.2091, 50.2854};
}

```

Идентификаторы ресурсов

В предыдущем разделе мы использовали имена ресурсов для инициализации полей других ресурсов. Но что делать, если ресурс был объявлен в другом файле ресурсов? В этом случае вместо имени ресурса вы можете использовать его идентификатор. Каждый именованный ресурс в RSS-файле имеет свой идентификатор, соответствующий его порядковому номеру. Такие идентификаторы задаются с помощью директивы `#define` в отдельном файле с расширением `“.rsg”`. Вот как будет выглядеть RSG-файл для нашего примера.

```

#define MYRESOURCE1 1
#define MYRESOURCE2 2

```

Чтобы не путать имена ресурсов и их идентификаторы, первые всегда задаются в нижнем регистре, а последние — в верхнем. Для ресурсов без имени (например, элементы массива в последнем примере) идентификаторы не задаются. RSG-файл обычно не создается пользователем, а генерируется автоматически во время компиляции ресурса. Для этого необходимо в MMP-файле добавить ключевое выражение `HEADER` в объявление файла ресурсов. Полученный RSG-

файл с именем файла ресурсов появится в каталоге, заданном ключевым выражением `SYSTEMINCLUDE`.

Если в проекте используются несколько RSS-файлов, идентификаторы содержащихся в них ресурсов могут совпасть. Чтобы избежать этого, в каждом RSS-файле проекта объявляется собственный идентификатор файла ресурсов. Идентификатор файла задается после выражения `NAME` и содержит от одной до четырех латинских букв в верхнем регистре. По сути, выражение `NAME` — альтернативный способ задания `UID3` файла. При компиляции символьный идентификатор файла интерпретируется в виде числа и добавляется в качестве префикса (первые 20 бит) к идентификаторам ресурсов файла. Это также означает, что идентификатор файла должен быть задан раньше любого, объявляемого в нем ресурса. Например, добавим следующую директиву в наш RSS-файл.

```
NAME USER
```

В этом случае в сгенерированном RSG-файле получим.

```
#define MYRESOURCE1      0x68553001
#define MYRESOURCE2      0x68553002
```

Здесь `0x68553` — 20-битовый идентификатор файла, полученный из слова `USER`.

Идентификаторы ресурсов используются и в исходном коде программы, так как это единственный способ обратиться к ресурсам после их компиляции. Файл RSG может быть подключен к файлу ресурсов или файлу исходного кода с помощью директивы `#include`. При этом на него распространяются все те же правила определения местоположения, что и на прочие подключаемые заголовочные файлы.

Перечисления в файлах ресурсов

В ресурсах часто используются различные флаги, а также идентификаторы элементов. Например, в самом начале раздела вы можете найти объявление ресурса диалогового окна `r_gui_container_multi_query1`, содержащее множество именованных значений для управления внешним видом этого окна. Для этих целей используются псевдонимы и перечисления. С объявлениями псевдонимов мы уже сталкивались в RSG-файле, они задаются при помощи директивы `#define`, например:

```
#define EAknEditorTextCase      0x4
```

Синтаксис объявления перечислений в файлах ресурсов совпадает с синтаксисом объявления перечислений в C++ и выглядит следующим образом.

```
enum [<enum-label>]
{
    <enum-list>
};
```

Здесь параметр `<enum-list>` — список значений перечисления, разделенных запятыми. Значения перечисления имеют вид: `<member-name> [= <initializer>]`. В случае если инициализатор явно не задан, то используется значение предыдущего элемента списка, увеличенное на единицу. Значение первого элемента перечисления по умолчанию нулевое, например:

```
enum TGuiContainerViewControls
{
    EGuiContainerViewEdit1 = 1,
    EGuiContainerViewEdit2
};
```

Объявленные константы и перечисления могут многократно использоваться в файлах ресурсов и в исходном коде приложения. Поэтому их помещают в отдельные заголовочные файлы с расширением `“.hrh”`, которые затем подключаются с помощью директив `#include`.

Прочие выражения файлов ресурсов

Помимо вышеописанного, в состав файла ресурса могут входить еще три выражения: `CHARACTER_SET`, `UID2` и `UID3`. Первое позволяет указать кодировку, в которой задано содержимое файла: либо `CP1252` (по умолчанию), либо `UTF8`. Пример:

```
CHARACTER_SET UTF8
```

Выражения `UID2` и `UID3` позволяют задать соответствующие идентификаторы файла. `UID1` файла ресурса всегда равен `0x101F4A6B`. По умолчанию `UID2` нулевой, а `UID3` хранит 20-битовый идентификатор, заданный выражением `NAME`. В редких случаях (для специализированных файлов ресурсов) им требуется присвоить предопределенные значения. Например, в регистрационном файле ресурсов они имеют следующий вид.

```
UID2 KUidAppRegistrationResourceFile // Константа из appinfo.rh
UID3 0x10001234 // Должен быть равен UID3 исполняемого файла
```

Локализация и компиляция файла ресурса

Для того чтобы уменьшить размер файла ресурса, он компилируется с помощью входящей в состав SDK утилиты `gcomp` в бинарный файл. Формат бинарного файла ресурса описан в справочнике SDK.

Как уже отмечалось выше, использование в программе подключаемых ресурсов позволяет изменять язык ее интерфейса и даже внешний вид с помощью простой замены файла ресурса. Symbian OS позволяет автоматизировать этот процесс. Во время запуска GUI приложения операционная система (а точнее, подсистема `Uikon`) загружает используемый им “главный” файл ресурса (подробнее об этом будет рассказано в главе 6, раздел “Регистрация программы

в меню приложений”). Если он имеет расширение “.rsc”, то он будет загружен. Если же файла с расширением “.rsc” не окажется, то система постарается найти файл с расширением “.rXX”, где XX — двузначный код используемого в данный момент в системе языка. Коды языков можно найти в перечислении TLanguage, объявленном в файле e32std.h. Зачастую дистрибутив приложения содержит сразу несколько локализованных “.rXX”-файлов. Если в системе выбран язык, локализация ресурса для которого отсутствует, то будет выбрана и подключена другая подходящая локализация.

Таким образом, локализация приложения для того или иного языка сводится к компиляции переведенного файла ресурса и размещению его в дистрибутиве с правильным расширением. SDK поддерживает ряд методов для автоматизации этого процесса. Как вы помните, для каждого файла ресурсов в MMP-файле проекта содержится его объявление, в котором вы можете с помощью ключевого выражения LANG указать коды поддерживаемых им локализаций. Например:

```
SOURCEPATH ..\data
START RESOURCE users.rss
    HEADER
    LANG 01 03 16
END
```

Объявляет ресурс users.rss с поддержкой английской (01), немецкой (03) и русской (16) локализации. Это значит, что наш файл ресурса users.rss будет скомпилирован трижды, и каждый раз результаты компиляции будут сохраняться в файлы users.r01, users.r03 и users.r16 соответственно. Перед каждой компиляцией выполняется директива #define LANGUAGE_XX, где XX — код локализации. Это позволяет в файле ресурсов определить, для какой локализации в данный момент происходит компиляция.

Для локализации файла ресурса необходимо сделать следующее:

- выделить все данные, нуждающиеся в переводе, в отдельный файл;
- создать несколько версий этого файла для различных языков;
- подключать нужную версию файла с локализованными строками при компиляции.

Заменяем все используемые в нашем файле users.rss строки идентификаторами и подключаем к нему с помощью директивы #include файл users.loc. Ту же операцию выполните с заголовочным файлом users.rh. В итоге они примут вид, представленный в листингах 2.2 и 2.3.

Листинг 2.2. Файл users.rh

```
#include "users.loc"

#define max_username 20

STRUCT USER_INFO
```

```

{
    BUF<max_username> name = STR_name_def;
    BYTE age;
    STRUCT friends[]; // USER_INFO
    DOUBLE loc[2] = {53.2091, 50.2854};
}

```

Листинг 2.3. Файл users.rss

```

#include "users.rh"
#include "users.loc"

NAME USER

RESOURCE USER_INFO myresource1
{
}

RESOURCE USER_INFO myresource2
{
    name = STR_name1;
    age = 25;
    friends =
    {
        USER_INFO
        {
            name = STR_name2;
            age = 18;
            friends =
            {myresource2};
            loc =
            {56.3127, 44.0174};
        },
        USER_INFO
        {
            name = STR_name3;
            age = 27;
            friends =
            {myresource2};
            loc =
            {55.0293, 82.9677};
        }
    };
}

```

Теперь мы можем расположить в каталоге \data\ проекта заголовочный файл users.loc следующего содержания.

```

#ifdef LANGUAGE_01
#include "users.l01"
#endif

#ifdef LANGUAGE_03
#include "users.l03"
#endif

#ifdef LANGUAGE_16
#include "users.l16"
#endif

```

Как видите, заголовочный файл `users.loc` в зависимости от идентификатора текущей локализации подключает один из трех файлов `users.lXX`. В таких файлах хранятся локализованные строки. Все файлы, хранящие нелатинские символы, должны быть в кодировке UTF-8 (без byte order mark) и содержать выражение `CHARACTER_SET UTF8`. Например, как выглядит файл `users.l16`, показано в листинге 2.4.

Листинг 2.4. Файл `users.l16`

```

CHARACTER_SET UTF8

#define STR_name_def "Неизвестно"
#define STR_name1 "Александр"
#define STR_name2 "Виктор"
#define STR_name3 "Олег"

```

Есть альтернативный способ задания локализованных строк — с помощью файлов RLS. В этом случае содержимое файла LOC примет следующий вид.

```

#ifdef LANGUAGE_01
#include "users_01.rls"
#endif

#ifdef LANGUAGE_03
#include "users_03.rls"
#endif

#ifdef LANGUAGE_16
#include "users_16.rls"
#endif

```

Файлы RLS не должны содержать директив `#include` или `#define`. Объявление строк в RLS-файлах имеет следующий формат.

```

rls_string <symbolic-identifier> <string>

```

Например:

```
rls_string STR_name_def "Неизвестно"  
rls_string STR_name1 "Александр"  
rls_string STR_name2 "Виктор"  
rls_string STR_name3 "Олег"
```

Традиционно RLS-файлы использовались в проектах для UIQ, а “.1XX” — в проектах для S60. Но принципиальная разница между ними лишь в том, что RLS-файлы нельзя по ошибке подключить к файлам исходного кода. Официально рекомендуется использовать RLS-файлы, но несмотря на это, на практике файлы “.1XX” встречаются чаще.

Подготовка к сертификации ASD

➤ Понимание роли ресурсов и текстовых файлов локализации в Symbian OS.

Прочие файлы проекта

Мы не будем подробно останавливаться на файлах исходного кода и заголовочных файлах проекта, так как предполагается, что вы уже знакомы с C++, а никаких существенных отличий в синтаксисе Symbian C++ не имеет.

В состав проекта также входят файл(ы) с расширением “.pkg”. Они хранятся в отдельном каталоге \sis\ и являются настройками для сборки SIS-пакета после компиляции приложения. Формат этих файлов, как и инструменты для работы с ним, будут рассмотрены в *главе 3*.

В корневом каталоге некоторых проектов вы можете встретить файл `application.uidesign` и другие файлы с расширением `.uidesign`. Это служебные файлы инструмента UI Designer, входящего в состав среды разработки Carbide.c++.

ГЛАВА 3

Работа с SDK

Прежде чем перейти к изучению программирования для Symbian OS, мы рассмотрим инструменты, которые понадобятся нам для компиляции приложений. Прежде всего, для этой цели потребуется **SDK** (Software Development Kit) — инструментарий для разработки программного обеспечения. В состав SDK входят документация, примеры, необходимые для компиляции утилиты, библиотеки и заголовочные файлы для доступа к API устройства, а также эмулятор.

Наличие SDK необходимо и достаточно для создания приложений, но вряд ли вам захочется писать исходный код в текстовом редакторе и компилировать проект из командной строки. Для того чтобы упростить и ускорить процесс разработки, используются IDE. С точки зрения программиста, лучшей IDE для языка Symbian C++ на сегодняшний день является Carbide.c++, работа с которой будет рассмотрена в *главе 3*. Использование IDE скрывает сложность компиляции и сборки приложения, тем не менее мы подробно рассмотрим данную процедуру во второй части этой главы — это может пригодиться вам при решении потенциальных проблем.

Выбор SDK

Прежде всего, необходимо отметить, что до появления Symbian^1 SDK создавались не для операционной системы, а для использующей ее *платформы*. Поэтому единой SDK для Symbian 9.x не существует. Вам потребуется версия SDK либо для S60, либо для UIQ. Но поскольку на сегодняшний день выпускаются устройства только с платформой S60, то и рассматривать мы будем именно ее.

SDK для S60 или UIQ следует искать на страницах сообществ разработчиков-производителей устройств, использующих эти платформы. Для S60 таким сообществом является Forum Nokia. Существует множество различных SDK для платформы S60: каждая редакция и Feature Pack имеет свою SDK, более того, с течением времени они могут обновляться (например, alpha-версия сменяется версией 1.0 и т.д.). Вам необходимо выбрать и скачать нужную версию. Выбор SDK осуществляется исходя из того, для каких устройств вы собираетесь создавать приложения. На сайте Forum Nokia есть раздел **Device Specifications**, где вы можете узнать версию платформы S60 на любой модели смартфонов Nokia. Например, для Nokia 3250 (S60 3-й редакции) вам потребуется S60 3rd Edition SDK for Symbian OS Maintenance Release, а для Nokia E90 (S60 3-й редакции,

Feature Pack 1) — S60 3rd Edition SDK for Symbian OS Feature Pack 1. Информацию о версии платформы можно получить и для моделей других производителей, использующих S60, — для этого нужно посетить их сайты. Определившись с версией платформы, вы можете скачать необходимый вариант SDK.

Скорее всего, вы уже задались вопросом: неужели мне придется компилировать отдельную версию программы для каждой модели устройства или версии платформы? Это не так. Все пакеты Feature Pack в рамках одной редакции обратно совместимы. Более того, 5-я редакция платформы S60 обратно совместима с 3-й. Это значит, что, создав приложение с помощью SDK для S60 3rd edition, вы вправе рассчитывать на то, что оно запустится на всех версиях этой редакции платформы и даже на 5-й. Почему бы тогда не использовать эту версию SDK для всех проектов? Многие так и поступают, но новые версии платформы предлагают разработчику новые возможности, а SDK для них — новые API, улучшенную документацию, более быстро работающий эмулятор и т.д. Некоторые опытные программисты для удобства используют SDK от FP1 или FP2 и создают с их помощью приложения, прекрасно работающие на всех версиях 3-й редакции платформы. Но это требует некоторого мастерства и знания нюансов изменений, произошедших в различных версиях платформы. Например, в S60 3rd FP1 немного изменился формат главной пиктограммы приложения¹, и может статься, что у приложения, скомпилированного с помощью SDK для 3rd FP1 или FP2, на устройстве с первой версией S60 3rd его пиктограмма отображаться не будет. Поэтому на период обучения, пока вы не набрались достаточно опыта, я рекомендую использовать именно ту SDK, которая предназначена для устройства, на котором вы будете тестировать свои приложения.

Установка SDK

Прежде чем устанавливать SDK, прочитайте содержащийся в дистрибутиве документ Installation Guide. Обратите внимание на минимальные требования к ПК, в частности на частоту процессора (500 МГц-1,0 ГГц), объем RAM (128–512 Мбайт) и свободного места на диске (от 1 Гбайт). Затем вам потребуется скачать и установить ActivePerl 5.6.1 (SDK использует скрипты на Perl) и Java Runtime Environment (требуется утилитам Sisar, AIF Builder, CS Help Compiler и для настроек эмулятора). Обратите внимание, что версия JRE различается в зависимости от используемой вами SDK — скачайте нужную. В то же время ActivePerl должен быть именно версии 5.6.1.x — более старые или новые не подойдут. Устанавливать JRE и ActivePerl можно в любом порядке, даже после SDK. Но приступать к работе с SDK можно лишь тогда, когда все три пакета будут установлены.

Дистрибутив SDK снабжен инсталлятором, содержащим простой мастер установки. С его помощью вы легко установите SDK, но здесь есть некоторые нюансы.

¹ KIS000531 — Compatibility problem with binary-encoded SVG images.

- Во-первых, запускать инсталлятор следует под учетной записью администратора. Если вы работаете в Windows Vista, щелкните на исполняемом файле инсталлятора правой кнопкой мыши и выберите в раскрывшемся контекстном меню команду **Запустить от имени администратора**.
- Во-вторых, устанавливать SDK можно в любой каталог, но ваши проекты должны будут располагаться в том же разделе диска, а путь установки не должен содержать пробелов. Я настоятельно советую выбирать диск C: и не менять каталог установки — примеры в книге предполагают именно такой вариант установки.
- В-третьих, большинство SDK было выпущено до появления Windows Vista и не полностью совместимо с этой ОС. Обязательно найдите и прочитайте раздел *Moving to Windows Vista* в Википедии сообщества Forum Nokia: там собраны сведения о всевозможных проблемах, возникающих в работе некоторых SDK под Windows Vista, и предлагаются способы их решения.

Состав SDK

После установки SDK в каталог `C:\Symbian\` вы обнаружите подкаталоги `\9.x\S60_3rd_yy\`, где *x* — часть номера версии Symbian OS, *yy* — версия SDK. Такое размещение данных позволяет устанавливать сразу несколько различных SDK на один компьютер. В каталоге `C:\Symbian\9.x\S60_3rd_yy\` находятся следующие подкаталоги.

- `\Eros32\` — здесь хранятся библиотеки, заголовочные файлы, инструменты и эмулятор платформы:
 - `BUILD\` — каталог, в который помещаются сгенерированные скрипты и всевозможные файлы, необходимые для сборки проекта;
 - `Data\` — содержит файлы настроек и ресурсов для эмулятора;
 - `GCC\` — некоторые инструменты, используемые при компиляции (`bison`, `flex++` и пр.);
 - `include\` — в этом каталоге хранятся заголовочные файлы библиотек и ресурсов, содержащие объявления классов и констант всевозможных API Symbian OS;
 - `kit\` — содержит компоненты (документацию), которые могут подключаться различными средами разработки (`Carbide.c++`), использующими эту SDK;
 - `release\` — каталог, содержащий библиотеки (`".lib"` и `".dso"`) для компоновки исполняемых файлов;
 - `tools\` — содержит командные файлы, скрипты и утилиты (в основном консольные) SDK;
 - `winscw\` — в этот каталог помещены файлы и подкаталоги, имитирующие файловую систему устройства для эмулятора.

- `\Examples\` — исходный код примеров приложений (в основном консольных), ограничивающихся использованием сервисов Symbian OS (графика, потоки, базы данных, ECom и пр.).
- `\S60Doc\` — набор документации. Чаще всего документы предоставляются в форматах PDF и CHM. В этом случае работу со справочником можно начать, открыв файл `_index.chm`. Ссылку на документацию SDK всегда можно найти в меню Пуск Windows.
- `\S60Ex\` — исходные коды примеров приложений, использующих API уровня платформы.
- `\S60Tools\` — дополнительные утилиты:
 - `Ecmt\` — содержит `EcmtAgent`, инструмент для установки USB-, WLAN- или Bluetooth-соединений в целях отладки;
 - `MBMViewer\` — программа для просмотра содержимого `.mbm`-файлов (пакет изображений);
 - `Simpsyconfigurator\` — содержит модуль позиционирования PSY; будучи установленным на устройство, он позволяет имитировать его местонахождение в различных географических координатах;
 - `svg2svgt\` — инструмент для конвертирования векторной графики в формате SVG в формат SVG-T, применяемый для мобильных устройств.

Следует отметить, что в SDK входят не все API уровня платформы и операционной системы — часть заголовочных файлов и библиотек отсутствуют. Использование ряда API невозможно или ограничено в целях безопасности. С другой стороны, API, включенные в SDK, имеют некоторые гарантии совместимости со следующими версиями ОС и платформы. Чем больше их опубликовано, тем больше связывают себе руки разработчики Symbian OS и платформ. Поэтому в состав SDK входят лишь те API, за неизменность и совместимость которых можно не опасаться. Не вошедшие в состав SDK API обычно называют **приватными** (private API).

Несмотря на это, все же существует ряд способов получить некоторые дополнительные API.

- Во-первых, на Forum Nokia публикуются дополнения к SDK (Extensions Plug-in for S60 SDK). В них публикуется часть API, не вошедших в стандартную SDK по причине отсутствия гарантий совместимости. Такие дополнения можно бесплатно скачать со страниц портала Forum Nokia.
- Во-вторых, Symbian Developer Network имеет партнерскую программу, члены которой могут получить **SODK** (Symbian OS Development Kit), **BAK** (Binary Access Kit) или **DevKit** (Development Kit). Это инструментарии (аналогичные SDK) разной степени комплектации, предоставляющие доступ ко многим API Symbian OS (компоненты, формирующие платформу, в них отсутствуют).

- В-третьих, партнерские программы Forum Nokia позволяют запросить то или иное приватное API уровня платформы.

Forum Nokia и SDN поддерживают несколько разноуровневых партнерских программ — одни из них бесплатны, в другие можно вступить только по приглашению, третьи — предполагают выплату членского взноса при вступлении.

На момент написания книги организация Symbian Foundation еще не функционирует в полной мере, но уже известно, что Symbian Foundation предполагает участие сторонних разработчиков в создании открытой платформы. Однако пока не ясно, насколько открытым и полным будет исходный код новой платформы и на каких условиях к нему можно будет получить доступ.

Выбор текущего SDK

Работа с входящими в состав SDK утилитами ведется из командной строки Windows. Для того чтобы открыть редактор командной строки, достаточно в окне утилиты Выполнить, доступной через меню Пуск Windows, ввести команду **cmd**.

На компьютере может быть установлено сразу несколько SDK для различных версий платформы. Содержащиеся в них утилиты имеют одинаковые имена исполняемых файлов, поэтому, прежде чем приступить к работе, необходимо активировать ту версию SDK, которая будет использоваться для сборки проекта. Для этого необходимо изменить переменные среды EROSCROOT и PATH, что легче всего сделать с помощью утилиты `devices.exe`.

Введите в редакторе командной строки команду `devices`, и вы получите список всех SDK, установленных на компьютере. Пример.

```
C:\>devices
```

```
S60_5th_Edition_SDK_v0.9:com.nokia.s60
UIQ3.1:com.symbian.UIQ
S60_3rd_FP1:com.nokia.s60 - default
S60_3rd_FP2_Beta:com.nokia.s60
S60_3rd_MR:com.nokia.s60
```

Метка “- default” указывает на используемый в данный момент SDK. Другой способ определить это имя — запустить команду `devices` с ключом `-default`.

```
C:\>devices -default
```

```
Default device: S60_3rd_FP1:com.nokia.s60
```

Для того чтобы изменить SDK, используемый по умолчанию, следует воспользоваться командой `devices` со следующими ключами: `-setdefault @device-identifier`, где параметр `device-identifier` — название требуемого SDK в формате *идентификатор.имя*. Пример.

```
C:\>devices -setdefault @S60_3rd_MR:com.nokia.s60
```

По сути, утилита `devices.exe` является редактором XML-файла `C:\Program Files\Common Files\SYMBIAN\devices.xml`, каждая запись которого имеет следующий вид.

```
<device id="sdk_id" name="name" alias="alias" default="yes\no"
userdeletable="yes\no">
<epocroot>epoc_path</epocroot>
<toolsroot>tools_path </toolsroot>
</device>
```

Пример.

```
<device id="S60_3rd_FP2_Beta" name="com.nokia.s60" default="no"
userdeletable="yes">
<epocroot>C:\Symbian\9.3\S60_3rd_FP2_Beta</epocroot>
<toolsroot>C:\Symbian\9.3\S60_3rd_FP2_Beta</toolsroot>
</device>
```

Каждый SDK при установке регистрируется в этом файле. Утилита `devices` позволяет добавлять и удалять хранящиеся в нем записи с помощью ключей `-add` и `-remove`, выводить данные о них с помощью ключа `-info`, а также назначать записям более удобные символьные названия с помощью ключа `-setalias`. Информацию о необходимых при этом параметрах можно получить, выполнив команду `devices -?`.

Если на вашем компьютере установлен только один SDK, то в файле `devices.xml` будет единственная запись, и никаких дополнительных действий для ее активации вам проводить не потребуется.

Компиляторы, платформы и режимы компиляции

Приложения и библиотеки, предназначенные для использования в эмуляторе SDK, должны быть скомпилированы для набора инструкций процессора архитектуры x86. В то же время на устройствах под управлением Symbian OS применяются процессоры архитектуры ARM с отличным набором инструкций. Это главная, но не единственная причина того, что SDK содержит в себе механизмы компиляции приложений под различные платформы.

Как вы уже знаете, перечень поддерживаемых проектом платформ задается в файле `bld.inf` после выражения `PRJ_PLATFORMS`. Существует несколько видов таких платформ. Наиболее часто используемые из них приведены в табл. 3.1.

В состав SDK входит два компилятора: **Nokia x86** и **GCCE**. Компилятор Nokia x86 был разработан компанией Metrowerks и приобретен Nokia в 2004 году. Он используется исключительно для компиляции кода для Windows-эмуляторов устройств под управлением Symbian OS (в более ранних версиях SDK в аналогичных случаях использовался компилятор Microsoft C++ и платформа WINS).

Таблица 3.1. Платформы компиляции

Платформа	Описание
WINSW	Компиляция файлов для исполнения в эмуляторе Windows. Используется компилятор Nokia x86
GCCE	Компиляция для набора инструкций ARMv5 (EABI v2). Используется компилятор GCCE
ARMv5	Компиляция для набора инструкций ARMv5 (EABI v1). Используется компилятор ARM RVCT 2.2
ARMv5_EABIv2	Компиляция для набора инструкций ARMv5 (EABI v2). Используется компилятор ARM RVCT 2.2
ARMv6	Компиляция для набора инструкций ARMv6. Используется компилятор ARM RVCT 2.2

Компилятор GCCE (GNU Compiler Collection for Embedded) в SDK для Symbian 9.x заменил компилятор GCC. SDK содержит GCCE версии 3.4.3, но некоторые разработчики на свой страх и риск подключают GCCE 4.x.

Компилятор RVCT 2.2 (ARM® RealView® Compilation Tools) не является частью SDK, а входит в состав ARM® RealView® Development Suite (RVDS) 2.2 — коммерческого пакета для разработки приложений под любые ARM-процессоры. По сравнению с GCCE, RVCT осуществляет лучшую оптимизацию кода. Частой ошибкой у неопытных разработчиков является попытка собрать проект для целевой платформы ARMv5 при отсутствии установленного RVCT 2.2.

EABI (Embedded Application Binary Interface, бинарный интерфейс встраиваемых приложений) — низкоуровневый интерфейс, регламентирующий взаимодействие бинарного кода, библиотек и ОС (например, передачу аргументов функций). Соответствие одному бинарному интерфейсу (EABIv2) позволяет свободно использовать библиотеки и приложения, скомпилированные с помощью RVCT, в программах, скомпилированных с помощью GCCE. Интерфейс EABIv1 совместим с EABIv2 во время выполнения (но не во время компоновки). Исключение составляет код EABIv1, скомпилированный RVCT 2.1.

На практике компилятор RVCT применяется намного реже, чем GCCE. Его использование оправданно при создании драйверов и компонентов системы, предъявляющих высокие требования к производительности и размеру кода. Так как в состав SDK компилятор RVCT 2.2 не входит, мы ограничимся вопросами работы с компилятором GCCE.

Архитектура ARMv5 поддерживает два набора инструкций: 32-битовый **ARM** и 16-битовый **THUMB**. THUMB предоставляет 16-битовые аналоги инструкций ARM, которые во время выполнения “на лету” транслируются в 32-битовые. Поэтому THUMB-инструкции выполняются медленнее, но занимают меньше памяти и имеют меньшее общее энергопотребление.

Для всех платформ компиляции поддерживаются два режима: **UDEB** (Unicode Debug Build) и **UREL** (Unicode Release Build). При компиляции в режиме UDEB в исполняемый файл добавляется информация для отладчика. Помимо этого, в зависимости от режима компиляции некоторые функции могут

вести себя по-разному в случае возникновения ошибок. Например, возвращать код ошибки при UREL-сборке и вызывать панику при UDEB-сборке. Благодаря этому UDEB-режим удобен в процессе разработки, так как обращает внимание программиста на происходящие в программе ошибки.

Подготовка к сертификации ASD

- Понимание того, что ARM C++ EABI является стандартом индустрии, оптимизированным для разработки встраиваемых приложений.
 - Знание основных сведений о компиляторах RVCT и GCC.
 - Понимание того, что ARMv5 поддерживает и 32-битовый (ARM), и 16-битовый (THUMB) набор инструкций, а также их влияние на скорость выполнения и размер кода.
-

Сборка проекта

Для того чтобы скомпилировать исполняемые файлы и ресурсы проекта, необходимо выполнить всего две операции.

- Во-первых, вы должны перейти в каталог, содержащий файл `bld.inf`, и выполнить следующую команду.

```
bldmake bldfiles
```

В результате в этом же каталоге будет создан пакетный файл `abld.bat`. С его помощью и производится сборка проекта. Команду `bldmake bldfiles` необходимо вызывать лишь в том случае, если файл `abld.bat` отсутствует или файл `bld.inf` был изменен. В остальных ситуациях можно сразу переходить ко второму этапу.

- Во-вторых, вызовите полученный на первом шаге пакетный файл `abld.bat` с командой `build`.

```
abld build
```

Этого будет достаточно для того, чтобы собрать проект. При этом будет выполнена компиляция ресурсов, изображений, библиотек и исполняемых файлов, входящих в проект.

Не правда ли, просто? Теперь мы подробнее рассмотрим, что же на самом деле происходит во время сборки проекта.

Начнем с того, что при вызове команды `bldmake bldfiles` создается не только пакетный файл `abld.bat`, но и по два вспомогательных MAKE-файла для каждой объявленной в `bld.inf` платформы компиляции. Файлы с именами `<platform>.MAKE` и `<platform>TEST.MAKE` называют **сборщиками первого уровня** (first level makefiles), они появляются в каталоге `%EPOCROOT%\epoc32\build\<absolute_path_to_Bld.Inf>\`. В MAKE-файлах содержатся платформо-зависимые команды.

При вызове команды `abld build` происходит запуск скрипта `abld.pl` с тем же параметром и путем к каталогу, хранящему ММР-файлы проекта. (Вот для

чего работа SDK требует установки Active Perl.) Скрипт `abld.pl` содержит инструкции для выполнения различных команд (почти два десятка), переданных в качестве параметра. Команда `build` сама по себе никаких операций не выполняет, а служит лишь для последовательного запуска следующего набора команд: `export`, `makefile`, `library`, `resource`, `target` и `final`. Эти команды можно вызывать и по одной, передавая их `abld.bat`.

- `abld export` — копирует файлы из разделов `prj_exports` и `prj_testexports` `bld.inf` в назначенные им каталоги.
- `abld makefile` — с помощью MAKE-сборщиков первого уровня создает для каждого MMP-файла платформозависимые сборщики второго уровня. Они сохраняются в каталоге `%EPOCROOT%\epoc32\build\<absolute_path_to_Bld.Inf>\<MMP file name>\<platform>` под именами `<MMP file name>.<platform>`. При работе используется утилита SDK `makmake`.
- `abld library` — в случае, если в вашем проекте есть библиотеки, экспортирующие некоторые функции, то на основании их DEF-файлов создаются библиотеки импорта. Подробнее о DEF-файлах мы поговорим в следующем разделе.
- `abld resource` — сборка ресурсов проекта. Сначала компилируются файлы ресурсов RSS. Для этого используется скрипт `epocrs.pl`. Он, в свою очередь, вызывает инструменты препроцессинга C++ (выполняются макросы и директивы, содержащиеся в файле ресурсов), а затем компилятор RComp (resource compiler). RComp создает бинарный файл ресурсов RSC и (при необходимости) заголовочный файл ресурса RSG. После подготовки ресурсов выполняется сборка MBM-файлов изображений с помощью утилиты `bmconv`. Она же при необходимости создает для MBM-файла файл с расширением `“.mbg”`. В нем объявляется перечисление, содержащее значения индексов, находящихся в MBM-файле изображений. Файл MBG подключается к исходному коду приложения, так же как RSG.
- `abld target` — сборка исполняемых файлов проекта. Сначала вызывается нужный компилятор (в зависимости от целевой платформы) для файлов исходного кода проекта. Компилятор производит объектные файлы с расширением `“.o”`. Затем для них вызывается компоновщик. Компиляторы GCCE и RVCT создают исполняемые файлы в формате `elf` (executable and linkable format), применяющиеся в `pix`-системах. Symbian OS имеет собственный формат для исполняемых файлов — `E32Image`. Поэтому после компиляции и компоновки `elf`-файлов они транслируются в формат `E32Image` с помощью утилит `elf2e32` (для ABIv2) или `elftran` (для ABIv1). Полученные файлы помещаются в каталог `%EPOCROOT%\epoc32\release\<TargetBuild>\<BuildType>`.
- `abld final` — вызов дополнительных сборочных скриптов. Такие скрипты чаще всего имеют расширение `“.mk”` и находятся в каталоге `\group\` проекта.

Вышеизложенный процесс сборки продемонстрирован на схеме, представленной на рис. 3.1. Полужирным шрифтом на нем отмечены компоненты, которые понадобятся нам для создания SIS-дистрибутива.

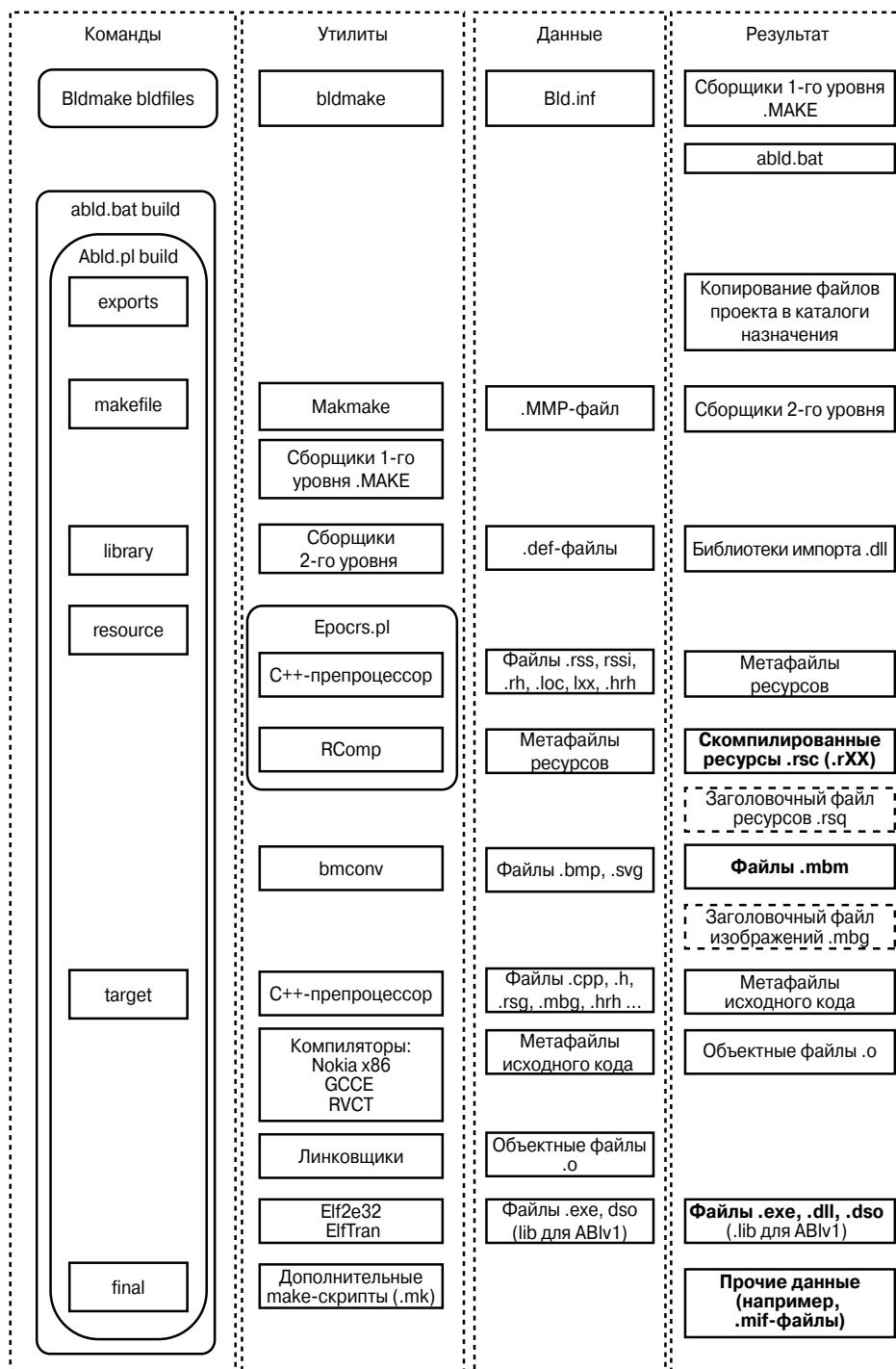


Рис. 3.1. Сборка проекта

Подготовка к сертификации ASD

- Знание основ использования команды `bldmake`, файлов `bld.inf` и `abld.bat`.

Заморозка проекта, def-файлы

В библиотеках Symbian OS отсутствует информация об именах экспортируемых ими функций. Обращаться к ним можно только по их номеру (ординалу). Поэтому для того чтобы не нарушить бинарную совместимость вашей библиотеки с использующими ее программами, важно, чтобы порядок экспортируемых ею функций не менялся. Для того чтобы контролировать этот порядок, используются файлы с расширением `.def` (module definition file). Вот пример простейшего DEF-файла.

EXPORTS

```
???1Clib@@UAE@XZ @ 1 NONAME ; Clib::~~Clib(void)
?NewL@Clib@@SAPAV1@XZ @ 5 NONAME ; class Clib * Clib::NewL(void)
?NewLC@Clib@@SAPAV1@XZ @ 6 NONAME ; class Clib * Clib::NewLC(void)
```

В нем объявлены три экспортируемые библиотекой функции и их параметры (деструктор и двухфазный конструктор).

Создавать DEF-файлы вручную не нужно — для этого служит утилита командной строки `abld freeze`. В ходе ее выполнения для библиотеки генерируется новый или обновляется существующий DEF-файл. Сам процесс создания таких файлов для входящих в состав проекта библиотек называют **заморозкой** (freeze project). Для хранения DEF-файлов в проекте создаются каталоги с именами использованных бинарных интерфейсов (BWINS для платформы WINSCW и EABI для GCCE\ARMV5).

Работа с эмулятором

Все SDK для различных платформ Symbian OS содержат эмулятор, позволяющий протестировать созданные вами приложения.

Чтобы запустить эмулятор текущей SDK, достаточно ввести в командной строке `eros` или `sdk` либо воспользоваться соответствующей командой в меню Пуск. Запуск эмулятора может занять достаточно продолжительное время, после чего откроется окно, показанное на рис. 3.2.

В заголовке этого окна указана версия SDK, в состав которого входит запущенный эмулятор. Содержимое панели меню окна мы рассмотрим чуть позже. Сразу под ней выводится текущее разрешение и ориентация экрана эмулируемого устройства. Щелкнув мышью на кнопке слева от этой надписи, можно изменить настройку экрана. Эмулятор поддерживает ограниченное количество вариантов настройки, и, последовательно щелкая на этой кнопке, вы сможете перебрать их все.



Рис. 3.2. Окно эмулятора SDK 3-й редакции

Клавиатура под экраном эмулируемого устройства эквивалентна клавишам набора смартфона (keypad). К сожалению, QWERTY-клавиатуру (keyboard) эмуляторы S60 3-й редакции не предоставляют (для ряда моделей существуют решающие эту проблему дополнения к SDK, например, Nokia Enterprise Solutions SDK Plug-in). Такая возможность появилась в эмуляторе S60 5-й редакции. Нажимать на кнопки эмулятора следует, щелкая на них мышью. Если вместо щелчка нажать и удерживать, не отпуская, левую кнопку мыши, можно симитировать удерживание соответствующей клавиши эмулируемого устройства (например, показ диспетчера приложений при удерживании кнопки **Applications**). Некоторым кнопкам и функциям эмулятора соответствуют комбинации клавиш клавиатуры ПК, представленные в табл. 3.2. По умолчанию во время работы эмулятор использует системную звуковую карту и текущее подключение к Интернету.

Таблица 3.2. Горячие клавиши эмулятора

Клавиша	Функция
<Esc>	Закрыть (отмена) текущий диалог или окно
<F1>	Отобразить меню Options
<F9>	Выключить экран эмулятора. Повторное нажатие включит его
<F10>	Имитация экстренного отключения (разрядка батареи). Нажатие <F9> включит эмулятор
<~>	Соответствует кнопке выделения. (Кнопка ОК в навигационных клавишах)
<Alt+1>, <Alt+2>	Соответственно левая и правая командные клавиши (soft key)
Кнопки курсора	Соответствуют навигационным клавишам
<0>->9>, <*> и <#>	Соответствуют цифровой клавиатуре стандарта ITU-T
<Home>	Соответствует клавише Applications
<Ctrl+H>, <Backspace>	Соответствуют клавише Clear
<Shift>	Соответствует клавише Edit
<Ctrl+Alt+Shift+O>	Переворачивает оболочку эмулятора на 180°
<Ctrl+Alt+Shift+J>	Отображает цветные рамки вокруг элементов управления. Полезно при определении размеров при разных настройках экрана эмулятора
<Ctrl+Alt+Shift+P>	Отображает настройки утилиты для генерации ошибок обращения к ресурсам (ошибки при выделении памяти в куче приложением или оконным сервером, при обращении к файлам). Ошибки могут генерироваться случайным образом или с заданной периодичностью
<Ctrl+Alt+Shift+Q>	Отключение режима генерации ошибок при выделении памяти в куче
<Ctrl+Alt+Shift+R>	Полная перерисовка экрана эмулятора
<Ctrl+Alt+Shift+F>	Отключает использование буфера сообщений в оконном сервере (auto-flush mode). Это, в целом, замедляет перерисовку элементов управления, что позволяет заметить мерцание
<Ctrl+Alt+Shift+G>	Включает использование буфера сообщений в оконном сервере (по умолчанию включено)
<Ctrl+Alt+Shift+M> или <Ctrl+Alt+Shift+Enter>	Отображает диалоговое окно Move Me. Его можно перемещать по экрану навигационными клавишами. Используется для тестирования частичной перерисовки элементов управления
<Ctrl+Alt+Shift+N>	Включение или выключение режима Scheduler Shaker (по умолчанию выключен)
<Ctrl+Alt+Shift+V>	Включение или выключение отображения сообщений, посланных с помощью метода <code>CEikonEnv::VerboseInfoMsg</code>
<Ctrl+Alt+Shift+Z>	Эквивалентно быстрому вводу символов от A до J. Позволяет протестировать скорость обработки нажатий в элементах управления

Меню эмулятора содержит несколько полезных команд.

- **File⇒Open** — позволяет симитировать открытие файла в системе. При этом запускается ассоциированная с данным типом файла программа (Media Gallery для изображений, Audio Player для музыки и т.д.). Данная функция удобна для тестирования механизмов регистрации определенных типов файлов и автозапуска программ при их открытии.
- **File⇒Open URL** — позволяет открыть в системе веб-ресурс по заданному URL. Эмулятор автоматически использует текущий канал подключения к Интернету вашего ПК.
- **Tools⇒Switch Configuration** — по действию аналогична щелчку на кнопке переключения настроек экрана эмулятора.
- **Tools⇒Diagnostics** — запускает утилиту профилирования приложений. С ее помощью можно определить, сколько оперативной памяти и процессорного времени расходует тот или иной процесс. В окне **Diagnostics** также можно просмотреть журнал HTTP-запросов приложений.
- **Tools⇒Utilities** — инструмент для эмуляции различных системных событий: подключения\отключения виртуальной карты памяти, зарядного устройства, SIM-карты, различных аксессуаров, получения SMS/MMS и даже срабатывания будильника. Помимо этого, в окне **Utilities** предоставляется возможность задания текущего местоположения (географических координат) и траектории перемещения устройства.
- **Tools⇒Preferences** — открывает окно параметров настройки эмулятора. Файл `epoc.ini` настроек эмулятора расположен в каталоге `\epoc32\data\`, и в данное окно позволяет задать некоторые его параметры.
 - Вкладка **C++ Debug** — вывод отладочной информации.
 - **Log to debugger** — вывод в отладчик IDE. Функция включена по умолчанию и позволяет просматривать отладочную информацию в среде Carbide.c++.
 - **Just in time** — позволяет регулировать поведение эмулятора при возникновении паники в исполняемом в нем приложении. По умолчанию данный режим включен, а эмулятор завершает работу вместе с аварийно закрытым процессом. Если же его отключить, то работа будет продолжена и после возникновения паники, что позволит увидеть системное диалоговое окно с ее описанием.
 - **Enable EPOCWIND.OUT logging** — сохранение отладочной информации в файле `EPOCWIND.OUT`. Файл расположен в каталоге `C:\Documents and Settings\\Local Settings\Temp\`. Перед использованием убедитесь, что для него не установлен атрибут **Только для чтения**. В противном случае открыть этот файл во время работы эмулятора вы не сможете — он будет заблокирован.
 - **Extended panic code file** — имитируется присутствие в системе файла `errrd`. Назначение и использование файла `errrd` обсуждается в *главе 5*, раздел “Обработка ошибок и исключений: сбросы, ловушки и паника”.
 - Вкладка **General Settings** — основные параметры эмулятора.

- **Memory Capacity** — объем используемой RAM-памяти в мегабайтах.
- **Initial Resolution** — разрешение экрана по умолчанию. (Переключается с помощью команды меню **Tools⇒Switch Configuration**.)
- **PAN (Personal Area Network)** — эмуляция беспроводной сети.
- **Bluetooth** — настройки Bluetooth. Здесь можно включить или отключить поддержку Bluetooth, задать используемый протокол (H4/BCSP) и номер COM-порта для работы с ним.
- **IrDa** — инфракрасный порт эмулятора. Допускает включение или отключение и настройку номера COM-порта.
- **Network** — включение и настройка прокси-сервера для доступа к Интернету.
- **Platform Security** — настройка платформы безопасности эмулируемого устройства.
- **Enable security debug messages** — включить вывод отладочных сообщений платформы безопасности (в файл или отладчик IDE).
- **Panic|Leave on insecure API calls** — генерация исключительных ситуаций (сброса или паники) при вызове API, требующих доступа к защищенным возможностям, недеklarированным приложением.
- **Platform capability check** — позволяет отключить проверку доступа к выбранным защищенным возможностям.

В окне **Preferences** отображены далеко не все параметры, которые могут быть настроены в файле `epoc.ini`. Например, редактируя файл `epoc.ini` вручную, разработчик может добавить новую конфигурацию экрана или подключить еще один виртуальный диск. Полное описание формата файла `epoc.ini` можно найти в справочнике SDK (начиная с SDK 3-й редакции FP1).

В зависимости от версии SDK внешний вид и состав команд панели меню могут незначительно отличаться. Следует помнить, что запуск эмулятора на некоторых ПК может занимать довольно продолжительное время (около 10 секунд). Также довольно долго открываются окна инструментов. Начиная с SDK 3-й редакции FP2, в окне параметров настройки, открываемом командой меню **Tools⇒Preferences**, на вкладке **General Settings** имеется параметр настройки, позволяющий запускать минимальный набор серверов при старте эмулятора, что уменьшает время его запуска. По умолчанию эмулятор запускается с неполным набором служб (*partial*). Индикатор текущего режима выводится в правом верхнем углу (литеры **P** или **F**, как показано на рис. 3.3).

Исполняемые файлы и библиотеки самого эмулятора хранятся в каталоге `%EP0CROOT%\epoc32\release\<emulator-build>\<variant>`, где `<emulator-build>` и `<variant>` — целевая платформа и режим компиляции, использованные при их создании. Обычно в состав SDK входит только одна версия: `winscw\udeb`. Все файлы настроек хранятся в каталоге `\epoc32\data\`. Файловая система эмулятора имитируется в каталоге `%EP0CROOT%\epoc32\release\<emulator-build>`, например:

78 Symbian C++. Программирование для мобильных телефонов

C:\Symbian\9.1\S60_3rd_MR\Epoc32\winscw\c\
C:\Symbian\9.1\S60_3rd_MR\Epoc32\winscw\d\



Рис. 3.3. Окно эмулятора SDK 5-й редакции с расширенной индикацией текущего режима

Исключение составляет виртуальный диск Z :, находящийся в следующем каталоге.

`%ЕРОСROOT%\epoc32\release\<emulator-build>\<variant>\z\`

Как вы уже знаете, исполняемые файлы приложений Symbian 9.x должны быть расположены в каталоге `\sys\bin\`, но такого каталога на виртуальных дисках эмулятора вы не найдете. Дело в том, что эмулятор переадресует все обращения к `\sys\bin\` в запросы к содержимому каталога `%ЕРОСROOT%\epoc32\release\winscw\<BuildVariant>\`. Таким образом, эмулятор полу-

чает доступ ко всем собранным исполняемым файлам напрямую, без необходимости копировать их на виртуальный диск. Это значит, что ваша программа появится в эмуляторе, как только она будет собрана для платформы WINSCW, без необходимости создания SIS-пакета и его установки системным инсталлятором. Как уже объяснялось в *главе 1*, раздел “Платформа безопасности Symbian OS”, именно инсталлятор SIS-файлов является компонентом, отвечающим за проверку сертификатов и подтверждение правомерности доступа приложения к декларированным им защищенным возможностям. Так как эта проверка не выполняется, то приложение автоматически получает доступ ко всем объявленным в нем защищенным возможностям. По тем же причинам в эмуляторе автоматически *не создается* приватный каталог процесса.

К сожалению, это не единственные отличия эмулятора от настоящего устройства. Эмулятор Symbian-платформы в Windows представляет собой один-единственный процесс, а все запускаемые в нем программы и службы являются его потоками. Это необходимо учитывать, в том случае если тестируемая вами программа оперирует процессами системы напрямую. По той же причине в эмуляторе невозможно реализовать планировщик процессов, идентичный настоящему (отсутствует достаточное количество значений приоритетов потока). При выполнении программы в эмуляторе размер используемого ею стека увеличивается автоматически, в то время как на устройстве исчерпание стека приведет к аварийному завершению работы. Процессоры, применяемые на устройствах под управлением Symbian OS, допускают адресацию памяти с точностью не менее 32-х бит. Другими словами, обращение к адресу памяти, не кратному четырем, на устройстве приведет к ошибке. Результат выполнения функции `sizeof()` для структур также всегда будет кратным четырем.

Наконец, имеются существенные различия в производительности эмулятора и устройства. Например, процессор ПК имеет аппаратную поддержку вычислений с плавающей точкой, тогда как процессор смартфона ее может и не иметь. И это не весь список особенностей эмулятора, которые необходимо учитывать при тестировании и отладке программ.

Все вышесказанное вкупе с невнимательностью и ленью разработчика довольно часто приводит к ситуации, когда программа отлично работает в эмуляторе, но на устройстве даже не запускается. Поэтому я рекомендую во время разработки приложений периодически проверять их работоспособность на смартфоне — это позволит выявить потенциальные проблемы еще на ранних стадиях разработки и быстрее определить их причину.

Подготовка к сертификации ASD

- Понимание назначения эмулятора Symbian OS для Windows.
 - Осознание различий в выполнении кода на эмуляторе и мобильном устройстве.
-

Очистка проекта

Очистка (clean) — операция по удалению большинства файлов, сгенерированных автоматически во время сборки проекта. Выполнять ее следует каждый раз при внесении изменений в файлы настроек проекта (ММР-файлы, `bld.inf`).

Для выполнения очистки проекта существует несколько команд:

- `abld clean` — удаляет все файлы созданные при выполнении команды `abld target`;
- `abld cleanexport` — удаляет файлы, экспортированные из проекта с помощью команды `abld export`;
- `abld reallyclean` — удаляет все, что было сгенерировано с помощью команд `abld makefiles`, `abld export` и `abld target`.

Рекомендую не терзаться сомнениями, а всегда использовать именно последний вариант.

Если изменения были внесены в файл `bld.inf`, то после выполнения команды `abld reallyclean` необходимо будет вызвать команду `bldmake clean` (не забудьте сделать активным нужную версию SDK с помощью команды `devices`). В результате будут удалены MAKE-сборщики и сам файл `abld.bat` в каталоге `\group\`.



Очисткой проекта пренебрегать не следует. Я советую выполнять ее всякий раз, когда при сборке произошли необъяснимые ошибки или в приложении не оказалось внесенных вами изменений. Если это не поможет — не поленитесь удалить файлы вручную. Возможно, это и не решит проблему, но, по крайней мере, вы будете знать, где ее точно нет.

Создание дистрибутива приложения

Итак, вы уже научились выполнять сборку проекта, теперь мы рассмотрим процесс установки приложения на устройство. Как уже неоднократно говорилось ранее, единственный способ инсталлировать программу — это поместить ее в SIS-пакет. В SIS-файл упаковываются исполняемые файлы, библиотеки, необходимые ресурсы и данные. Помимо этого, в нем содержится служебная информация и цифровой сертификат, подтверждающий доступ к защищенным возможностям Symbian OS. Создание дистрибутива приложения можно условно разделить на следующие этапы.

1. Сборка проекта для целевой платформы GCCE или ARMV5.
2. Подготовка PKG-файла, содержащего описание пакета.
3. Создание SIS-пакета с помощью утилиты `makesis.exe`.
4. Подписывание SIS-пакета имеющимся цифровым сертификатом с помощью утилиты `signsis.exe`.

Файл PKG

Файлы с расширением “.pkg” содержат информацию, позволяющую утилитам SDK собрать скомпилированное приложение в SIS-пакет. В большинстве случаев они хранятся в каталоге \sis\ проекта и представляют собой текстовый ASCII-файл, содержащий следующую информацию:

- поддерживаемые локализации;
- заголовок пакета (название, версия и тип);
- производитель;
- платформа или модель устройства, для которой предназначен пакет;
- зависимости пакета;
- описание данных и условий их установки;
- встраиваемые пакеты.

Локализации SIS-пакета

Поддерживаемые пакетом локализации объявляются в PKG-файле в виде разделенного запятыми списка кодов языков, начинающегося с символа “&”:

```
&lang-code [(dialect-ID)], lang-code [(dialect-ID)], ...
```

где *lang-code* — двухсимвольный код языка (например, EN — английский, RU — русский, FR — французский и т.д.), а *dialect-ID* — необязательный код диалекта языка (например, ZU (1024) — один из диалектов зулусского). Пример:

```
&EN, RU
```

Полный перечень кодов языков можно найти в справочнике SDK. Не следует путать локализации приложения и локализации SIS-пакета. Локализации SIS-пакета позволяют менять отображаемое название проекта, устанавливаемые файлы и различные сообщения в соответствии с текущим языком устройства.

В случае если перечень поддерживаемых локализаций опущен, по умолчанию используется значение &EN.

Заголовок SIS-пакета

Заголовок пакета имеет следующий формат.

```
#{"Package name for language 1"[, ...]}, (package-uid),  
major, minor, build-number[, package-options, ...]
```

Здесь "Package name for language 1" — название пакета для первой локализации. Если вы объявили поддержку нескольких локализаций, то необходимо через запятую перечислить название для каждой из них в соответствующем порядке. Название пакета используется в различных диалогах во время установки, а также выводится в списке установленных пакетов в системном менеджере приложений.

Параметр *package-uid* — идентификатор пакета (также в документации упоминается, как Package ID, pUID или PID). Идентификатор пакета PID является самостоятельным параметром. Использование UID3 какого-либо из входящих в пакет приложений в качестве его PID считается дурным тоном. Значение PID должно быть из защищенного диапазона и запрашивается разработчиком на страницах портала Symbian Signed. Однако на практике эти рекомендации не выполняются и для простоты используют в качестве PID пакета UID3 одной из входящих в него программ. В ряде случаев это даже необходимо (например, при использовании Startup List Management API). На этапе разработки и тестирования вы можете использовать любое значение PID. Идентификатор пакета служит для однозначного определения установленного дистрибутива и используется для проверки зависимостей в условных операторах блока данных PKG-файла, а также в ряде системных API (например, SW Installer Launcher API).

Числовые параметры *major*, *minor* и *build-number* составляют версию пакета.

Параметр *package-options* — в конце заголовка через запятую перечисляются необязательные опции дистрибутива. Всего их четыре.

- SH или SHUTDOWNAPPS — требование принудительного завершения работы всех пользовательских приложений перед установкой пакета.
- NC или NOCOMPRESS — не использовать компрессию содержимого пакета.
- IU или IUNICODE — сигнализирует о том, что файл PKG в кодировке UTF-8.
- TYPE=XX — тип пакета, где XX принимает одно из следующих значений.
 - SA или SISAPP — пакет устанавливает обычное приложение. Этот тип используется по умолчанию. Если пакет с таким же PID и названием в системе уже установлен, то сначала выполняется его деинсталляция.
 - SP или SISPATCH — дополнение к уже установленному пакету. Дополнение должно иметь тот же PID, что и обновляемый пакет, но другое название. SP позволяет лишь добавлять файлы, но не изменять уже установленные файлы. Дополнение можно деинсталлировать отдельно от пакета.
 - PU или PARTIALUPGRADE — частичное обновление пакета. Обновление должно иметь тот же PID и название, что и обновляемый пакет. Оно может и добавлять, и заменять файлы пакета. Обновление не отображается в списке установленных пакетов системного менеджера приложений, и поэтому не может быть деинсталлировано отдельно.
 - PA или PIAPP — пакет для предустановленного (например, на карту памяти) приложения. Такой SIS-файл не содержит данных, а служит исключительно для регистрации приложения в системе. Подробнее о правилах подготовки предустановленных приложений можно прочесть в справочнике SDK.
 - PP или PIPATCH — предустановленное дополнение к уже установленному пакету. Дополнение должно иметь тот же PID, что и обновляемый пакет, но другое название.

Пример объявления заголовка пакета.

```
#{"Package name", "Название пакета"}, (0x28B8C3CD), 1, 0, 0, IU
```

Диалоговое окно, которое будет выведено в устройстве с русским интерфейсом при инсталляции данного пакета, показано на рис. 3.4.

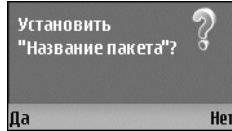


Рис. 3.4. Локализованное имя пакета во время установки

В случае, если в PKG-скрипте используется кириллица или другой нелатинский шрифт, этот файл должен быть сохранен в кодировке Unicode, а в заголовке пакета обязательно должна присутствовать опция IU.

Производитель

Имя производителя указывается в PKG-файле дважды. Первым задается уникальное имя, записанное по-английски, предназначенное для отображения в тех случаях, когда используемый в устройстве язык не совпадает ни с одной из локализаций. Уникальное имя указывается следующим образом.

```
: "Unique vendor name"
```

Вторым указывается локализованное имя, содержащее варианты для всех объявленных локализаций. Для приведенного нами ранее примера (английский и русский) оно будет выглядеть, например, следующим образом.

```
%{"Vendor", "Производитель"}
```

Обратите внимание, что строка, содержащая уникальное имя производителя, начинается с двоеточия, а локализованные имена — со знака “%”.



При установке дистрибутива, подписанного сертификатом типа self-generated, имя производителя может игнорироваться (вместо него всегда будет отображаться значение Unknwon). Это зависит от принятой политики безопасности устройства.

Поддерживаемая платформа или модели устройств

Каждой платформе и модели устройства назначен свой уникальный идентификатор (Product UID). Формат PKG-файла позволяет указать идентификатор платформы, для которой предназначен ваш SIS-пакет. Все версии платформ на базе Symbian 9.x совместимы снизу вверх. Если дистрибутив попытается инсталлировать на неподдерживаемую версию платформы, то во время установки будет отображено диалоговое окно с соответствующим предупреждением и предложением отказаться от установки (рис. 3.5).

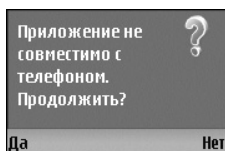


Рис. 3.5. Установка пакета на неподдерживаемую платформу или устройство

Платформа, для которой предназначен дистрибутив, указывается следующим образом.

```
[ProductUID], version-range, {"ProductName"}
```

Здесь параметры *ProductUID* — уникальный идентификатор платформы, а *ProductName* — ее символьное имя (используется в исключительно информационных целях). В табл. 3.3 представлены уникальные идентификаторы платформ на базе Symbian 9x.

Таблица 3.3. Значение Product UID для платформ на базе Symbian 9.x

Платформа	Product UID
S60 3-й редакции (Symbian 9.1)	0x101F7961
S60 3-й редакции FP1 (Symbian 9.2)	0x102032BE
S60 3-й редакции FP2 (Symbian 9.3)	0x102752AE
S60 5-й редакции (Symbian 9.4)	0x1028315F
UIQ3 (Symbian 9.1)	0x101F6300
UIQ3.1 (Symbian 9.2)	0x101F63DF

Вместо Product UID платформы в PKG-файле может быть использован идентификатор модели устройства. Например: 0x2000060B — Product UID для Nokia N95. Полный список идентификаторов устройств можно найти в Википедии сообщества Forum Nokia или на страницах портала NewLC.com.

Параметр *version-range* указывает версию прошивки платформы, для которой предназначен пакет. Версия задается кортежем *major*, *minor* и *build number*. Вы можете указать как точную версию, так и диапазон версий с помощью символа “~”. Примеры: 1, 2, 3 — точное указание версии; 1, 2, 3 ~ 7, 8, 9 — диапазон версий. Также допускается использование символа “*” или значения -1 в качестве произвольного числа. Например: 1, *, * — подойдет и для 1.0.0, и для 1.2.6. Если пакет подходит для любых версий прошивки (чаще всего), значение должно быть нулевым. Пример:

```
;Платформа S60 3.0 (Symbian 9.1)
[0x101F7961], 0, 0, 0, {"Series60ProductID"}
```

В случае, когда поддерживаемая пакетом платформа не указана, диалоговое окно с предупреждением о несовпадении будет выводиться при каждой установке.

Помимо платформы, разработчик может указать в PKG-файле и зависимость приложения от других пакетов. Объявление подобных зависимостей имеет следующий формат.

```
(package-UID), version-range, {"name lang_1", "name lang_n", ...}
```

Обратите внимание, что PID указывается в круглых скобках. Символьные имена `name lang_N` поддерживают локализацию и используются в информационных целях.

Следующий пример демонстрирует объявление зависимости от пакета библиотеки P.I.P.S. (в Symbian 9.1-9.2 она не предустановлена).

```
(0x20009A80), 1, 0, 0, {"Symbian OS PIPS", "Библиотека PIPS для Symbian OS"}
```

Если при инсталляции SIS-дистрибутива системный установщик обнаружит, что в нем объявлены зависимости от отсутствующих пакетов, то будет выведено диалоговое окно с предупреждением, показанное на рис. 3.6.

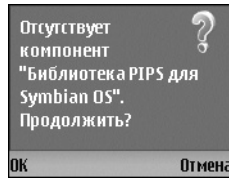


Рис. 3.6. Предупреждение о нарушении зависимостей во время установки пакета

Добавление файлов

Чтобы поместить файл в SIS-пакет, вы должны добавить в PKG-скрипт строку следующего вида:

```
"source-filename" - "destination-filename"[, install-options]
```

где параметр `source-filename` — путь (относительный или абсолютный) и имя файла, помещаемого в дистрибутив. Параметр `destination-filename` — путь и имя файла для установки на устройство. В качестве литеры диска можно использовать восклицательный знак. Если SIS-пакет содержит хотя бы один такой файл, то в этом случае перед инсталляцией пользователю будет выведено диалоговое окно с предложением выбрать носитель (память телефона или карту памяти) для размещения файлов приложения (рис. 3.7).

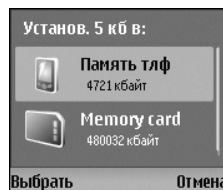


Рис. 3.7. Предложение выбрать носитель для файлов при установке пакета

Например, добавим в наш пакет следующие два файла.

```
"..\gfx\siren.wav"-"C:\Data\Sounds\siren.wav"
```

```
"$(EPOCROOT)Epos32\release\$(PLATFORM)\$(TARGET)\application.exe"  
-"!:\sys\bin\application.exe"
```

В соответствии с платформой безопасности, файлы также могут быть помещены в каталог `\private\<SID>\` любой из содержащихся в SIS-пакете программ. Например, если наше приложение `application.exe` имеет SID, равный `0x200000FE`, то мы сможем поместить файл `siren.wav` в его приватный каталог.

```
"..\gfx\siren.wav"-"!:\private\200000FE\siren.wav"
```

```
"$(EPOCROOT)Epos32\release\$(PLATFORM)\$(TARGET)\application.exe"  
-"!:\sys\bin\application.exe"
```

Обратите внимание, что составляющее путь к приватному каталогу значение SID записано в *шестнадцатеричном* виде и без префикса "0x".

В случае если параметр *destination-filename* окажется пустым, файл войдет в состав дистрибутива, но на устройство устанавливаться не будет.

Параметр *install-options* задает необязательные параметры, перечисленные через запятую, определяющие тип файла и его роль в процессе инсталляции. Можно использовать следующие значения.

- FF или FILE — обычный файл. Просто копируется на устройство.
- FT или FILETEXT — текстовый файл в кодировке UNICODE, содержимое которого отображается в ходе установки в специальном диалоговом окне. Этот параметр часто используется для вывода на экран лицензионного соглашения. Для того чтобы определить ход процесса установки после отображения текста, за параметром FT необходимо указать одно из следующих значений:
 - TC или TEXTCONTINUE — диалоговое окно будет содержать одну кнопку **Continue**, после щелчка на которой установка будет продолжена;
 - TA или TEXTABORT — диалоговое окно будет содержать кнопки **Yes** и **No**; после щелчка на кнопке **Yes** инсталляция будет продолжена, щелчок на кнопке **No** приведет к отмене установки приложения.
- FR или FILERUN — указывает на необходимость запустить файл. Следующие дополнительные параметры, помещенные после FR, уточняют, когда он должен быть запущен:
 - RI или RUNINSTALL — запуск во время инсталляции;
 - RR или RUNREMOVE — запуск при деинсталляции;
 - RB или RUNBOTH — комбинирует значение RI и RR.
- Параметр FR позволяет запускать как исполняемые файлы, так и документы. Сразу после запуска файла установка (или деинсталляция) будет продолжена. Для того чтобы установщик ждал окончания выполнения запущенного

файла, прежде чем продолжить процесс установки (деинсталляции), к вышеперечисленным параметрам необходимо добавить RW (RUNWAITEND).

- FM или FILEMIME — аналогичен параметру FR за тем исключением, что файл не запускается, а передается ассоциированному приложению. Для этого сразу после параметра FM должен быть указан MIME-тип файла.



Параметр FILERUN может игнорироваться в self-signed-дистрибутивах. Это зависит от политики безопасности устройства.

Примеры.

1. Отображение лицензионного соглашения из файла `license.txt` с кнопками **Yes** и **No** во время установки. Сам файл `license.txt` содержится в SIS-пакете, но на устройство не копируется.

```
"..\docs\license.txt"-"", FT, TA
```

2. Запуск приложения `application.exe` во время инсталляции SIS-пакета. Системный установщик будет ждать завершения работы приложения, прежде чем продолжить инсталляцию.

```
"$(EPOCROOT)Epos32\release\$(PLATFORM)\$(TARGET)\application.exe"
-"!:\sys\bin\application.exe", FR, RI, RW
```

3. Открытие GIF-файла во время установки.

```
"my_picture.gif"-"", FM, "image/gif", RI
```

Поддержка локализаций в SIS-дистрибутиве распространяется и на содержащиеся в нем файлы. Вы можете включить в пакет несколько файлов (по одному для каждой локализации), а использовать лишь один из них — в зависимости от текущего языка устройства. Файлы должны перечисляться в том же порядке, что и языки в заголовке. Следующий пример позволяет при установке отобразить лицензионное соглашение на родном языке пользователя (предполагается, что пакет поддерживает две локализации — английскую и русскую).

```
{"..\docs\license_en.txt", "..\docs\license_ru.txt"}-"", FT, FA
```

Если ваша программа использует локализованные ресурсы, то вы можете установить на устройство только те из них, которые соответствуют текущему языку. Это позволяет создать дистрибутив приложения с поддержкой множества локализаций, а устанавливать только одну из них. Такой подход практикуется при необходимости уменьшить размер используемой приложением памяти носителя. Предполагается, что пользователь не будет менять язык своего устройства после установки приложения, иначе она не сможет переключаться между своими локализациями (так как необходимые файлы не были установлены). Пример:

```
{"$(EPOCROOT)Epos32\data\z\resource\apps\application.r01",
"$(EPOCROOT)Epos32\data\z\resource\apps\application.r16"}"
-"!:\resource\apps\application.rsc"
```

Во время деинсталляции SIS-пакета все установленные им файлы будут удалены. В то же время, файлы, созданные в результате выполнения самих программ, автоматически не удаляются. Критерии сертификации приложений для Symbian OS требуют, чтобы при деинсталляции приложений после них не оставалось никаких лишних файлов (настройки, временные файлы и пр.). Чтобы соответствовать этому условию, все создаваемые программой файлы, не предназначенные для использования после ее удаления, необходимо хранить в приватном каталоге. Каталог `\private\<SID>\` удаляется вместе со всем содержимым при деинсталляции программы.

Добавление опций

Если файлы вашего приложения можно логически разделить на независимые опциональные компоненты, то можно предложить пользователю во время установки выбрать те из них, которые ему действительно необходимы. Для этого потребуется выполнить два действия: объявить опции и поместить составляющие компоненты файлы в условные блоки.

Опции в PKG-файле объявляются следующим образом.

```
! ("Option1" [, ..., "OptionN"])
```

Если в PKG-файле объявлена поддержка нескольких языков локализаций, то необходимо задать название опции для каждого из них.

```
! ({ "Option_1_lang_1" [, ..., "Option_N_lang_1" ] } [, ..., { "Option_1_lang_M" [, ..., "Option_N_lang_M" ] } ] )
```

После объявления списка опций в PKG-скрипте можно использовать специальные булевы атрибуты с именами `option1`, `option2` и т.д. для определения выбранных пользователем опций. Файлы, составляющие опциональный компонент, следует заключить в условный блок такого вида.

```
IF _condition_
package-body
[ ELSEIF _condition_
package-body
]
[ ELSE
package-body
] ENDIF
```

В качестве примера добавим опцию, позволяющую включать или отключать установку поддержки русской локализации приложения.

```
! ({ "Russian language", "Русский язык" })
```

```
IF option1
"$ (EPOCROOT)Epos32\data\z\resource\apps\application.r16" -"!:\resource\apps\application.r16"
ENDIF
```

```
"$(EPOCROOT)Epos32\data\z\resource\apps\application.r01"}"
- "!:\resource\apps\application.r01"
```

Заметьте, что мы не сделали опциональной установку английской локализации ресурса. Чтобы программа смогла нормально работать — хоть одна локализация должна обязательно попасть на устройство. Диалоговое окно выбора опций, которое будет выведено во время инсталляции, показано на рис. 3.8.

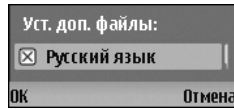


Рис. 3.8. Окно выбора опции дистрибутива

Другие условия инсталляции

Помимо атрибутов опций, в качестве выражения условного блока могут быть использованы ряд других атрибутов и несколько функций. Полный список доступных атрибутов вы найдете в справочнике SDK. С их помощью можно во время установки получить информацию о модели устройства и поддерживаемом аппаратном обеспечении (количество процессоров, их частота, объем памяти, размер экрана и пр.). Например, следующий блок позволяет отобразить сообщение с разъяснениями в том случае, если пакет устанавливается на Nokia 5800 XpressMusic.

```
IF MACHINEUID = 0x2000DA56
"..\\docs\\5800.txt" -"", FT, TC
ENDIF
```

Помимо атрибутов, в условных блоках могут использоваться три функции: `package()`, `exists()` и `appprop()`.

Функция `package(PID)` возвращает `true`, если пакет с таким PID уже установлен в системе. Следующий пример демонстрирует отображение диалогового окна с предупреждением, которое выводится, если на устройстве не установлен пакет с PID = 0x20009A80 (это библиотека P.I.P.S.).

```
IF NOT exists(0x20009A80)
"..\\docs\\no-pips.txt" -"", FT, TC
ENDIF
```

Функция `exists(name)` позволяет проверить существование каталога или файла с именем `name`. Единственное исключение — она не может использоваться для корневых каталогов (например, "e:\"). Функция `exists()` часто применяется для определения версии платформы устройства².

```
IF exists("z:\system\install\Series60v3.0.sis")
; установка файлов, специфичных для платформы S60 3.0
ENDIF
```

² Technical Solution: TSS000052, Technical Solution: TSS000464.

Функция `appprop(PID, propid)` позволяет получать значение свойства `propid` установленного в системе пакета с идентификатором `PID`. Свойства пакета могут принимать только числовые значения и объявляются в PKG-файле следующим образом:

```
+(propid=value,...)
```

где `propid` — номер свойства, `value` — его значение.

Например, закодируем с помощью свойств версию приложения в пакете с `PID = 0x20000001`.

```
+(0=3, 1=2, 2=0); Версия приложения — 3.2.0
```

После того как этот пакет будет установлен в системе, любой последующий устанавливаемый дистрибутив сможет обратиться к нему и проверить значения его свойств с помощью функции `appprop()`.

```
IF appprop(0x20000001, 0) >= 3  
;выполнение действий  
ENDIF
```

Встраиваемые пакеты

Наряду с остальными файлами, SIS-пакет может содержать и другие SIS-дистрибутивы. Они могут быть объявлены в PKG-скрипте следующим образом:

```
@ "source-filename", (package-uid)
```

где параметры `source-filename` — имя SIS-файла и путь к нему (относительный или абсолютный) для добавления в пакет, а `package-uid` — PID встраиваемого SIS-файла.

При деинсталляции пакета все встроенные в него пакеты также удаляются. Исключением является случай, при котором другой пакет объявил зависимость от тех же пакетов — в такой ситуации они не будут деинсталлированы.

Использование встроенных пакетов позволяет поставлять вместе с приложением и используемые им компоненты (возможно, сторонних производителей). Например, следующим образом вы сможете поместить в ваш дистрибутив пакет, содержащий библиотеки P.I.P.S., и при необходимости установить его вместе с вашей программой.

```
IF NOT package(0x20009A80)  
@"..\openc\pips_s60_wp.sis", (0x20009A80)  
ENDIF
```

Установка встроенных пакетов проходит как независимый этап установки приложения. Если установка встроенного пакета завершится ошибкой, то установка всего приложения будет прервана.

Подготовка к сертификации ASD

➤ Знание формата PKG-файлов, используемых для создания SIS-пакетов.

Создание SIS-файла

После того как содержимое и поведение будущего дистрибутива приложения описано в PKG-скрипте, необходимо создать на его основе SIS-пакет. Для этого используется утилита `makesis.exe`, входящая в состав SDK. Вызов `makesis` имеет следующий формат.

```
makesis [-h] [-i] [-s] [-v] [-d directory] pkgfile [sisfile]
```

Ключи:

- `-h` — встроенная справка по формату PKG файлов;
- `-d` — каталог, в котором находится PKG-файл;
- `-i` — отобразить ключи Open SSL;
- `-s` — генерация ROM Stub-файла, который является SIS-файлом специального вида, содержащим лишь служебную информацию и используемым для регистрации в системе предустановленных (например, на карты памяти) приложений;
- `-v` — вывод подробной информации о ходе работы на экран.

Параметр `pkgfile` — имя файла с расширением “`.pkg`”. Часто задается вместе с путем к нему (в этом случае не нужен ключ `-d`).

Параметр `sisfile` — имя генерируемого SIS-пакета. Возможно указание вместе с путем к нему. Если параметр `sisfile` отсутствует, то пакет создается в каталоге PKG-файла с тем же именем.

Следующий пример демонстрирует создание SIS-файла из скрипта `application.pkg` (текущий каталог — каталог SIS-проекта).

```
makesis -v application.pkg application.sis
```

Подписывание SIS-файла сертификатом

Как вы помните, Symbian 9.x не допускает установку SIS-файлов, не подписанных цифровым сертификатом. Цифровой сертификат используется для подтверждения доступа к декларированным в исполняемых файлах и библиотеках защищенным возможностям. Цифровой сертификат представляет собой пару файлов с расширениями “`.cer`” — собственно сертификат и “`.key`” — его закрытый ключ. Виды сертификатов и их получение уже обсуждались в *главе 1*, раздел “Установка приложений и сертификаты”.

Создание сертификата типа `self-generated`

Сертификат-пустышка позволяет приложению удостоверить доступ лишь к тем защищенным возможностям, которые отмечены в системе в качестве “пользовательских”. Для его создания необходимо воспользоваться консольной утилитой `makekeys.exe`, входящей в состав SDK. Формат использования утилиты `makekeys` выглядит следующим образом.

```
makekeys -cert [-v] [-password <password>] [-len <key-length>] -dname  
<distinguished-name-string> <private-key-file> <public-key-cert>
```

Назначение ключей здесь следующее.

- `-cert` — указывает на необходимость создания self-generated сертификата.
- `-v` — включает режим вывода дополнительной информации о ходе работы на экран.
- `-password` и его аргумент — задают пароль, используемый для шифрования сертификата и его закрытого ключа.
- `-len` — длина ключа (от 512 до 4096, по умолчанию 1024).
- `-dname` — позволяет задать информацию о владельце сертификата (так называемый *distinguished name*). Информация задается в виде заключенной в двойные кавычки строки вида `"KEY=VALUE [KEY=VALUE ...]"`. Ключи могут быть следующими:
 - CN — имя;
 - OU — отдел;
 - OR — организация;
 - LO — город;
 - ST — штат;
 - CO — страна (двузначный код);
 - EM — E-mail адрес.

При задании информационной строки для ключа `-dname` необходимо использовать как минимум два ключа.

- Параметры `<private-key-file>` и `<public-key-cert>` являются именами получаемых файлов сертификата и закрытого ключа.

Пример создания сертификата типа self-generated.

```
makekeys -cert -password abcdef -dname "CN=My Name OR=My Company
CO=RU EM=my@email.ru" my.key my.cer
```

Во время создания ключа вас попросят подвигать мышью или понажимать клавиши для инициализации данных случайными значениями. Полученный сертификат `my.cer` будет действителен ровно один год с момента создания. Просмотреть закодированную в CER-файле информацию можно с помощью той же утилиты `makekeys`.

```
makekeys -view my.cer
```

Подписывание SIS-файла

Теперь, имея SIS-пакет и сертификат, мы можем подписать наш дистрибутив. Для этого служит утилита `signsis.exe`, входящая в состав SDK. Использовать `signsis` можно следующим образом.

```
signsis [-?] [-c...] [-o] [-s] [-u] [-v] <input sis> [<output sisx>
<certificate> <key> [<passphrase>] ] ]
```

Назначение ключей здесь следующее.

- `-?` — справка.
- `-с...` — позволяет указать используемый при подписывании алгоритм (`-cd` для DSA либо `-cr` для RSA).
- `-o` — вывод на экран информации о содержащихся в SIS-файле цифровых подписях.
- `-s` — подписывание SIS-файла (необходимо указать сертификат, ключ и пароль).
- `-u` — удаление последней цифровой подписи из SIS-пакета. Позволяет получить неподписанный SIS-файл (Если файл подписан множеством сертификатов, потребуется вызвать утилиту несколько раз.)
- `-v` — включает режим вывода дополнительной информации о ходе работы на экран.
- Параметр `<input sis>` — путь и имя SIS-пакета, полученного с помощью утилиты `makesis`.
- Параметр `<output sisx>` — имя подписанного SIS-пакета.
- Параметры `<certificate>`, `<key>` и `<passphrase>` — соответственно файлы сертификата, закрытого ключа и пароль, использованный при их получении.

Использование утилиты `makesis` для подписывания SIS-файла `self-generated`-сертификатом, созданным нами в предыдущем примере.

```
signsis -s my.sis my.sisx my.cer my.key abcdef
```

Подписанный дистрибутив появится в том же каталоге с расширением `“.sisx”`. Расширение `“.sisx”` часто используется разработчиками для того, чтобы отличить неподписанный SIS-пакет от подписанного. Зачастую во время дистрибуции созданного приложения расширение вновь меняют на `“.sis”`. Это связано с тем, что некоторые каналы распространения контента и веб-порталы для мобильных устройств были созданы еще в те времена, когда SIS-файлы можно было распространять неподписанными.

Проблемы, часто возникающие при установке

Подавляющее большинство ошибок, возникающих при установке SIS-дистрибутивов, так или иначе связано с платформой безопасности. Их можно классифицировать следующим образом.

1. Попытка поместить файл в недопустимый каталог (например, в приватный каталог другого приложения или на диск `Z:`).
2. Попытка заменить какой-либо файл, уже установленный на устройство. Обратите внимание на правила именования файлов EXE и DLL, описанные в главе 6, раздел “Именованье исполняемых файлов, смена идентификаторов”.

3. Попытка установить исполняемый файл с нулевым или уже используемым другим исполняемым файлом значением идентификатора SID.
4. Список защищенных возможностей, к которым декларируют обращение содержащиеся в пакете исполняемые файлы и библиотеки, не покрывается сертификатами.
5. Срок действия сертификата истек или еще не наступил. Проверьте текущую дату на устройстве. Проблема может возникнуть с сертификатами разработчика, полученными по программе Open Signed Offline. Иногда из-за труднообъяснимого воздействия часовых поясов работоспособность сертификата начинается не с момента создания, а на следующий календарный день. В этом случае необходимо перевести системную дату на день вперед.
6. SIS-файл может быть установлен только на устройства с определенными номерами IMEI.

Перечень различных сообщений об ошибке, возникающих во время инсталляции SIS-файлов, а также их возможных причин, можно найти в документе “Troubleshooting Installation Errors”³ Википедии Symbian Developer Community.

³ http://developer.symbian.org/wiki/index.php/Troubleshooting_Installation_Errors

ГЛАВА 4

Интегрированная среда разработки Carbide.c++

Немного истории

В предыдущей главе мы подробно рассмотрели сборку проекта и создание SIS-дистрибутивов с помощью инструментов, входящих в состав SDK. Это полезные навыки, необходимые для обнаружения ошибок и более глубокого понимания процесса разработки приложений. На практике же средствами одного SDK при работе с проектами не ограничиваются. Разработка приложений на Symbian C++ занимает довольно много времени, и, конечно же, тратить его на вызов консольных утилит и настройку командных файлов было бы неразумно. Поэтому все разработчики пользуются интегрированными средами.

За длительную историю развития Symbian OS интегрированных сред разработки для языка C++ было создано довольно много. В свое время достаточно популярной была IDE CodeWarrior компании Metrowerks, но она уже не поддерживается ни автором, ни производителями устройств. Тем не менее многие разработчики так привыкли к ней, что используют до сих пор. Также устарела и не поддерживается IDE Borland C++ BuilderX Mobile Studio. Так и не стала достаточно популярной IDE VistaMax компании Wirelaxsoft.

На сегодняшний день безоговорочным лидером среди IDE, предназначенных для Symbian C++ разработчика, является IDE серии Carbide от компании Nokia. Существует две версии этой IDE: Carbide.c++ и Carbide.vs. Первая является независимой IDE на базе Eclipse. Она имеет средства визуального проектирования графического интерфейса — UI Designer, профилировщик, средство отладки приложений на устройстве (TRK) и многое другое. Работает Carbide.c++ не только с SDK для платформы S60, но и с SDK для UIQ.

В отличие от Carbide.c++, среда Carbide.vs устанавливается как дополнение к Microsoft Visual Studio 2003 (Carbide.vs версии 2.x) или Visual Studio 2005 (Carbide.vs версии 3.x). Эта IDE не имеет средств UI Designer и не обеспечивает поддержку TRK.

В 2008 году компания Nokia приняла важное решение и сконцентрировала свои усилия на развитии IDE Carbide.c++. Поддержка Carbide.vs была прекращена. Также решено было отказаться от совместимости с SDK для платформ на

базе Symbian 6.x–8.x, поддерживаемых вплоть до Carbide.c++ версии 1.3. Дистрибутивы IDE Carbide.vs и Carbide.c++ 1.3 можно найти в архивах на страницах портала сообщества Forum Nokia¹.

Благодаря освободившимся ресурсам, вышедшая в декабре 2008 года Carbide.c++ версии 2.0 стала абсолютно бесплатной. На данный момент интегрированная среда разработки Carbide.c++ 2.x поддерживает разработку приложений не только на языке Symbian C++, но и с помощью языка Qt. Скорее всего, Carbide.c++ в скором времени станет единой IDE для большинства языков программирования, используемых на мобильных устройствах.

В этой книге мы рассматриваем использование IDE Carbide.c++ 2.0.x как наиболее популярной, функциональной и перспективной интегрированной среды разработчика на сегодняшний день.

Инсталляция Carbide.c++ 2.x

Скачать дистрибутив Carbide.c++ 2.x (около 200 Мбайт) можно со страниц портала Forum Nokia (соответствующий URL можно найти в *приложении Б*, раздел “Ссылки”). Для работы рекомендуется Windows XP SP2 или Windows Vista. На ПК должен быть установлен транслятор ActivePerl 5.6.1.635 и хотя бы один комплект SDK. Carbide.c++ 2.x поддерживает работу со следующими версиями SDK:

- S60 5-й редакции;
- S60 3-й редакции, FP2;
- S60 3-й редакции, FP1;
- S60 3-й редакции, Maintenance Release;
- UIQ 3.1;
- UIQ 3.0.

При установке вам будет предложено выбрать одну из трех редакций IDE: Developer, Professional или OEM Edition. Все они отличаются комплектацией. В Developer Edition помимо стандартных компонентов входят UI Designer и обеспечивается возможность отладки на устройстве (TRK). В Professional Edition дополнительно предоставляются такие инструменты, как Crash Debugger, Performance Investigator (профилировщик), Dependency Explorer (определение требуемых защищенных возможностей) и CodeScanner (стилистическая проверка кода). Редакция OEM Edition отличается от Professional некоторыми компонентами, используемыми производителями устройств. Обычному разработчику рекомендуется устанавливать Professional Edition.

¹ http://www.forum.nokia.com/Resource_and_Information/Tools/Archive.xhtml

Запуск, интерфейс и рабочее пространство

Запустить Carbide.c++ можно с помощью команды меню Пуск Windows. Во время первого запуска вам будет предложено выбрать папку для хранения **рабочего пространства** (workspace). Вы можете указать любую папку, в пути которой нет пробелов и нелатинских символов. Папка должна располагаться на том же диске, на котором установлен используемый вами SDK, и по умолчанию она называется Workspace. В папке рабочего пространства хранятся настройки интерфейса Carbide.c++ (в папке `\.metadata\`), а также созданные или импортированные проекты. В процессе работы вы сможете создавать новые рабочие пространства и переключаться между ними. Этот механизм позволяет разделять группы проектов, а также использовать несколько версий Carbide.c++ на одном ПК.

Главное окно IDE Carbide.c++ выглядит так, как показано на рис. 4.1. Как видите, это окно состоит из **панелей** (view). Совокупность различных панелей называют **перспективой** (perspective). Перспективы отвечают тому или иному виду деятельности. Их можно настраивать, создавать новые или переключаться между ними. По умолчанию открывается перспектива **Carbide C/C++**, содержащая набор панелей для работы с файлами и исходным кодом проекта. Еще одна перспектива, с которой вы будете часто сталкиваться, называется **Debug** (Отладка).

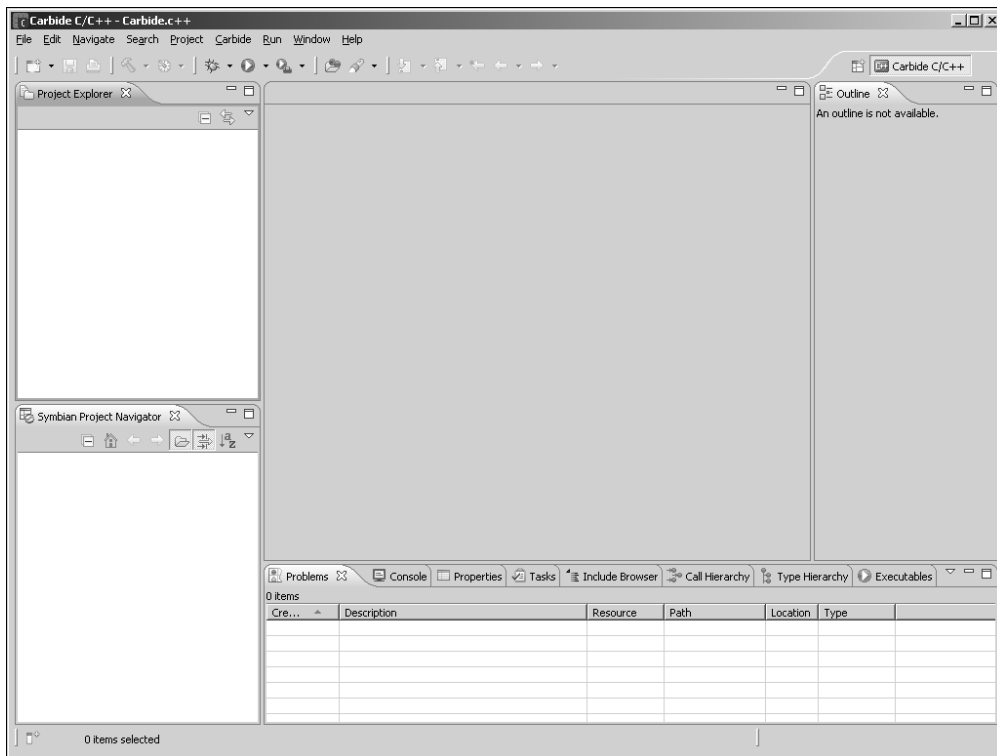



Рис. 4.1. Интерфейс IDE Carbide.c++ 2.x

Открыть новую перспективу можно с помощью команд в подменю **Window⇒Open Perspective** либо с помощью кнопки  в правом верхнем углу окна Carbide.c++. Открытые перспективы отображаются в виде кнопок (рис. 4.2), что позволяет быстро переключаться между ними. Используя контекстное меню кнопок перспектив, можно настроить их внешний вид.

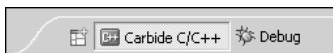


Рис. 4.2. Список открытых перспектив

Состав панели инструментов и некоторых меню может меняться в зависимости от текущей перспективы. Настроить его можно с помощью команды меню **Window⇒Customize Perspective**.

Изменять положение и размер панелей, составляющих перспективу, можно с помощью мыши. При закрытии перспективы положение панелей запоминается. Вернуться к стандартным настройкам панелей позволяет команда меню **Window⇒Reset Perspective**. Можно также сохранить текущий интерфейс в виде новой перспективы с помощью команды меню **Window⇒Save Perspective As**. Добавление новых панелей осуществляется командами подменю **Window⇒Show view**.

Создание и импорт существующих проектов

Создание нового проекта

Для того чтобы создать новый проект, необходимо вызвать команду меню **File⇒New⇒Symbian OS C++ Project**. Сразу после этого вы увидите первое окно мастера создания нового проекта, предназначенное для выбора его типа (рис. 4.3). Содержимое этого окна формируется в Carbide.c++ на основании установленных в системе SDK (в состав SDK входят необходимые для этого конфигурационные файлы).

Список обнаруженных Carbide.c++ SDK и их параметры всегда можно просмотреть в окне, открываемом после выбора команды меню **Windows⇒Preferences**, на вкладке **Carbide.c++\SDK Preferences**. Тип проекта, по сути, определяет используемый при его создании шаблон. Вы всегда можете “переделать” проект одного типа в другой или создать на его основе нечто свое.

Основными типами проектов Symbian C++ являются следующие.

- **Basic console application (EXE)** — проект консольного приложения helloworld.
- **Basic dynamically linked library (DLL)** — проект DLL-библиотеки, экспортирующей несколько методов.
- **Basic static library (LID)** — шаблон статической библиотеки.
- **Empty Project for Symbian** — проект, содержащий лишь файлы `bld.inf` и `.mmp`.

Остальные типы проектов предназначены для создания GUI приложений и являются платформено-зависимыми.

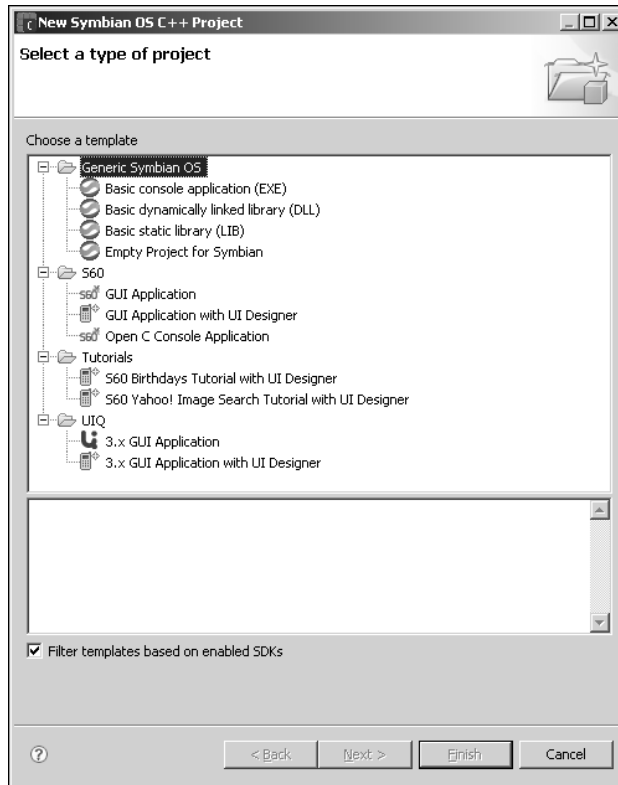


Рис. 4.3. Выбор типа нового проекта

После выбора нужного типа проекта вам будет предложено ввести его имя (без пробелов и только латинские символы) и местоположение (рис. 4.4). Папка проекта должна располагаться в папке текущего рабочего пространства. Лучше местоположение проекта не изменять, в этом случае он будет помещен в папку `<workspace_path>\<project_name>\`.

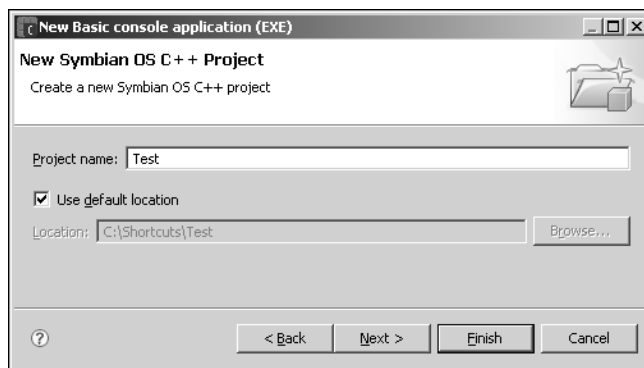


Рис. 4.4. Ввод названия нового проекта

В следующем окне мастера (рис. 4.5) потребуется указать список SDK и целевых платформ, для которых можно будет собирать новый проект. Этот выбор впоследствии всегда можно будет изменить в окне, открываемом после выбора команды меню **Project**⇒**Properties**, на вкладке **Carbide.c++\Build Configurations**. О том, что такое целевая платформа, уже рассказывалось в *главе 3*, раздел “Компиляторы, платформы и режимы компиляции”.

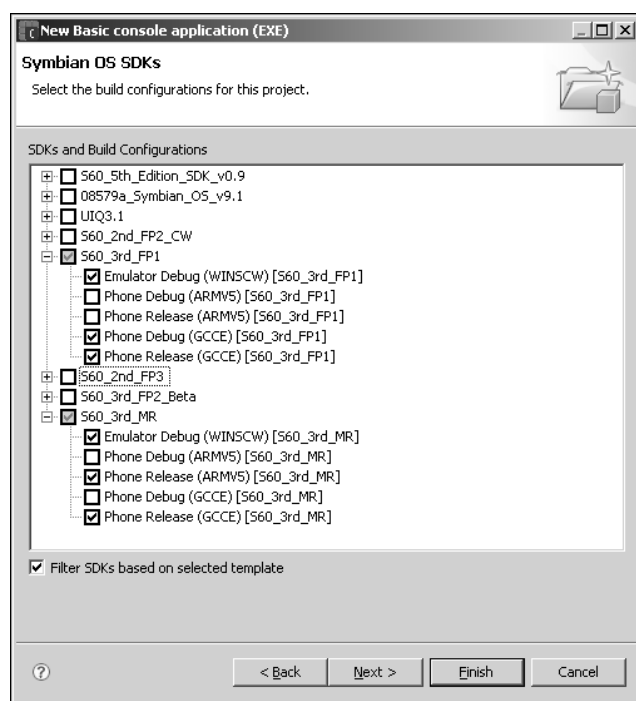


Рис. 4.5. Выбор целевых платформ для сборки нового проекта

В следующем окне мастера создания нового проекта (рис. 4.6) предлагается ввести значение UID3 создаваемого приложения, информацию об авторе и копирайт. Информация об авторе и копирайт будут вставляться в качестве комментария в начале каждого созданного в Carbide.c++ файла исходного кода. Значение UID3 по умолчанию выбирается из тестового диапазона. Поменять его в дальнейшем можно только вручную в ММР-файле проекта (см. *главу 6*, раздел “Именованье исполняемых файлов, смена идентификаторов”). В проектах GUI-приложений UID3 используется не только в ММР-файле, но и в некоторых ресурсах, заголовочных файлах и даже файлах, создаваемых в UI Designer. Если вы планируете впоследствии сертифицировать программу и вам не хочется искать и изменять UID3 во всех этих файлах, то, возможно, в этом случае стоит озаботиться получением значения из защищенного диапазона заранее (см. *главу 7*, раздел “Резервация идентификаторов”).

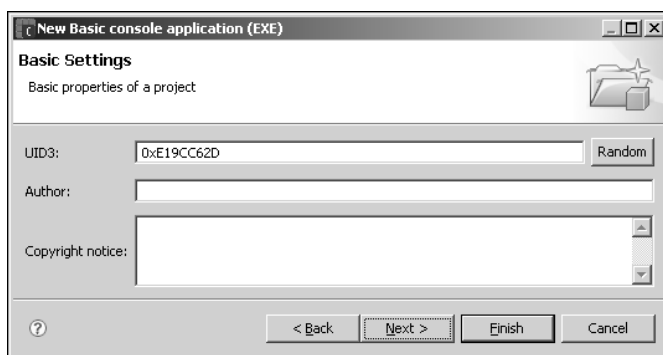


Рис. 4.6. Ввод UID3 проекта

Очередное окно мастера (рис. 4.7) позволяет изменить имена каталогов структуры проекта. Я настоятельно не советую этого делать. На этом работа мастера создания нового проекта заканчивается.

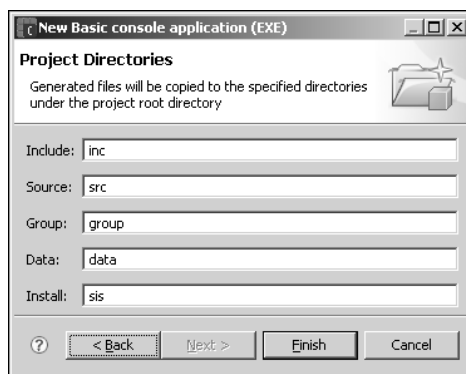


Рис. 4.7. Окно мастера для изменения названий каталогов структуры проекта

Импорт существующего проекта

Импорт существующего проекта осуществляется с помощью команды меню **File⇒Import**. После ее выбора откроется диалоговое окно со списком всех подключенных к IDE мастеров импорта информации. Так как Carbide.c++ базируется на IDE Eclipse, то ею унаследовано довольно большое число мастеров. Нас интересуют только два из них: Symbian OS Bld.inf file и Symbian OS Executable (рис. 4.8).

Мастер импорта Symbian OS Executable позволяет загрузить для последующей отладки исполняемый файл или библиотеку. Импортированные файлы отображаются в панели **Executables**. Этот механизм используется только в том случае, если у разработчика нет исходного кода бинарного файла, но есть символичный (".sym") файл с отладочной информацией для него. На практике вам импорт исполняемых файлов вряд ли потребуется, и подробнее останавливаться на нем мы не будем.

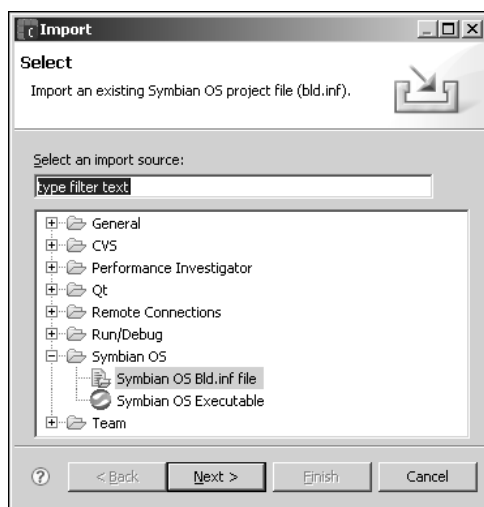


Рис. 4.8. Импорт существующего проекта

Мастер импорта Symbian OS Bld.inf file позволяет подключить к рабочему пространству Carbide.c++ новый проект и включает четыре этапа. Сами файлы проекта при этом в корневой каталог рабочего пространства не перемещаются. На первом этапе работы мастера вы должны выбрать файл `bld.inf` импортируемого проекта. Затем вам будет предложено указать поддерживаемые им целевые платформы для сборки (см. рис. 4.5). На третьем этапе потребуется выбрать компоненты проекта, над которыми планируется работать (рис. 4.9). Компонентами являются ММР-файлы и сборочные скрипты, полученные из указанного файла `bld.inf`. Компоненты проекта, не отмеченные на этом этапе, будут игнорироваться во время сборки. Изменить свой выбор можно будет и после импорта проекта, в диалоговом окне, открывающемся при выборе команды меню **Project**⇒**Properties**, на вкладке **Carbide.c++\Project Settings**.

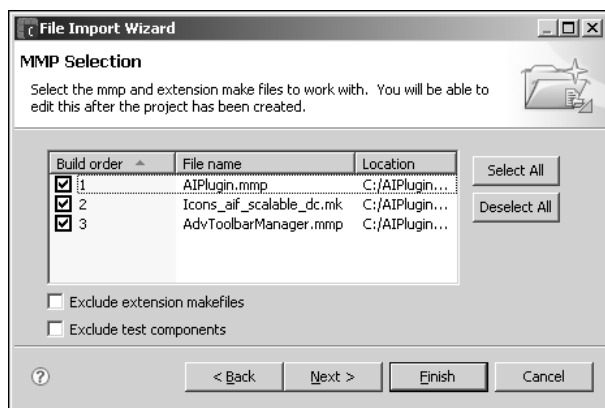


Рис. 4.9. Выбор компонентов импортируемого проекта

Наконец, на заключительном четвертом этапе, необходимо будет указать название проекта и его корневой каталог. Корневой каталог проекта определяется автоматически, и менять его не следует. Название проекта должно быть уникальным в рамках текущего рабочего пространства.

В том случае, если импортируется проект GUI-приложения, созданный в IDE Carbide.c++ 1.x, то вам может быть предложено внести ряд изменений в его исходный код. При этом все необходимые изменения отображаются в специальном окне (рис. 4.10). Рекомендуется соглашаться со всеми требованиями мастера импорта Carbide.c++.

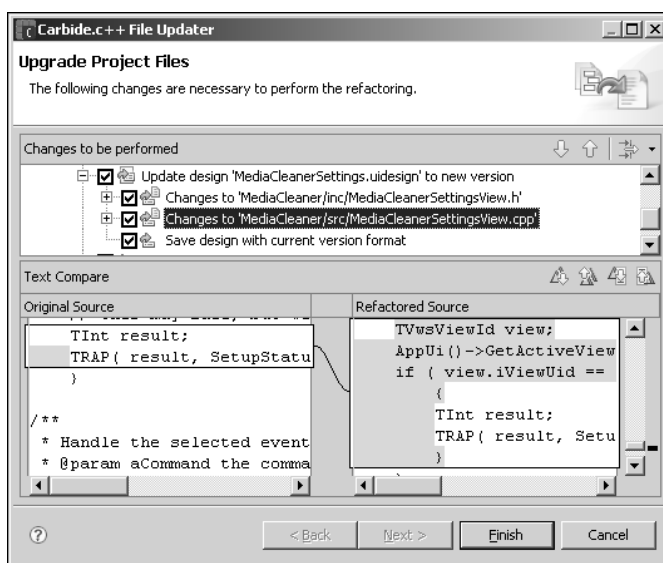


Рис. 4.10. Обновление файлов проекта

Работа с проектом

Навигация

После того как был создан новый или импортирован существующий проект, он будет отображен в навигационных панелях в левой части окна Carbide.c++. По умолчанию в перспективе Carbide C/C++ отображаются только две навигационных панели: **Project Explorer** (рис. 4.11) и **Symbian Project Navigator** (рис. 4.12).

Панель **Project Explorer** отображает структуру всех проектов текущего рабочего пространства в виде дерева. Используемые для обозначения элементов пиктограммы обозначают тип файлов или каталогов. Содержимое каталогов с маркером “C” считается исходным кодом. Такие файлы индексируются для обеспечения быстрого поиска и перехода между объявлениями методов и классов.

Каталог **Includes**, представленный в дереве проекта, является виртуальным и содержит список всех заголовочных файлов, используемых для сборки проекта. С помощью меню **Project Explorer** (кнопка со стрелкой в правом верхнем углу панели) можно настроить фильтрацию файлов определенного типа, а также группировку проектов (working set).

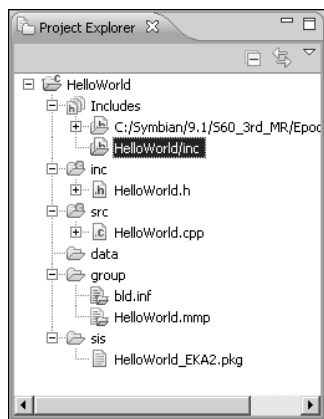


Рис. 4.11. Панель *Project Explorer*

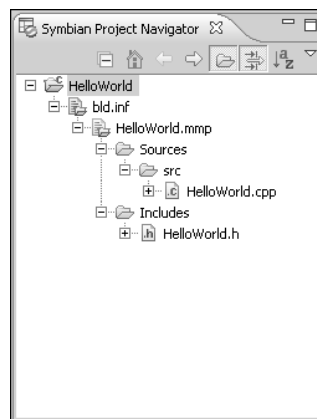


Рис. 4.12. Панель *Symbian Project Navigator*

В отличие от **Project Explorer**, панель **Symbian Project Navigator** предлагает для навигации дерево компонентов проекта. Она может быть полезна в тех случаях, когда требуется определить, в каком из компонентов проекта используются те или иные файлы исходного кода. IDE Carbide.c++ позволяет отобразить в перспективе еще одну навигационную панель — **Navigator**, на которой отображаются все файлы проекта без исключения. На практике же для работы вполне достаточно одной панели — **Project Explorer**.

Любой открытый в панелях навигации проект может быть закрыт или удален. Закрыть проект можно с помощью команды меню **Project⇒Close Project**. В этом случае все открытые на редактирование файлы проекта закрываются, а дерево структуры проекта в навигационных панелях сворачивается до корневого элемента. Продолжить работу с проектом в этом случае можно, выполнив обратную операцию, — выбрав команду меню **Project⇒Open Project**. При закрытии проекта каких-либо изменений в его файлы не вносится, процедура закрытия лишь позволяет сэкономить пространство в панелях навигации и защитить проект от случайного изменения.

Удаление проекта выполняется с помощью команды **Delete**, выбираемой в контекстном меню пиктограммы проекта. При этом Carbide.c++ попросит уточнить, желаете ли вы удалить файлы проекта, либо намерены просто убрать его из рабочего пространства. В случае, если файлы проекта не удаляются, впоследствии его можно будет импортировать через файл **bld.inf** еще раз. Однако операция удаления проекта, повлекшая удаление файлов с диска, необратима.

Работа с файлами проекта

Содержащиеся в проекте файлы могут быть открыты для редактирования в центральной части окна Carbide.c++. Для этого достаточно дважды щелкнуть на пиктограмме требуемого файла. При этом выбор редактора, используемого для обработки открываемого файла, зависит от типа последнего. Можно открыть сразу несколько файлов и переключаться между окнами их редакторов посредством щелчка на корешке соответствующей вкладки (рис. 4.13). Звездочкой в корешке вкладки помечены те файлы, содержимое которых было изменено, но еще не сохранено.

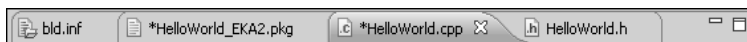


Рис. 4.13. Корешки вкладок окон редакторов открытых файлов проекта

Файл проекта также может быть открыт во встроенном текстовом редакторе или приложении, ассоциированном с ним в Windows. Для этого необходимо воспользоваться командой **Open with** в контекстном меню открываемого файла. Быстро перейти к файлу проекта в окне проводника Windows можно с помощью команды **Show in Explorer**.



IDE Carbide.c++ включает подсистему *контроля версий* файлов. С ее помощью можно легко выявить или “откатить” внесенные изменения. Для этого достаточно воспользоваться командой **Local History**⇒**Compare With** или **Local History**⇒**Replace With** в контекстном меню пиктограммы файла в навигационной панели. Более того, с помощью команды **Local History**⇒**Restore from** в контекстном меню каталога можно даже восстановить удаленный ранее файл.

Чтобы создать в проекте новый файл, нужно либо сделать это средствами Windows и обновить проект (команда меню **Project**⇒**Update Projects**), либо воспользоваться одной из команд подменю **New** в контекстном меню проекта. Последний вариант предпочтительнее, так как в этом случае вам, в зависимости от типа файла, будет предложено воспользоваться одним из имеющихся шаблонов. Например, при создании файла типа **Symbian OS C++ Class** мастер позволяет использовать шаблоны C-класса либо активного объекта.

Все операции по копированию, перемещению и удалению файлов, как в рамках одного проекта, так и между различными проектами, также лучше всего осуществлять средствами Carbide.c++. В этом случае IDE предложит вам автоматически внести необходимые изменения в другие файлы проекта. (Например, при удалении файла исходного кода ссылка на него должна быть убрана из соответствующего MMP-компонента.)

Панель Console

Панель **Console**, пожалуй, самая простая и в то же время одна из самых необходимых панелей Carbide.c++. Она расположена в нижней части экрана и используется для вывода различной текстовой информации. В частности, в ней отображается информация о ходе сборки проекта и его отладке.

В каждый момент времени панель **Console** отображает содержимое одной из открытых консолей, название которой выводится сразу под заголовком панели. В IDE Carbide.c++ консоли представляют информацию различного типа. Некоторые из них агрегируют информацию из других панелей. Переключаться между открытыми в панели консолями можно с помощью меню, раскрываемого щелчком на кнопке  **Display Selected Console**. Для того чтобы открыть новую консоль определенного типа, следует воспользоваться командами в меню кнопки  **Open Console**. В большинстве случаев консоли открываются и переключаются автоматически.

Работа с исходным кодом

Редактирование исходного кода осуществляется с помощью двух панелей: **C/C++ Editor** и **Outline**.

Панель **C/C++ Editor** располагается в центральной части окна Carbide.c++ и используется для отображения содержимого редактируемого файла. Каждый открываемый файл отображается в собственном окне редактора C/C++, переключение между которыми осуществляется с помощью корешков вкладок (рис. 4.14). Отображаемый в окне редактора исходный код подсвечивается в соответствии с выбранным стилем. Настройки, отвечающие за цветовую схему подсветки, можно найти в окне, отображаемом после выбора команды меню **Window⇒Preferences**, на вкладке **C/C++\Editor\Syntax Coloring**.

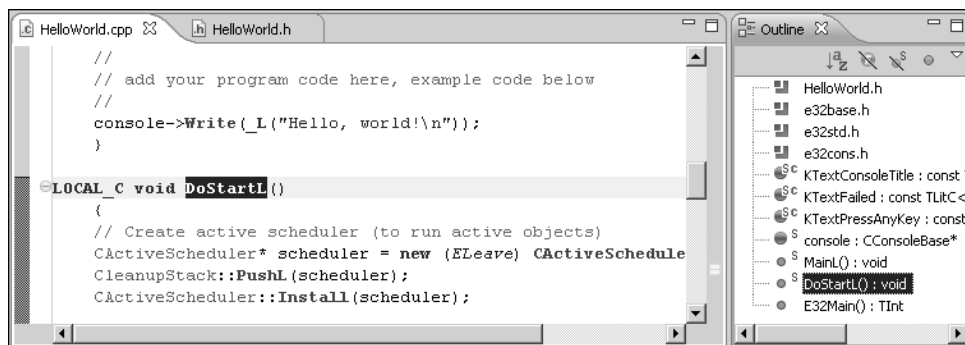


Рис. 4.14. Панели **C/C++ Editor** и **Outline**



Быстрый переход между файлом исходного кода и его заголовочным файлом осуществляется с помощью комбинации клавиш **<Ctrl+Tab>**. При этом, если один из файлов еще не был открыт в редакторе кода, он автоматически откроется.

В правой части окна Carbide C++ находится панель **Outline**. В ней отображается список объявленных в файле типов, классов, их методов и т.д. (см. рис. 4.14). Панель **Outline** позволяет быстро находить эти объекты и переходить к ним в окне редактора кода. Кнопки в верхней части панели позволяют

настроить сортировку, фильтрацию и группировку ее содержимого. Если панель редактора была развернута на весь экран и панель **Outline** не видна, при необходимости можно воспользоваться командой **Quick Outline** в контекстном меню окна редактора: при выборе этой команды содержимое панели **Outline** отображается в открывшемся окне подсказки.

Большинство команд для запуска утилит и форматирования кода имеет горячие клавиши (или позволяет их назначить). IDE Carbide.c++ поддерживает несколько схем горячих клавиш, среди которых такие, как Carbide.c++, Microsoft Visual Studio и Nokia CodeWarrior. В дальнейшем мы будем исходить из того, что вы используете схему Carbide.c++, выбираемую по умолчанию. Для того чтобы изменить или настроить текущую схему соответствия горячих клавиш командам, воспользуйтесь окном, открывающимся при выборе команды меню **Windows⇒Preferences**, вкладка **General\Keys**.

Помимо панели **Outline** для быстрого перехода к объявлению метода или типа можно воспользоваться командой **Open Declaration** контекстного меню элемента. Есть и другой способ: выделите в коде интересующее выражение (например, дважды щелкните на нем мышью) и нажмите <F3>.

Если при наличии выделенного в тексте названия типа или класса нажать <F4>, откроется панель **Type Hierarchy**. В ней будет представлена цепь потомков выбранного класса, объявленные в них методы, а также все классы, от которых он сам был унаследован, и их методы (рис. 4.15). Данная панель также может быть открыта после выбора команды **Type Hierarchy** контекстного меню выделенного элемента. Как и в случае панели **Outline**, если панель редактора была развернута на весь экран, чтобы увидеть содержимое панели **Type Hierarchy** в окне подсказки, воспользуйтесь командой **Quick Type Hierarchy** в контекстном меню окна редактора.

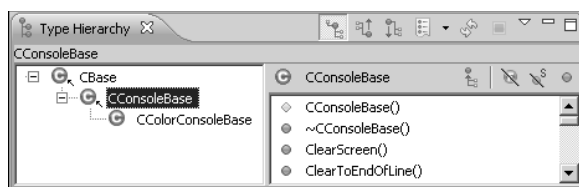


Рис. 4.15. Панель **Type Hierarchy**, открытая для класса *CConsoleBase*

Еще одним полезным инструментом является панель **Call Hierarchy**, в которой отображается список всех методов, вызывающих данную функцию (рис. 4.16). Чтобы открыть эту панель, воспользуйтесь командой **Call Hierarchy** в контекстном меню, предварительно выделив имя требуемой функции в тексте кода.

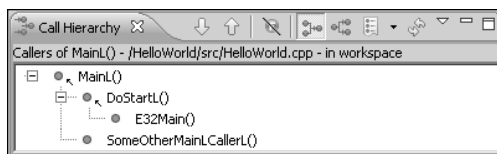


Рис. 4.16. Панель **Call Hierarchy**, в которой отображены вызовы метода *MainL()*

Команда контекстного меню **Explore Macro Expansion**, выбранная при наличии выделенного в коде названия макроса, позволяет увидеть в открывшемся диалоговом окне его объявление и результат, полученный после его раскрытия (рис. 4.17). Макросы в Symbian C++ встречаются довольно часто (символьные дескрипторы, ловушки, метки для проверки утечек памяти кучи и пр.).

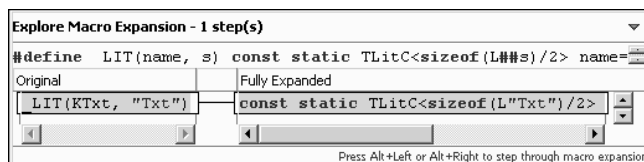


Рис. 4.17. Окно *Explore Macro Expansion* для макроса `_LIT(KTxt, "Txt")`

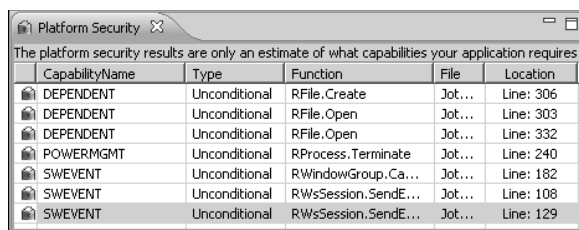
Редактор C/C++ Editor включает ряд функций для форматирования исходного кода. Для этого необходимо выделить участок кода и воспользоваться одной из следующих команд, собранных в подменю **Source** контекстного меню.

- **Comment/Uncomment** (`<Ctrl+/>`) — весь выделенный блок превращается в комментарий с помощью добавления `/*` в начало каждой строки.
- **Add Block Comment** (`<Ctrl+Shift+/>`) — весь выделенный блок превращается в комментарий с использованием `/*` и `*/`. Обратная операция — **Remove Block Comment** (`<Ctrl+Shift+\>`).
- **Shift Right** (`<Tab>`) и **Shift Left** (`<Shift+Tab>`) — сдвиг кода влево или вправо с помощью символов табуляции.
- **Correct Indentation** (`<Ctrl+I>`) — выравнивание сдвига выделенного фрагмента кода в соответствии с текущим стилем форматирования.
- **Format** (`<Ctrl+Shift+F>`) — форматирование выделенного фрагмента кода в соответствии с текущим стилем. Стилль форматирования кода задается в окне **Window Preferences**, вкладка **C/C++\Code Style**, и по умолчанию равен **Symbian OS**.
- **Add include** (`<Ctrl+Shift+N>`) — если команда выбрана при наличии выделенного в коде названия типа или класса, осуществляется поиск его объявления в заголовочных файлах SDK, после чего предлагается подключить один из найденных вариантов к файлу исходного кода с помощью директивы `#include`.
- **Content Assist** (`<Ctrl+Space>`) — при выборе этой команды открывается окно подсказки, содержащее список выражений для вставки. Аналогичное окно автоматически появляется, когда вы начинаете вводить имя класса, имя метода или члена класса.

Еще одна важная группа команд контекстного меню редактора кода находится в подменю **Refactor**. Это функции рефакторинга или реорганизации исходного кода.

- **Rename** (<Alt+Shift+R>) — переименование выделенного названия метода, типа, переменной или макроса. При этом замена имени будет выполнена во всех использующих данный объект файлах проекта. Перед переименованием разработчик может воспользоваться кнопкой **preview** и просмотреть список вносимых изменений.
- **Extract Constant** (<Alt+C>) — позволяет заменить какое-либо значение во всем файле на константу с этим значением. После использования этой функции я рекомендую отредактировать объявление константы в соответствии с правилами Symbian C++.
- **Extract Function** (<Alt+Shift+M>) — добавляет в файл объявление функции, содержащей выделенный фрагмент кода, и заменяет его вызовом этой функции.
- **Hide Method** — переносит объявление метода класса в раздел `private`.
- **Implement Method** — используется только для объявлений методов, не имеющих реализации. Позволяет автоматически создать пустую реализацию для метода.
- **Generate Getters and Setters** — контекстное меню для выбора этой команды должно вызываться на выделенном объявлении члена класса. Позволяет автоматически создать публичные `Get`- и `Set`-методы для его полей.

IDE Carbide.c++ имеет инструмент для анализа исходного кода: **CodeScanner**. Его применение позволяет предупредить или выявить ошибки еще на стадии разработки. Утилита **CodeScanner** может быть запущена из контекстного меню панели **Project Explorer** и осуществляет поиск примитивных ошибок и нарушение конвенции об именовании методов и классов Symbian C++. Результаты работы утилиты **CodeScanner** отображаются в панели **Problems**, но также могут быть экспортированы в HTML- или XML-файл. Соответствующие настройки, а также перечень правил, используемых для анализа исходного кода, можно найти в окне, открываемом при выборе команды меню **Window**⇒**Preferences**, вкладка **Carbide.c++\CodeScanner**. Однако практика показывает, что во всем следовать рекомендациям утилиты **CodeScanner** необязательно, а зачастую и просто невозможно.



The platform security results are only an estimate of what capabilities your application requires.

CapabilityName	Type	Function	File	Location
DEPENDENT	Unconditional	RFile.Create	Jot...	Line: 306
DEPENDENT	Unconditional	RFile.Open	Jot...	Line: 303
DEPENDENT	Unconditional	RFile.Open	Jot...	Line: 332
POWERMGMT	Unconditional	RProcess.Terminate	Jot...	Line: 240
SWEVENT	Unconditional	RWindowGroup.Ca...	Jot...	Line: 182
SWEVENT	Unconditional	RWsSession.SendE...	Jot...	Line: 108
SWEVENT	Unconditional	RWsSession.SendE...	Jot...	Line: 129

Рис. 4.18. Результат работы утилиты *Capability Scanner*

Еще одним полезным инструментом разработчика является утилита **Carability Scanner**. Она предназначена для определения защищенных возможностей, необходимых для работы программы. Запустить ее можно только из



контекстного меню ММР-файла. После сканирования проекта (довольно продолжительного) результат выводится в панели **Platform Security** (рис. 4.18). К сожалению, эта утилита не всегда может точно определить требования декларации доступа к защищенным возможностям для ряда функций, так как они зависят от условий их использования. (В этом случае вместо имени защищенной возможности отображается значение **DEPENDENT**.)

Очистка и заморозка проекта

Как вы помните, очистка проекта позволяет удалить все автоматически создаваемые во время сборки файлы и скрипты. Она выполняется с помощью консольной команды `abld clean`. IDE Carbide.c++ позволяет сделать то же самое для текущей целевой сборки посредством выбора команды меню **Project⇒Clean** или команды **Clean Project** в контекстном меню проекта.

Заморозка (freeze) проекта необходима только в том случае, если одним из компонентов проекта является библиотека, экспортирующая ряд функций. Подробнее об этой процедуре уже было рассказано в *главе 3*, раздел “Заморозка проекта, DEF-файлы”. IDE Carbide.c++ также позволяет выполнить заморозку проекта посредством выбора команды меню **Project⇒Freeze Exports**.

Сборка проекта

Чтобы собрать проект для текущей целевой платформы, достаточно выбрать команду меню **Project⇒Build Project** или щелкнуть на кнопке  **Build** панели инструментов Carbide.c++. При этом будет выполнена команда `bldmake bldfiles <target_platform>`. Раскрывающееся меню кнопки **Build** позволяет одновременно сменить целевую платформу и выполнить сборку проекта. Если вам необходимо только сменить целевую платформу, то сделать это можно после выбора команды меню **Project⇒Active Build Configurations** либо с помощью команд раскрывающегося меню кнопки  **Build Configurations** панели инструментов.

Проследить за тем, какие команды выполняются во время сборки, можно с помощью панели **Console** (консоль C-Build). В нее попадает вся информация, выводимая на экран разнообразными консольными утилитами SDK. После выполнения сборки эта информация анализируется на наличие ошибок. Обнаруженные ошибки отображаются в панели **Problems**. С помощью этой панели можно, дважды щелкнув мышью на ошибке в списке, открыть для редактирования файл, в котором она возникла. Файл, компиляция которого содержала ошибки, также помечается специальным маркером в панели **Project Explorer**.




Старые версии Carbide.c++ иногда могли “не заметить” ошибку, выводимую инструментами SDK. Разработчик мог обнаружить ее самостоятельно в панели **Console**, но при этом панель **Problems** оставалась пустой. Ситуацию усугубляло то, что процесс сборки проекта при возникновении ошибок компиляции не прерывается. И если перед этим проект был хоть раз

успешно собран, то использовались старые результаты компиляции. Все это приводило к тому, что разработчик был уверен в том, что сборка проекта прошла успешно, а при запуске программы обнаруживал, что внесенные в исходный код изменения не вступили в силу. В Carbide.c++ 2.x такой проблемы не замечено. Но если у вас закрадываются сомнения о достоверности содержимого панели Problems, то поищите слово Error в панели Console и выполните очистку проекта перед сборкой.

IDE Carbide.c++ также предоставляет команды меню для сборки проекта сразу для всех целевых платформ и вызова командного файла ABLD с различными аргументами. Все они находятся в меню **Project**. Помимо этого, контекстное меню исполняемого файла содержит команды **Compile** и **Preprocess**, позволяющие скомпилировать или провести предобработку отдельного CPP-файла. Подобного рода функциональность на практике используется чрезвычайно редко. Если же вам потребуется выполнить еще более специфическую операцию, то вы всегда можете открыть командный редактор Windows (команда **Open Command Window** в контекстном меню проекта) и вызвать соответствующий инструмент SDK самостоятельно.

Запуск приложения в эмуляторе

Чтобы запустить приложение в эмуляторе из Carbide.c++, необходимо, чтобы текущей целевой платформой была WINSCW. Поменять ее можно с помощью команды меню **Project**⇒**Build Configurations**⇒**Set Active**. Проект должен быть собран. Если с момента последней сборки под WINSCW в нем произошли какие-либо изменения, то он будет заново собран перед запуском. Если во время сборки возникнут ошибки, разработчику будет предложено отказаться от запуска до их устранения.

Для запуска выполнения приложения достаточно щелкнуть на кнопке  **Run**(<Ctrl+F11>) или воспользоваться командой меню **Run**⇒**Run**. При первом запуске приложения откроется диалоговое окно создания новой конфигурации типа Symbian OS Emulation (рис. 4.19). При последующих запусках приложения для WINSCW созданная конфигурация будет использоваться автоматически.

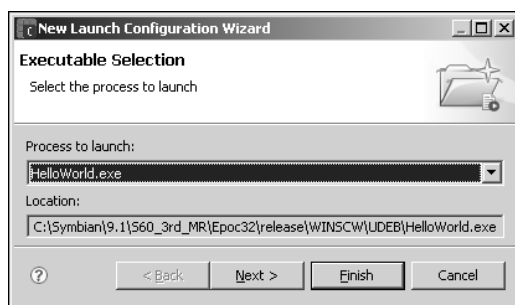



Рис. 4.19. Автоматическое создание первой конфигурации

В диалоговом окне создания новой конфигурации необходимо выбрать запускаемый процесс. По умолчанию им может стать либо один из исполняемых файлов вашего проекта, либо сам Emulator. В первом случае указанная программа будет запущена автоматически сразу после старта эмулятора. Особенностью такого запуска является то, что эмулятор и программа неразрывно связаны, и по завершению работы программы эмулятор закрывается вместе с ней. В этом кроется небольшая опасность. Если программа содержит ошибки, которые не дают ей даже запуститься и ее работа аварийно завершается, то эмулятор закроется вместе с ней, и вы не увидите никаких сообщений о произошедшем сбое. Более того, может возникнуть впечатление, что эмулятор до конца не запустился, и проблема именно в нем, а не в программе.

Если в качестве запускаемого процесса вы выберете Emulator, то будет запущен исключительно эмулятор (ерос.exe), и никакие пользовательские программы после его открытия автоматически не стартуют. В этом случае жизненный цикл эмулятора не связан с запускаемыми в нем процессами, и вы сможете увидеть сообщения о произошедших ошибках. Но здесь кроется другая проблема. Программы в эмуляторе необходимо запускать *вручную*, а для этого они должны отображаться в виде команд в меню **Applications**. Чтобы приложение появилось в этом меню, оно должно быть зарегистрировано в системе с помощью специального файла ресурсов (см. главу 6, раздел “Регистрация программы в меню приложений”). Есть ряд программ, которые не имеют и не должны иметь пиктограмм для запуска — например, серверы, автоматически стартующие при подключении первого клиента. Запустить их вручную вообще не получится.

Все приложения в Carbide.c++ запускаются в соответствии с определенными конфигурациями. **Конфигурация** (run configuration) содержит информацию о запускаемом процессе и различные параметры для его запуска. Чтобы запустить приложение в эмуляторе, оно должно иметь хотя бы одну конфигурацию типа Symbian OS Emulation. Именно такая и создается автоматически вышеописанным способом. Просмотреть и изменить список конфигураций проекта можно в диалоговом окне, открываемом при выборе команды меню **Run**⇒**Run Configurations**. На практике редко возникает потребность иметь более одной конфигурации запуска для эмулятора.

Отладка в эмуляторе

Запуск приложения для отладки в эмуляторе осуществляется с помощью кнопки  **Debug** (<F11>) или команды меню **Run**⇒**Debug**. Также как и во время обычного запуска, активной целевой платформой должна быть WINSCW. Операции отладки в эмуляторе и простого запуска для тестирования разделяют один и тот же набор конфигураций. Поэтому если вы уже создали конфигурацию типа Symbian OS Application, то она и будет использована. В противном случае появится уже описанное выше диалоговое окно создания новой конфигурации (см. рис. 4.19).

После запуска приложения для отладки в эмуляторе окно IDE Carbide.c++ автоматически переключается на использование перспективы **Debug**. При этом навигационные панели исчезают, а в верхней части экрана появляются панели отладки: **Debug**, **Variables**, **Breakpoints**, **Symbian OS Data**, **Registers** и **Modules**. Панель **Debug** (рис. 4.20) позволяет оперировать потоками в запущенном эмуляторе (процессы Symbian OS в Windows вынуждены эмулироваться в виде потоков), а также содержит кнопки для управления пошаговым выполнением программы.

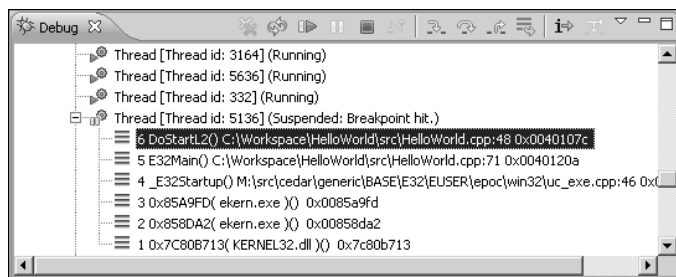


Рис. 4.20. Панель **Debug**

Точки останова (breakpoints), как и во многих других IDE, можно создать, щелкнув слева от строки кода в панели C/C++ Editor. Каждая созданная точка останова отображается в виде синего шарика.

В момент срабатывания точки останова и перехода программы в режим пошагового выполнения в панели **Variables** отображаются все константы, переменные и объекты в текущей области видимости. В левом столбце **Name** выводятся имена, а в правом столбце **Value** — значения и другие характеристики всех перечисленных в панели переменных. При выделении того или иного объекта в нижней части панели выводится его “осмысленное” значение. Например, на рис. 4.21 в панели **Variables** выделен символьный дескриптор **KTextConsoleTitle**. В столбце **Value** приведен адрес памяти, по которому располагается этот объект, а в нижней части панели отображены содержащиеся в дескрипторе данные. Объекты в панели **Variables** можно раскрывать и инспектировать значения их членов. В контекстном меню элементов присутствует команда **Cast To Type**, позволяющая приводить их к другому типу данных. Выбор типа влияет на способ обработки и представления их значений.

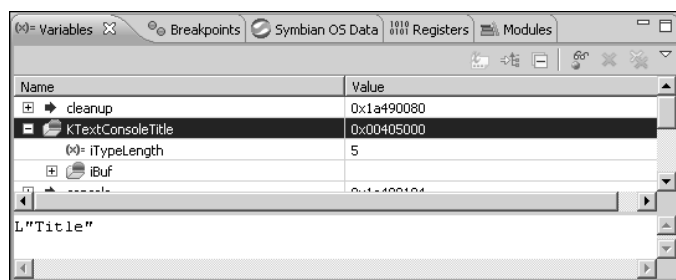


Рис. 4.21. Панель **Variables** с выделенным символьным дескриптором и его значением

Формат отображения числовых значений переменных (десятичный, двоичный, шестнадцатеричный) может быть изменен с помощью команды **Format** контекстного меню. Изменить само значение можно с помощью команды **Change Value** в том же меню.



Есть еще один способ быстро проверить значение переменной — для этого нужно навести указатель мыши на ее имя в панели C/C++ Editor и дождаться появления всплывающего окна подсказки.

Панель **Breakpoints** содержит список всех точек останова в проекте (рис 4.22). С ее помощью их можно отключать без удаления и включать по мере необходимости. Двойной щелчок мышью на точке останова в панели **Breakpoints** вызывает в окне редактора кода переход на соответствующую строку файла.

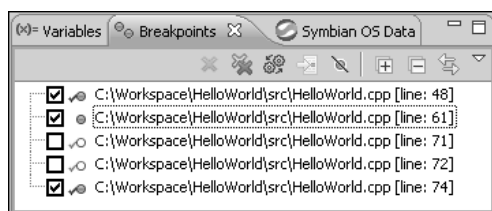


Рис. 4.22. Панель **Breakpoints**

Для каждой точки останова выводится контекстное меню, позволяющее настроить параметры ее срабатывания (команда **Properties**). Разработчик может указать, сколько раз она должна игнорироваться перед срабатыванием (это полезно в циклах или часто вызываемых функциях), условие ее срабатывания, а также выполняемые при срабатывании действия. Такими действиями могут быть проигрывание звукового файла, вывод отладочного сообщения, запуск внешней программы или продолжение работы через определенный промежуток времени.

Для объектов, отображенных в панели **Variables**, можно добавить **точки наблюдения** (watchpoints). Они появляются вместе с точками останова в панели **Breakpoints** и срабатывают во время обращения к объекту (на запись и/или на чтение). Для установки точки наблюдения предназначена команда **Add Watchpoint (C/C++)** в контекстном меню панели **Variables**. Список точек останова и наблюдения можно экспортировать из файла или импортировать в файл.

Панель **Symbian OS Data** отображает системную информацию и работает только при отладке на устройстве. Панель **Registers** позволяет увидеть значение регистров процессора. Во время пошагового выполнения *желтым цветом* подсвечиваются регистры, состояние которых изменилось по сравнению с предыдущим шагом. Наконец, панель **Modules** отображает список загруженных библиотек и системную информацию о них. Панели **Symbian OS Data**, **Registers** и **Modules** редко бывают полезны на практике, поэтому подробнее на них мы останавливаться не будем.

Отладочная информация эмулятора выводится на консоль **Emulator Output**. Она является своеобразным агрегатором данных из других консолей. Это может помочь вам отфильтровать нужную информацию. Например, если вы переключитесь на консоль **PlatSec Diagnostics**, то увидите лишь сообщения о нарушении платформы безопасности.

Сборка SIS-пакета

IDE Carbide.c++ позволяет упростить процесс создания SIS-пакета для собранного проекта. Для этого выберите команду меню **Project**⇒**Properties** и перейдите в открывшемся окне на вкладку **Carbide.c++\Build Configurations**. В этом окне на вкладке **SIS Builder** настраиваются параметры сборки SIS-пакетов. Прежде всего, убедитесь, что в качестве значения параметра **Active Configuration** указана целевая платформа и режим компиляции, которые вы будете использовать при сборке проекта для устройства. Можно также добавить новую конфигурацию, щелкнув на кнопке **Add**, или отредактировать одну из уже существующих, щелкнув на кнопке **Edit**. В любом случае откроется окно **SIS Properties**, показанное на рис. 4.23.

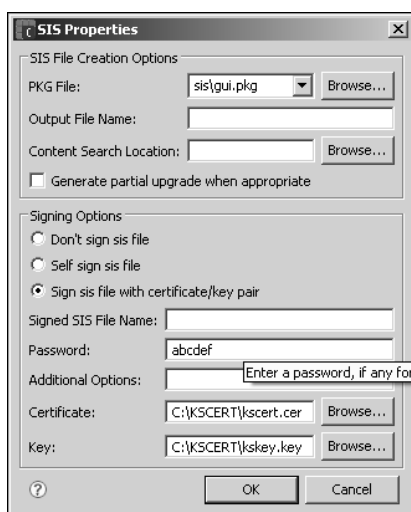


Рис. 4.23. Окно редактора конфигураций SIS Builder

В окне редактора параметров SIS-пакета в списке **PKG File** выберите тот из PKG-файлов проекта, который будет использован для его генерации. В поле **Output File Name** можно изменить имя создаваемого SIS-файла. Эта возможность может пригодиться, если вы используете несколько конфигураций SIS Builder с одним PKG-скриптом для разных целевых платформ.

Группа параметров **Signing Options** позволяет автоматизировать подписывание созданного SIS-пакета. Дистрибутив может быть подписан self-generated-сертификатом (положение переключателя **Self sign sis file**) или сертификата-

том, указанным разработчиком (положение переключателя **Sign sis file with certificate/key pair**). Self-generated-сертификат создается и используется средой Carbide.c++ автоматически. Для использования собственного сертификата вам необходимо указать его местоположение, приватный ключ и пароль. В поле **Signed SIS File Name** можно изменить имя подписанного дистрибутива (по умолчанию это имя PKG-файла с расширением “.sisx”).

После задания новой конфигурации SIS Builder она отображается в соответствующем списке окна **Build Configurations**. Этот список имеет поле **Enabled**, позволяющее включить автоматический запуск создания SIS-файла с помощью выбранной конфигурации сразу после сборки проекта.

Запустить процесс создания SIS-дистрибутива вручную можно после выбора команды **Build PKG File** контекстного меню PKG-файла в панели **Project Explorer**. При этом должна существовать конфигурация для этого PKG-скрипта и текущей целевой платформы, иначе вам предложат ее создать. Результаты работы утилит **makesis**, **makekeys** и **signsis** выводятся на консоль **C-Build**. Возникшие при этом ошибки отображаются в панели **Problems**.

Отладка на устройстве

Как уже говорилось в *главе 3*, в разделе “Работа с эмулятором”, эмулятор не может заменить полноценного тестирования приложения на устройстве. Поэтому в Carbide.c++ предусмотрена возможность отладки программ, выполняющихся непосредственно на смартфоне. Для этого используется специальный модуль **TRK** (Target Resident Kernel), устанавливаемый на устройство и соединяющийся с отладчиком IDE через USB-кабель или через Bluetooth-соединение. Таким образом, отладка приложения на устройстве включает в себя следующие этапы.

1. Установка TRK.
2. Подключение TRK к компьютеру.
3. Запуск приложения для отладки.

Установка TRK

Для того чтобы установить компонент TRK на устройство, нам потребуется его подписанный SISX-дистрибутив. Сам компонент TRK находится в каталоге `C:\Program Files\Nokia\Carbide.c++ v2.0\plugins\com.nokia.carbide.trk.support_<version>\`, где `<version>` — версия компонента (в процессе онлайн-обновления IDE может скачать новые версии). Необходимые нам дистрибутивы расположены в папке `\trk\<platform>\` компонента и имеют названия `<platform>_<platform_version>_app_trk_<trk_version>.sisx`. На используемое для отладки устройство необходимо установить именно ту версию, которая предназначена для его платформы. Например, для смартфона Nokia E90 подойдет дистрибутив `\trk\s60\s60_3_1_app_trk_3_0_9.sisx`. Если вы установите неподходящую версию TRK, то, скорее всего, она работать не будет.

После установки SISX-пакета на устройство, в нем появится новое приложение под названием TRK. Это и есть резидентный модуль отладчика. Приложение TRK не имеет функции автозапуска, поэтому его необходимо будет запускать вручную каждый раз перед отладкой. Во время отладки приложение TRK переходит в фоновый режим. Закрывать модуль TRK до окончания отладки нельзя — в этом случае IDE Carbide.c++ потеряет контроль над запущенными на устройстве процессами.

Подключение TRK к компьютеру

Прежде всего, необходимо подключить запущенное приложение TRK к ПК. Для этой цели может использоваться виртуальный последовательный порт, эмулируемый через Bluetooth или USB-соединение. По умолчанию для связи с компьютером приложение TRK использует Bluetooth. Для этого к ПК должен быть предварительно подключен Bluetooth-приемник и установлены соответствующие драйверы. Стандартные драйверы Windows для этих целей не подойдут, так как они не всегда корректно эмулируют последовательный порт. Сообщество Forum Nokia рекомендует использовать Bluetooth-драйверы WIDCOMM, я же предпочитаю пользоваться приложением IVT BlueSoleil 6.4.x. Для подключения TRK к компьютеру с помощью BlueSoleil необходимо выполнить следующие действия.

1. После установки IVT BlueSoleil, откройте его и загляните в каталог **Мое устройство**. Там вы обнаружите два последовательных COM-порта. Пиктограммы обоих устройств должны быть синего цвета — это означает, что порт свободен.
2. Запустите приложение TRK на смартфоне. Приложение должно автоматически вывести диалоговое окно поиска Bluetooth-устройств. Если этого не произошло — убедитесь, что в окне параметров **Options⇒Settings⇒Connection** выбрано значение **Bluetooth**, а затем воспользуйтесь командой **Connect**.
3. Как только приложение TRK обнаружит Bluetooth-адаптер ПК, выберите его в списке. (Возможно, потребуется выполнить процедуру сопряжения устройств.) После этого должно осуществиться подключение к вашему компьютеру, и на экране приложения TRK появится строка **Status: Connected**. Если вместо этого в поле статуса выводится код ошибки, попробуйте подключиться к компьютеру еще раз.
4. В момент подключения TRK к ПК через IVT BlueSoleil в правом нижнем углу экрана монитора компьютера появится всплывающее окно со словами **<Device> соединен с Последовательный порт COM<n>**, где **<Device>** — имя вашего устройства, а **<n>** — номер виртуального последовательного порта, к которому произошло подключение (запомните его). В том случае, если соединение произойдет с какой-либо другой службой программы BlueSoleil, его необходимо будет разорвать (команда **Options⇒Disconnect** в TRK), затем увеличить номер порта в окне **Options⇒Settings⇒Port**, вновь запустить команду **Options⇒Connect** и вернуться к п. 3.

После успешного соединения TRK с компьютером, пиктограмма одного из последовательных Bluetooth-портов в каталоге **Мое устройство** станет зелено-го цвета, а его состояние изменится на значение **Есть соединение**. Запомните номер используемого порта, он потребуется при создании конфигурации подключения в Carbide.c++.

При подключении TRK к ПК с помощью USB-кабеля на компьютере должны быть установлены соответствующие драйверы. Они входят в состав приложений Nokia PC Suite и Ovi Suite, поэтому вы можете просто установить последнюю версию одного из этих пакетов. После подключения USB-кабеля к устройству, на его дисплей будет выведено диалоговое окно с предложением указать режим соединения: выберите значение **PC Suite**. Далее, запустите TRK. При необходимости отмените предложение выполнить поиск Bluetooth-устройств, в окне **Options⇒Settings⇒Connection** установите значение **USB**, после чего выберите **Options⇒Connect**. После успешного соединения с ПК в окне TRK должно появиться значение **Status: Connected**.

Теперь необходимо определить, к какому именно виртуальному COM-порту подключилось устройство. Для этого в окне “Мой компьютер” Windows запустите приложение Диспетчер устройств. В его окне раскройте категорию **Порты COM и LPT** и найдите в списке COM-порт с названием **<device_model> USB (COM<n>)**, где **<device_model>** — модель устройства смартфона, а **<n>** — номер виртуального последовательного порта (запомните его).

Запуск приложения для отладки

Перед запуском приложения для отладки прежде всего необходимо выбрать его для платформы GCCE или ARMV5 и режима компиляции **UDEB (Phone Debug)**. Отладка на устройстве, в отличие от эмулятора, предполагает установку приложения системным установщиком, а не простое копирование файлов на устройство. Поэтому предварительно необходимо создать SIS-дистрибутив приложения. Удобнее всего позволить утилите SIS-Builder создать его автоматически, сразу после сборки приложения. Далее, имея дистрибутив и убедившись в окне Carbide.c++, что активной платформой является GCCE или ARMV5, можно щелкнуть на кнопке **Debug (<F11>)** для начала отладки приложения на устройстве.

Если запуск приложения происходит впервые, то откроется стартовое окно мастера создания новой конфигурации, включающего три этапа.

1. В первом окне мастера необходимо указать тип конфигурации запуска (рис. 4.24). Доступно пять вариантов:
 - **Application TRK Launch Configuration** — отладка приложения с помощью TRK;
 - **System TRK Launch Configuration** — отладка приложений и компонентов системы с помощью TRK, может использоваться только на специальных отладочных устройствах, так как TRK в этом случае должен быть вшит в прошивку;

- **Trace32 Launch Configuration** — отладка с помощью Trace32;
- **Sophia STI Launch Configuration** — отладка с помощью Sophia STI;
- **Attach to Process Launch Configuration** — подключение к уже запущенному приложению для отладки.

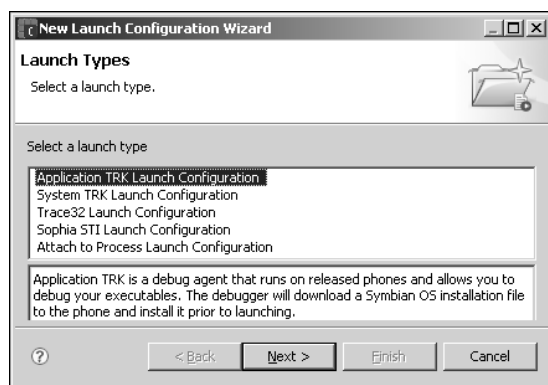


Рис. 4.24. Выбор типа конфигурации запуска

В нашем случае необходимо выбрать первый тип: **Application TRK Launch Configuration**.

- Во втором окне мастера (рис. 4.25) осуществляется выбор используемого соединения с TRK. Если таковое еще не зарегистрировано, необходимо создать новое с помощью специального мастера, щелкнув на кнопке **New**. Также можно отредактировать выбранное соединение, щелкнув на кнопке **Edit**.

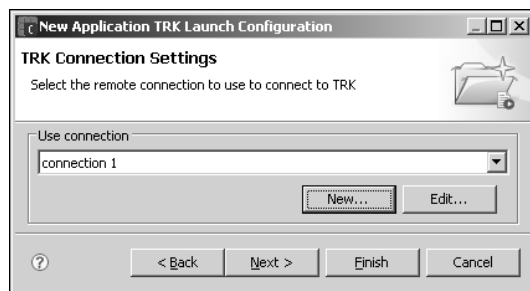


Рис. 4.25. Выбор подключения

Мастер создания нового соединения, в свою очередь, содержит два этапа.

- Выбор типа соединения (**Bluetooth**, **Serial** или **USB**) и его названия.
- Настройка соединения — в этом окне (рис. 4.26) необходимо указать последовательный порт, к которому подключен модуль TRK вашего устройства. Все 256 портов перечислены в раскрывающемся списке **Serial Port**. Некоторые из них содержат пояснения — возможно, рядом с именем порта, к которому подключен TRK, будет отображена модель вашего устройства. Если же это не так, воспользуйтесь номером, кото-

рый вы запомнили во время подключения TRK к компьютеру. Также необходимо правильно указать платформу вашего устройства в раскрывающемся списке **Device OS**, а затем щелкнуть на кнопке **Initiate Service Testing**. Если соединение с TRK произойдет успешно, вам останется лишь завершить работу мастера.

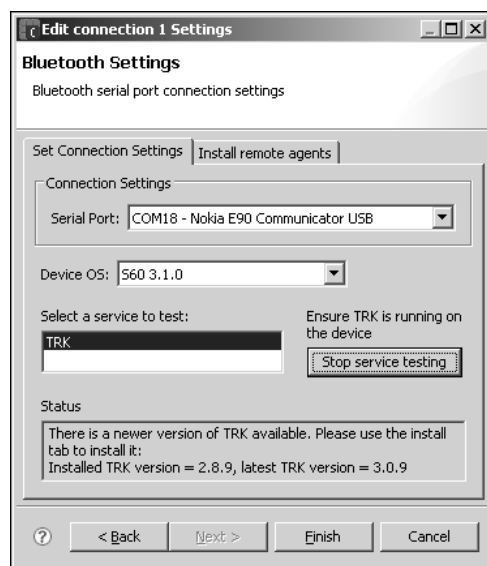


Рис. 4.26. Тестирование соединения с TRK в окне мастера создания нового соединения

- Во время тестирования соединения мастер определяет установленную на устройстве версию TRK и может предложить обновить ее. Для этого в окне мастера перейдите на вкладку **Install remote agents**, выберите подходящую версию TRK из списка (рис. 4.27) и щелкните на кнопке **Install**. Таким же образом может быть выполнена установка TRK на устройство, если на нем данный модуль еще не был инсталлирован.
- 3. В третьем и последнем окне мастера создания новой конфигурации (рис. 4.28) выбирается SISX-дистрибутив, используемый для инсталляции тестируемого приложения на устройство. В раскрывающемся списке **SIS File to Install** будут представлены все пакеты, созданные с конфигурациями в SIS Builder.

После завершения работы мастера конфигураций, SIS-дистрибутив будет автоматически установлен на устройстве и запущен с помощью приложения TRK. IDE Carbide.c++ автоматически переключится на использование перспективы **Debug**, и дальнейшая отладка будет проходить точно так же, как это имеет место в эмуляторе.

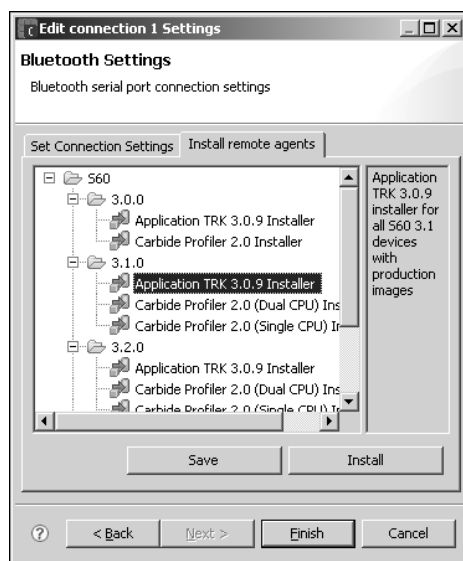


Рис. 4.27. Установка приложения TRK на устройство из окна мастера создания нового соединения

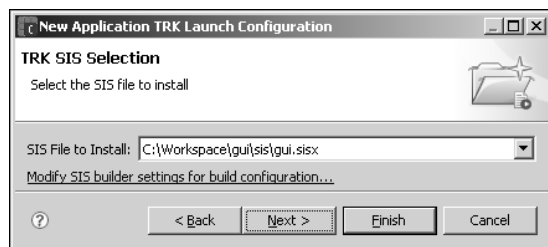


Рис. 4.28. Выбор пакета для установки

Некоторые замечания и советы при отладке на устройстве

- Следует заметить, что устанавливаемый для отладки дистрибутив должен в полной мере соответствовать платформе безопасности. В случае если во время запуска приложения для отладки на устройстве произошла ошибка, прежде всего, необходимо попытаться установить SIS-файл вручную — это позволит определить, вызвана проблема неверными настройками соединения или самим дистрибутивом.
- По умолчанию все отображаемые системным установщиком SIS-файлов диалоговые окна при инсталляции для TRK подавляются. Поэтому ключ `FILETEXT` в PKG-скрипте работать не будет. Изменить это поведение можно следующим образом: на ПК, в окне, открываемом после выбора команды меню **Run** ⇒ **Debug Configurations**, выберите нужную конфигурацию и на вкладке **Installation** сбросьте флажок **Do not show installer UI on phone**.

- После установки приложения один из исполняемых файлов по умолчанию запускается автоматически. В то же время программы с ключом `FILERUN`, `RUNINSTALL` в PKG-файле также запускаются. Это может привести к неприятной ситуации, когда исполняемый файл при отладке на устройстве запускается дважды. При этом лишь один из этих процессов будет управляем с помощью приложения TRK и возможны конфликты при попытках доступа к общим ресурсам. Избежать этого можно, либо убрав ключ автозапуска из пакета, либо изменив автозапускаемый процесс в настройках отладчика. Последнее можно сделать в окне, открываемом после выбора команды меню **Run\Debug Configurations** в поле **Remote process to launch**.
- Во время работы приложения TRK экран устройства постоянно светится. Поэтому вы не сможете протестировать функции, срабатывающие при переходе устройства в режим ожидания (например, заставку).
- Также невозможно с помощью TRK протестировать автозапуск приложений, осуществляемый при включении устройства.

Обновление Carbide.c++

Дистрибутивы новых версий IDE Carbide.c++ периодически публикуются на сайте Forum Nokia. Они содержат последние версии формирующих IDE компонентов и инструментов. В то же время нередко ситуация, когда разработчик либо не может ждать появления следующей версии пакета, либо нуждается в обновлении лишь некоторых компонентов IDE. В этом случае он может воспользоваться функцией онлайн-обновления IDE Carbide.c++.

Чтобы функционировал интерактивный поиск доступных для обновления компонентов IDE Carbide.c++, ПК должен иметь доступ к Интернету. Запуск мастера обновлений выполняется посредством выбора команды меню **Help⇒Software Updates⇒Find and Install**. В открывшемся диалоговом окне вам будет предложено выполнить следующие действия.

1. Во-первых, выбрать способ обновления — необходимо указать, хотите вы обновить уже присутствующие в Carbide.c++ компоненты или же выполнить поиск новых доступных компонентов. Следует выбирать последний вариант — установку новых компонентов.
2. Во-вторых, потребуется указать веб-ресурсы, на которых следует выполнить поиск обновлений. В самом начале предлагаемого списка присутствует значение **Carbide C++ Update Site**. Выберите его и щелкните на кнопке **Finish**, для того чтобы начать поиск доступных обновлений.
3. Через некоторое время вам будут представлены результаты поиска обновлений с разбивкой по категориям. Отметьте те компоненты, которые необходимо установить (рис. 4.29).

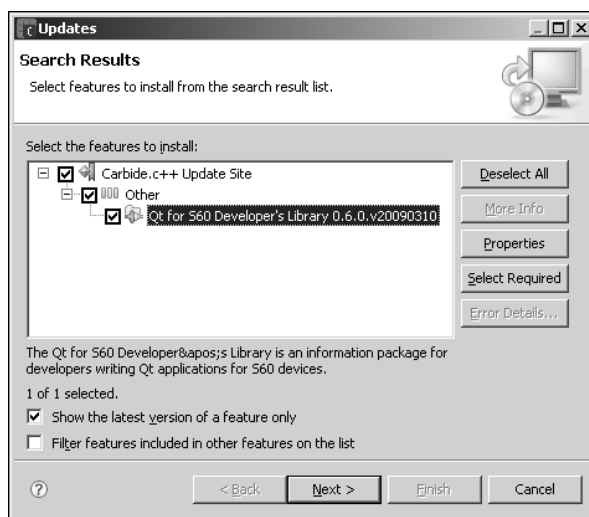


Рис. 4.29. Результаты поиска доступных обновлений

4. Далее потребуется прочесть и принять лицензионное соглашение об использовании устанавливаемых компонентов (обычно каждый из компонентов имеет собственное соглашение, поэтому их может быть довольно много).
5. В очередном окне вам будет предложено указать пути для установки. Рекомендуется не изменять значения, предлагаемые по умолчанию.
6. И наконец, пришло время для скачивания и установки выбранных компонентов. Для того чтобы выполненные изменения вступили в силу, потребуется перезапустить Carbide.c++.

ГЛАВА 5

Основы Symbian C++

Symbian C++ — довольно старый язык программирования и по сути является сильно измененным C++. Он проектировался в те времена, когда стандарта C++ еще не существовало, а возможности компиляторов сильно отличались. Поэтому все фундаментальные типы данных в нем переопределены для обеспечения переносимости исходного кода.

Изначально к Symbian OS предъявлялись беспрецедентные требования в отношении стабильности, ресурсов и энергоемкости. Аппаратные возможности устройств, для которых она предназначалась, были невысоки. В связи с этим, в Symbian C++ уделяется огромное внимание производительности и размеру бинарного кода. Например, первоначально язык Symbian C++ не поддерживал обработку стандартных исключений C++ — вместо него был разработан собственный более эффективный механизм сбросов и ловушек. Для того чтобы избежать расходов памяти на поддержание работы нескольких потоков при реализации многозадачности, в Symbian C++ был введен паттерн активных объектов, позволяющий использовать событийно-ориентированное программирование и асинхронные операции в рамках одного потока. Для того чтобы сделать Symbian OS более устойчивой, вместо небезопасных null-завершенных строк был разработан новый тип данных — *дескрипторы*. Очень серьезно в Symbian относятся и к проблемам утечки памяти или ресурсов.

Как видите, различия между Symbian C++ и C++ довольно существенны. Практика показывает, что даже опытному C++-разработчику бывает сложно освоить этот язык.

Фундаментальные типы данных

Вместо фундаментальных типов C++ в Symbian C++ используются их псевдонимы, определенные с помощью оператора `typedef` в заголовочном файле `e32def.h`. Пример.

```
typedef short int    TInt16;  
typedef long int     TInt32;
```

Подобное переобъявление было произведено для того, чтобы гарантировать переносимость кода между платформами и компиляторами. Во всех случаях

следует использовать новые имена типов данных. Их не сложно запомнить — все они состоят из имени типа данных с префиксом в виде заглавной литеры “T”, что демонстрирует их принадлежность к категории T-классов (о ней мы поговорим несколько позднее). В качестве суффикса может присутствовать размерность данных в битах.

В табл. 5.1 представлены разнообразные целочисленные типы данных Symbian C++ и их аналоги в C++.

Таблица 5.1. Соответствие целочисленных типов данных Symbian C++ и C++

Знаковые целочисленные типы		Беззнаковые целочисленные типы	
Symbian C++	C++	Symbian C++	C++
TInt8	signed char	TUInt8 и TText8	unsigned char
TInt16	short int	TUInt16, TText16, TText	unsigned short int
TInt32	long int	TUInt32	unsigned long int
TInt64	long long	TUInt64	unsigned long long
TInt	signed int	TUInt	unsigned int

Для хранения чисел с плавающей запятой используется тип TReal32 и TReal64 (двойная точность). Сведения об их соответствии типам в C++ приведены в табл. 5.2. Вместо булевых false и true в Symbian C++ используются значения EFalse (равное нулю) и ETrue (равное единице), определенные в файле e32const.h. В качестве булевого типа данных объявлен тип TBool, эквивалентный типу int в C++. Так как тип TBool не является перечислением, то в нем может оказаться значение, отличное от EFalse или ETrue. Как и в C++, в Symbian C++ любое ненулевое значение в условном операторе означает true. Поэтому прямое сравнение с ETrue не рекомендуется. Таким образом, для булевой переменной b вместо if (ETrue == b) лучше использовать if (b).

Таблица 5.2. Прочие фундаментальные типы данных Symbian C++

Symbian C++	C++
TReal32	float
TReal64, TReal	double
TBool	int
TAny*	void*

Тип void переопределен как TAny, но заменяется на TAny он лишь там, где используется в качестве обобщенного указателя (void*). В остальных случаях следует использовать void. Это исключение связано с тем, что разработчики Symbian C++ сочли тип void достаточно компиляторо-независимым. Пример.

```
// Объявление функции в C++
void foo(void* aParam);
// В Symbian C++ должно выглядеть так:
void foo(TAny* aParam);
```

Подготовка к сертификации ASD

- Знание того, как фундаментальные типы Symbian OS соотносятся с типами данных в C++.
 - Понимание того, что фундаментальные типы Symbian OS всегда должны использоваться вместо типов C++ (`bool`, `int`, `float` и пр.), так как они не зависят от компилятора.
-

Соглашение об именовании

В Symbian C++ имеется свод правил, определяющий принципы именования типов данных, классов, методов и даже переменных. Благодаря такому соглашению, лишь по названию объекта разработчик может многое узнать о его поведении и о правилах работы с ним. Я настоятельно советую всегда соблюдать соглашение об именовании — это существенно снижает вероятность того, что вы допустите ошибку, и повышает удобочитаемость кода. Кроме того, несоблюдение данных правил может ввести в замешательство разработчиков, которые, возможно, будут изучать исходный код вашего приложения.

Общее правило

Названия всех объектов, кроме автоматических переменных, должны начинаться с заглавной буквы и могут содержать префикс. Если имя содержит несколько слов, то все они пишутся слитно и начинаются с заглавной буквы (стиль `UpperCamelCase`). Префикс представляет собой один строчный или прописной символ и позволяет быстро определить по названию, с чем вы имеете дело: с аргументом, членом класса и т.д. Имена автоматических переменных должны начинаться со строчной буквы и не могут иметь префикс.

Константы и макросы

Константы в Symbian C++ содержат префикс “K”. Например, вот как объявлена в файле `e32const.h` константа, ограничивающая максимальный размер имени файла:

```
const TInt KMaxFileName=0x100;
```

Имена макросов записываются в верхнем регистре. Пример.

```
#define FOREVER for(;;)
```

В случае, если в названии присутствуют несколько слов, они разделяются с помощью символа подчеркивания “_”. Пример.

```
#define STATIC_CAST(type,exp) (static_cast<type>(exp))
```

Имя макроса также может начинаться с символа “_”. Следующий пример демонстрирует определение и использование макроса `_LIT`, объявляющего символичный дескриптор-константу.

```
#define _LIT(name,s) const static TLitC<sizeof(L##s)/2>
    name={sizeof(L##s)/2-1,L##s}

_LIT(KText, "Text");
```

Обратите внимание, что имя дескриптора, передаваемое в `_LIT`, содержит префикс “K”, так как макрос задает константу.

Большинство названий макросов, используемых для отладки, начинается с двух символов “_”. Таким же образом выделяются идентификаторы, используемые для определения целевой платформы во время препроцессинга кода. Пример.

```
#ifdef __WINS__
#define __DEBUGGER() {if (User::JustInTime()) __BREAKPOINT();}
#else
#define __DEBUGGER()
#endif
```

Классы и члены классов

Все классы в Symbian C++, за исключением статических (содержащих только статические методы), на основании их назначения и способа использования относят к одной из категорий (типов). В зависимости от категории, к которой принадлежит класс, его название содержит тот или иной префикс. Категория позволяет разработчику быстро установить правила, по которым можно безопасно создавать, использовать и уничтожать объект данного класса. Основных категорий четыре: `Type class`, `Class`, `Resource class` и `Mixine class`. Входящие в них классы имеют префиксы T, C, R и M соответственно (обязательно в верхнем регистре). В документации их чаще называют T-классами, C-классами и т.д. Статические классы не принадлежат ни к одной из категорий и не имеют префикса в названии. Более подробно с категориями классов мы познакомимся несколько позднее. На данный момент вам необходимо научиться определять категорию класса.

Например: `CActiveScheduler` — C-класс, `RFileStream` — R-класс, `TBuf` и `TPtr` — T-классы. Но `CleanupStack`, `Console` и `User` — статические классы. Вы не можете создать экземпляр статического класса.

Имена членов класса, независимо от их типа, должны содержать префикс “i” (обязательно в нижнем регистре). Пример.

```
class TAngle3D
{
public:
    TInt iPhi;
    TInt iTheta;
};
```

Структуры

Изначально для структур в Symbian C++ была выделена отдельная категория S-классов, и их названия начинались с префикса “S”.

```
struct SLocaleLocaleSettings
{
    TText    iCurrencySymbol[KMaxCurrencySymbol+1];
    TAny*    iLocaleExtraSettingsDllPtr;
};
```

Но так как структуры C++ удовлетворяли условиям категории T-классов, то впоследствии многие из них были переименованы. Сейчас в системных API вы можете столкнуться как с новым, так и со старым стилем именования. Разработчикам рекомендуется придерживаться нового стиля и использовать префикс “T” в названиях структур.

```
struct TInfo
{
    TUint32 iModuleVersion;
    TUidType iUids;
    TSecurityInfo iSecurityInfo;
};
```

Перечисления и их значения

Все перечисления относятся к категории T-классов и имеют соответствующий префикс в названии. Но для именования элементов перечислений, в отличие от членов настоящих классов, используют префикс “E”. Далее приводятся примеры нескольких объявлений перечислений из файла `e32const.h`.

```
enum TRadix
{
    EBinary = 2, EOctal = 8, EDecimal = 10, EHex = 16
};

enum TTimeFormat
{
    ETime12, ETime24
};
```



В редких случаях в системных API вы можете обнаружить константы с префиксом “E” и перечисления, члены которых начинаются с “K”. Дело в том, что иногда проектировщики API вносили изменения, превращая константы в перечисления, и наоборот. В то же время они хотели избежать несовместимости на уровне кода и сознательно шли на нарушение соглашения об именованиях.

Функции и аргументы

Имена аргументов функции всегда должны содержать префикс “a”.
Пример.

```
TBool IsEqual(TInt aParam1, TInt aParam2)
{
    return aParam1 == aParam2;
}
```

Сами названия функций подчиняются общим правилам именования, за тем исключением, что вместо префикса используют суффикс, состоящий из одного или нескольких символов в верхнем регистре. Такой суффикс позволяет предупредить разработчика об особенностях обработки исключений, работы со стеком очистки в функции и т.п. Следующие типы функций требуют добавления суффикса к имени.

1. Функции, способные вызвать сброс, должны содержать литеру “L” (от Leave) в суффиксе. Пример.

```
void Func1L(TInt aIdx)
{
    if (aIdx < 0) User::Leave(KErrArgument);
}
```

Более подробно исключения в Symbian C++ и их обработка рассматриваются в следующем разделе.

2. Методы, по завершению работы которых в стеке очистки становится больше объектов, должны содержать литеру “C” (от CleanupStack) в суффиксе. Зачастую, такие функции также способны вызвать сброс, и суффикс в имени выглядит как “LC”. Ярким примером такого метода является один из методов двухфазного конструирования.

```
CClassName* CClassName::NewLC()
{
    CClassName* self = new (ELeave) CClassName();
    CleanupStack::PushL(self);
    return self;
}
```

Стек очистки и двухфазное конструирование будет рассмотрено нами позднее.

3. Методы, уничтожающие объект, в котором они вызваны, должны содержать литеру “D” (от Destroy) в суффиксе имени. Пример.

```
void CMyDialog::RunLD()
{
    RunL();
    delete this;
}
```

Подготовка к сертификации ASD

- Умение определять категорию (тип) класса по имени. Понимание того, что префикс в имени класса позволяет быстро установить правила, по которым можно безопасно создавать, использовать и уничтожать объект данного класса.
 - Знание того, что к T-классам относят также фундаментальные типы, структуры и перечисления.
 - Знание того, что статические классы не имеют префикса.
 - Знание о невозможности создания объекта статического класса, так как в нем определены только статичные функции.
 - Знание значения "C", "L" и "D" суффиксов в именах функций.
-

Обработка ошибок и исключений

Сбросы

Большинство методов, ошибка в ходе выполнения которых не критична для работы приложения, возвращает ее код в качестве результата. Коды часто возникающих ошибок являются константами и определены в файле `e32err.h`. Для удобства все их значения меньше нуля. Исключение составляет код отсутствия ошибки `KErrNone`, равный нулю. Следующий пример метода `GetAge()` некоторого C-класса `CClassName` демонстрирует получение возраста (количество полных лет) из массива `iAgeArray` по индексу `aIdx`. Так как возраст не может быть отрицательным, мы можем возвращать из функции как результат, так и код ошибки.

```
TInt CClassName::GetAge(TInt aIdx)
{
    if (aIdx < 0)
        return KErrArgument;
    else if (iAgeArray.Count() <= aIdx)
        return KErrOverflow;
    else
        return iAgeArray[aIdx];
}
```

Если же ошибка в функции является критической, то она может сигнализировать о ней с помощью исключения. В Symbian C++ определен собственный, отличный от C++, механизм обработки исключений: **сбросы** (`leave`) и **ловушки** (`trap`). Сброс является аналогом исключения и содержит код произошедшей ошибки. Вызвать его можно методом `Leave()` статического класса `User`. При этом работа функции немедленно прерывается. Например, сделаем аналог приведенного в предыдущем примере метода с использованием сбросов.

```
TInt CClassName::GetAgeL(TInt aIdx)
{
    if (aIdx < 0)
        User::Leave(KErrArgument);
    else if (iAgeArray.Count() <= aIdx)
        User::Leave(KErrOverflow);
    else
        return iAgeArray[aIdx];
}
```

Обратите внимание, что, так как метод может вызвать сброс, его название теперь содержит суффикс “L”.

Класс User содержит еще несколько методов для вызова сброса.

- `LeaveNoMemory()` — сброс с кодом `KErrNoMemory`.
- `LeaveIfNull()` — вызывает сброс с кодом `KErrNoMemory` лишь в том случае, если аргумент равен `NULL`. В противном случае возвращает указатель `aPtr`.
- `LeaveIfError()` — в случае, если аргумент имеет отрицательное значение, вызывает сброс с таким кодом. Иначе возвращает аргумент.

С помощью этих методов мы можем переписать наш пример следующим образом.

```
TInt CClassName::GetAgeL(TInt aIdx)
{
    return User::LeaveIfError(GetAge(aIdx));
}
```

Ловушки

Произошедший в функции сброс должен быть перехвачен с помощью одного из трех макросов-ловушек: `TRAP`, `TRAPD` или `TRAP_IGNORE`. Макрос `TRAP` позволяет отловить возникший сброс и поместить код ошибки в переданную ему целочисленную переменную. Пример.

```
TInt CClassName::GetAlexAge()
{
    const TInt alex_idx = 3;
    TInt leave_code;
    TRAP(leave_code, TInt res = GetAgeL(alex_idx));
    if (KErrNone != leave_code)
    {
        // Обработка ошибки
    }
}
```

В методе `GetAlexAge()` мы получаем возраст по предопределенному в константе индексу. Заметьте, что в названии метода нет суффикса “L”, так как в нем

ни при каких обстоятельствах не может возникнуть сброс. По завершению выполнения макроса TRAP в переменной `leave_code` будет храниться код ошибки, приведшей к сбросу, либо значение `KErrNone`, если сброс не произошел. Аналогично работает и макрос TRAPD, с той разницей, что переменная для хранения кода ошибки объявляется в нем автоматически. Вам лишь нужно задать ее имя. Пример.

```
TInt CClassName::GetAlexAge()
{
    const TInt alex_idx = 3;
    TRAPD(leave_code, TInt res = GetAgeL(alex_idx));
    if (KErrNone != leave_code)
    {
        // Обработка ошибки
    }
}
```

Макрос TRAP_IGNORE принимает лишь один аргумент — выражение, способное привести к сбросу, и используется в тех случаях, когда вы хотите перехватить возникшее исключение, но не собираетесь его обрабатывать.

```
TRAP_IGNORE(GetAgeL(alex_idx));
```

Сами по себе ловушки относительно ресурсоемки. Их неграмотное использование может ухудшить производительность программы. Рекомендуется проектировать приложение таким образом, чтобы снизить число используемых в нем ловушек до разумного минимума. Например, если вам необходимо сделать сбросо-устойчивым (leave-safe) метод, вызывающий несколько способных привести к сбросу функций, то имеет смысл выделить их вызовы в отдельный метод и использовать одну ловушку. Рассмотрим следующий метод, инициализирующий массив `iAgeArray` тремя значениями. Метод `AppendL()`, добавляющий новый элемент в массив, потенциально способен вызвать сброс, например, если для его хранения не хватает памяти.

```
void CClassName::InitArrays()
{
    <...> // Некоторые операции
    TRAP_IGNORE(iAgeArray.AppendL(25));
    TRAP_IGNORE(iAgeArray.AppendL(33));
    TRAP_IGNORE(iAgeArray.AppendL(37));
    <...> // Некоторые операции
}
```

С точки зрения производительности целесообразнее использовать следующий вариант.

```
void CClassName::FillAgeArrayL()
{
    iAgeArray.AppendL(25);
```

```

    iAgeArray.AppendL(33);
    iAgeArray.AppendL(37);
}

void CClassName::InitArrays()
{
    <...> // Некоторые операции
    TRAP_IGNORE(FillAgeArrayL());
    <...> // Некоторые операции
}

```

Возникновение сброса прерывает выполнение не только той функции, в которой он произошел, но и всех функций из ее стека вызовов, до ближайшей ловушки. Работа программы возобновляется со следующего сразу после ловушки оператора. В этом смысле работа сбросов и ловушек аналогична выражениям `setjmp` и `longjmp` в С. Поэтому вышеприведенные примеры не эквивалентны. В первом случае добавление каждого элемента помещено в отдельный элемент `TRAP_IGNORE` и, вне зависимости от того, было ли успешным добавление первых двух элементов, будет произведена попытка добавить третий. Во втором случае заполнение массива прервется после первого же сброса, и работа программы продолжится сразу после вызова метода `FillAgeArrayL()` со следующего за ним оператора.

Также в целях повышения производительности следует избегать необоснованного использования большого числа вложенных ловушек. Например, если сброс может быть вызван как в самой функции А, так и в вызываемых ею функциях В, С и D, то сделать ее сбросо-устойчивой довольно сложно. Гораздо удобнее поместить в ловушку вызов самой функции А.

Жестких правил использования ловушек нет — только разработчик на этапе проектирования приложения может решить, где и как их использовать. Системные API зачастую предоставляют два варианта одной функции: вариант, возвращающий код ошибки в качестве результата, и другой вариант, вызывающий сброс.

Сброс всегда должен быть пойман ловушкой, в противном случае он приведет к панике (см. следующий раздел).

Начиная с Symbian OS 9.x, Symbian C++ также поддерживает и стандартный способ обработки исключений C++. Более того, сами сбросы и ловушки теперь реализуются на основе механизма `throw\catch`. В результате разработчик теперь может использовать исключения C++ при разработке приложений на Symbian C++. Однако я этого делать настоятельно не рекомендую: если вы откроете макрос-ловушку, то увидите, что помимо блока `try/catch` он содержит обращения к стеку очистки, поэтому ни ловить `throw`-исключения ловушками, ни обрабатывать сбросы в блоке `try\catch` вы не можете. Есть еще ряд серьезных ограничений на смешивание кода, использующего эти механизмы, с которыми вы можете ознакомиться в статье Бена Морриса “Exception Handling in

Symbian OS". В общем случае рекомендуется использовать лишь один из способов обработки исключений во всем бинарном файле. А так как у сбросов и ловушек есть ряд преимуществ (та же работа со стеком очистки, например), то и использовать механизм `throw\catch` имеет смысл разве что при портировании кода C++.

Паника

Паника (panic) — событие, происходящее при возникновении критической ошибки. Выполнение потока, в котором была вызвана паника, немедленно прекращается. Если это главный поток процесса, то вся программа будет аварийно завершена. При возникновении паники в процессе, отмеченном в Symbian OS как “критический для работы системы”, происходит перезагрузка устройства. Чаще всего к панике приводит неправильное использование системных API, — например, передача заведомо неверных значений аргументов. С этой точки зрения ее можно рассматривать как механизм дисциплинирования разработчика. В готовом приложении возникновение паники абсолютно неприемлемо.

В отличие от сброса, паника, помимо кода ошибки, содержит имя категории. В Symbian API определено довольно большое число различных категорий паники (более 50). Зная категорию и код паники, вы можете определить, что именно привело к ее возникновению. Для этого необходимо воспользоваться справочником SDK.

Разработчик может самостоятельно вызвать панику с помощью статического метода `Panic()` класса `User`. При этом вы можете указать собственные категории и код ошибки или использовать существующие общепринятые значения. Название категории должно быть довольно коротким: желательно не более 16 символов. Частой практикой является объявление перечисления, содержащего используемые в паниках и сбросах приложения коды ошибок. Пример.

```
enum TMyPanics
{
    EValueTooBig = -100, EValueTooLow = -101
};

const TInt KMaxAge = 130;

void RaisePanic(TMyPanics aCode)
{
    _LIT(KMyPanicCategory, "Panic in my APP!");
    User::Panic(KMyPanicCategory, aCode);
}

TInt CheckAge(TInt aAge)
{
    if (aAge < 0)
        RaisePanic(EValueTooLow);
}
```

```

else if (aAge > KMaxAge)
    RaisePanic(EValueTooBig);
return aAge;
}

```

В этом примере для задания категории паники объявляется символьный дескриптор (с помощью макроса `_LIT`). Различные виды дескрипторов будут обсуждаться нами позднее. Обратите внимание, что название функции `CheckAge()` не содержит суффикса “L”. Так как панику невозможно перехватить и любая функция потенциально способна к ней привести, то нет смысла сигнализировать об этом. В большинстве случаев в справочнике SDK, помимо описания назначения и аргументов различных методов, описываются также условия, при которых их использование способно привести к панике.

За предоставление пользователю информации о том, что процесс был аварийно завершен в результате возникшей паники, отвечает система. Зачастую в нормальном режиме работы Symbian OS отображает сообщение о том, что некоторое приложение закрыто в связи с ошибкой. При этом о категории паники и коде ошибки не сообщается. В некоторых случаях сообщение о панике может вообще не появляться. Такое поведение позволяет скрыть от глаз рядового пользователя ненужную ему информацию. В готовом приложении паника вообще никогда не должна возникать.

Для того чтобы увидеть дополнительную информацию о произошедшем сбое, на время тестирования приложения разработчик должен включить соответствующий режим в системе. В Symbian 9.x такой режим включается с помощью создания пустого файла с именем `errrd` (без расширения) в папке `C:\Resource\`. После чего сообщения об ошибках примут вид, показанный на рис. 5.1. Здесь **Main** — имя приложения, **Panic in my APP! -101** — категория паники, **-101** — код ошибки.

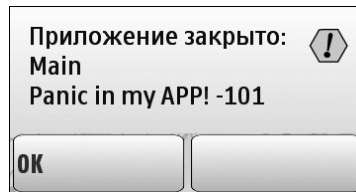


Рис. 5.1. Подробное сообщение о панике

Создать файл `errrd` можно с помощью стороннего файлового менеджера (встроенный не позволяет попасть в требуемый каталог) либо установив содержащий его SIS-файл. Второй способ мне кажется более предпочтительным. Добавляющий в систему файл `errrd` SIS-пакет можно найти в Википедии портала Forum Nokia, на Symbian Developer Network и в файловом архиве сообщества DevMobile. Его также легко подготовить самостоятельно.

При возникновении паники в эмуляторе аварийно завершается работа не только приведшего к ней потока, но и его самого. Это довольно неудобно. Для того чтобы эмулятор не закрывался и позволил рассмотреть системное

сообщение с информацией о панике, необходимо сбросить флажок опции **Just in time** в окне его параметров настройки, открываемом при выборе команды меню **Tools⇒Preferences⇒C++ Debug**. Того же можно добиться масштабах отдельно взятой программы, поместив в самом начале ее выполняемого кода следующий текст.

```
#ifdef __WINS__
    User::SetJustInTime(EFalse);
#endif
```

Макросы `__ASSERT_XXX` (утверждения)

Утверждениями (assertions) в Symbian C++ называют ряд макросов, использующихся для остановки работы программы в случае невыполнения определенного условия. Они принимают два аргумента: выражение, возвращающее числовое значение, и функцию, которая вызывается в том случае, если значение равно `EFalse` (или нулю). На практике чаще всего применяются макросы `__ASSERT_DEBUG` (работает только в режиме компиляции `UDEB`, в режиме `UREL` он исключается на стадии препроцессинга исходного кода) и `__ASSERT_ALWAYS` (выполняется всегда). Пример.

```
TInt CheckAge(TInt aAge)
{
    __ASSERT_DEBUG(aAge > 0, RaisePanic(EValueTooLow));
    __ASSERT_ALWAYS(aAge <= KMaxAge, RaisePanic(EValueTooBig));
    return aAge;
}
```

Не следует проверять в макросе `__ASSERT_DEBUG` код, выполняющий какие-либо необходимые для работы приложения операции. Не забывайте, что в `UREL`-сборках он вызываться не будет. Поэтому нужно выполнять такой код отдельно, получать его результат, а затем проверять этот результат в `__ASSERT_DEBUG`.

Технически утверждения не накладывают ограничений на тип вызываемой функции, но предполагается, что она должна останавливать ход выполнения приложения. В большинстве случаев для этих целей используются функции, вызывающие панику.

Подготовка к сертификации ASD

- Знание того, что версии Symbian OS до 9.x не поддерживают стандартные исключения C++ (`try/catch/throw`).
- Понимание того, что с Symbian 9.1 метод `User::Leave()` реализуется на основе исключений C++. При вызове `User::Leave()` выбрасывается исключение типа `XLeaveException` (простой `TInt`).
- Понимание того, что вложенные (nested) исключения на целевых сборках не поддерживаются, так как это требует динамического выделения памяти,

которое не может гарантироваться. Поэтому во время выполнения кода на устройстве выбрасывание исключения в момент обработки потоком другого исключения приводит к его аварийному завершению с помощью функции `abort()`.

- Знание того, что сбросы — фундаментальная часть механизма обработки ошибок Symbian OS и широко используются в системе.
- Понимание схожести сбросов и ловушек с выражениями `setjmp` и `longjmp` в C.
- Понимание того, что сбросы используются для уведомления об ошибках, что в большинстве случаев метод либо возвращает код ошибки, либо вызывает сброс с этим кодом. Крайне редко используется оба способа.
- Умение определять типичные системные функции, способные вызвать сброс (включая `User::LeaveXXX()` и `new(ELeave)`).
- Знание того, что функции, способные привести к сбросу, имеют суффикс "L" в названии (например, `AppendL()`).
- Умение отличить способную вызвать сброс функцию от безопасной по ее имени.
- Понимание разницы между сбросом и паникой.
- Понимание, что сбросы не должны использоваться для управления ходом выполнения процедур и функций.
- Знание характеристик различных макросов-ловушек.
- Понимание того, что использование ловушек должно быть сведено к минимуму и почему это так.
- Знание типов параметров, передаваемых функции `User::Panic()`, и методов их использования.
- Знание того, что паника может быть вызвана с помощью макросов-утверждений, используемых для обнаружения ошибок.
- Понимание того, как определять ошибки программирования в отладочном коде с помощью выражения `__ASSERT_DEBUG`.
- Сознание того, что макрос `__ASSERT_ALWAYS` должен использоваться с осторожностью, так как способен привести к панике не только в отладочном, но и в обычном коде.

Управление памятью: куча и стек

При запуске программы в памяти Symbian OS создается новый процесс, а в нем, в свою очередь, исполняемый **поток** (thread), считающийся *главным*. После этого само приложение может по мере необходимости создавать дополнительные потоки в рамках своего процесса. По завершению выполнения главного потока работа всего процесса прекращается. В подавляющем большинстве случаев вся управляющая логика программы содержится в одном потоке. Symbian C++ предлагает удобную концепцию активных объектов для реализации асинхронных вызовов, позволяющую максимально снизить необходимость

использования дополнительных потоков, так как переключение между ними снижает производительность.

Каждый процесс имеет собственное адресное пространство. Прямой доступ одного процесса к памяти другого процесса осуществить невозможно. Для обмена данными между ними следует воспользоваться одним из способов IPC-взаимодействия (например, клиент-серверной архитектурой или очередью сообщений).

Для хранения данных в потоке используется два типа памяти: **стек** (stack) и **куча** (heap). Память кучи доступна всем потокам процесса (при желании потоку можно назначить собственную кучу) и может динамически менять свой размер по мере необходимости. Ее минимальный и максимальный размеры задаются с помощью ключевого выражения `EPSCHEAPSIZE` в ММР-файле проекта. По умолчанию это 4 Кбайт и 1 Мбайт. Используемая кучей память выделяется постранично, поэтому размер кучи всегда кратен размеру страницы (4 Кбайт). Данные в куче хранятся в ячейках (cells). Symbian OS поддерживает адресацию с точностью не менее 32-х бит, поэтому размер ячейки всегда кратен машинному слову. Минимальный размер ячейки составляет 12 байт (каждая ячейка содержит небольшой заголовок). В куче хранятся относительно большие либо динамически меняющие свой размер объекты. Разработчик оперирует указателями на них. Для создания объекта в памяти кучи используется оператор C++ `new`. Оператор `new` выделяет в куче объем памяти, необходимый для размещения объекта указанного типа, создает этот объект и возвращает указатель на него. Следующий пример демонстрирует создание объекта типа `TInt` в куче.

```
TInt* obj = new TInt;
```

Переменная `obj` хранит указатель на созданный в куче объект. В том случае, если создать необходимый объект не удалось (например, в куче не хватило памяти), переменная `obj` примет значение `NULL`. Разработчик всегда должен проверять значение указателя на выделенный в куче объект перед первым обращением к нему. В Symbian C++ (файл `e32std.inl`) определен вариант оператора, вызывающий сброс при возникновении ошибки. Пример.

```
TInt* obj = new (ELeave) TInt;
```

По сути, он эквивалентен следующему выражению.

```
TInt* obj = (TInt*) User::LeaveIfNull(new TInt);
```

Использование вызывающего сброс оператора `new` очень удобно, так как гарантирует, что указатель не будет содержать `NULL`.

После того как работа с объектом будет завершена, он обязательно должен быть уничтожен с помощью оператора `delete`.

```
delete obj;
```

Оператор `delete` не вызывает ошибку, будучи вызванным для указателя, хранящего `NULL`. Поэтому при его использовании выполнение дополнительных проверок не требуется. В то же время оператор `delete` не помещает в указатель

значение NULL самостоятельно, и если после его применения вы еще раз попытаетесь по непустому указателю уничтожить объект, который уже был удален, то получите панику USER 44.

Для выделения в куче и уничтожения массивов используются операторы `new[]` и `delete[]`. В Symbian C++ они применяются крайне редко, так как работа с массивами ведется с помощью специальных R- и C-классов. Пример.

```
TInt* arr = new (ELeave) TInt[10];
<...> // Работа с arr
delete [] arr;
```

Если вы забудете уничтожить выделенный в куче объект или потеряете последний указатель на него, то зарезервированная на его хранение память будет недоступна для дальнейшего использования, и он так и останется в куче до конца работы программы. В случае если такая утечка памяти произойдет в цикле или часто вызываемом методе, то куча вскоре заполнится и, вполне возможно, недостаток памяти начнет испытывать вся система в целом. Поэтому политика Symbian OS в отношении утечек памяти довольно радикальна — их не должно быть никогда, нигде и ни при каких обстоятельствах. Если после завершения работы вашего приложения система обнаружит в куче оставшиеся не освобожденными ячейки, то вызовет панику категории ALLOC. Возможно, это и не скажется на функциональности программы, но испортит ее товарный вид, и сертификацию она уже не пройдет. Поэтому борьбу с утечками памяти следует начинать уже на стадии разработки.

Для обнаружения подобных проблем существуют макросы `__UNEAP_MARK`, `__UNEAP_MARKEND` и `__UNEAP_MARKENDC`. Эти макросы работают только в DEBUG-сборках и используются парами. Макрос `__UNEAP_MARK` сигнализирует о начале отслеживания состояния памяти кучи. Макрос `__UNEAP_MARKEND` это слежение оканчивает и вызывает панику, в случае если в куче остались не освобожденные ячейки памяти. Макрос `__UNEAP_MARKENDC` отличен от него лишь в том, что позволяет указать количество ячеек, которые оказались не освобожденными. Чаще всего, пару макросов `__UNEAP_MARK` и `__UNEAP_MARKEND` можно встретить недалеко от точки входа в коде довольно простых (часто консольных) программ.

Рассматривая вышеописанные примеры создания объекта `TInt` в куче, вы, вероятно, уже обратили свое внимание на то, что это довольно расточительно. Для хранения значения `TInt` достаточно 4-х байт, в то время как минимальная ячейка займет в три раза больше. К тому же вам приходится брать на себя ответственность за освобождение ячейки. Естественно, для хранения объектов фундаментальных типов данных больше подходит *стек*.

Стек является областью памяти сравнительно небольшого фиксированного размера (до 80 Кбайт). Каждый поток имеет собственный стек. Размер стека главного потока задается с помощью ключевого выражения `EPROCESSSTACKSIZE` в MMP-файле проекта и по умолчанию равен 8 Кбайт (0x2000). Следует отметить, что для GUI-приложений необходим стек размером 20 Кбайт (0x5000).

Так как размер стека фиксирован, в нем нет необходимости использовать строны и ячейки, поэтому помещаемые в него данные занимают ровно столько места, сколько должны. Освобождение выделенной объектам памяти в стеке происходит автоматически, как только объект выходит за пределы видимости.

Размер стека сравнительно невелик, и использовать его следует аккуратно. Если свободной памяти в стеке не останется, в потоке будет вызвана паника. В большинстве случаев стек хранит небольшие объекты, чаще всего — Т-классы, реже — R-классы. Но даже среди Т-классов есть представители довольно большого размера (например, некоторые дескрипторы). Поэтому об этом ограничении забывать не следует.

Подготовка к сертификации ASD

- Знание того, что оператор `new (ELeave)` способен вызвать сброс.
 - Способность перечислить типичные условия, при которых может возникнуть сброс (например, недостаточно свободной памяти для создания объекта в куче).
 - Понимание того, что оператор `new (ELeave)` всегда возвращает ненулевой работоспособный указатель либо вызывает сброс.
 - Знание того, что при использовании оператора `delete` проверка указателя на `NULL` может не производиться.
 - Умение обнаруживать утечки памяти с помощью макросов `__UHEAP_MARK` и `__UHEAP_MARKEND`.
-

Стек очистки: CleanupStack

В предыдущем разделе мы выяснили, что никаких сборщиков мусора в Symbian C++ нет, и за любую утечку памяти система будет нас беспощадно наказывать. В то же время, как вы помните, при возникновении сброса выполнение программы немедленно прерывается и продолжается после сработавшей ловушки. Таким образом, мы теряем возможность освободить память и ресурсы, выделенные непосредственно перед сбросом. Тривиальный пример.

```
void FuncL()
{
    TInt* i = new(ELeave) TInt;
    SomeLeavingFuncL();
    <...>
    delete i;
}
```

Если в этом примере в функции `SomeLeavingFuncL()` произойдет сброс, то выполнение `FuncL()` также прекратится, и в куче останется не освобожденная ячейка памяти, содержащая объект `TInt`.

Для борьбы с утечками памяти кучи при возникновении сбросов используется так называемый “**стек очистки**” (clean up stack). В данном случае слово *стек* не имеет отношения к памяти, а указывает на способ организации данных. Стек очистки предназначен для хранения указателей на объекты и ресурсы, которые должны быть освобождены во время сброса, и является неотъемлемой частью механизма обработки исключений Symbian C++.

Перед тем как начать работу со стеком очистки, вы должны его создать с помощью класса CTrapCleanup. Стек очистки функционирует в рамках одного потока. В GUI-приложениях и серверах ответственность за его инициализацию в главном потоке берет на себя система, и самостоятельно создавать экземпляр CTrapCleanup не следует. Но в консольных программах и при запуске дополнительных потоков вам придется создавать его вручную. Стек очистки необходим для работы макросов-ловушек TRAP, поэтому он должен быть инициализирован раньше их первого использования.

Сам по себе класс CTrapCleanup не несет никакой функциональности — его экземпляр можно только либо создать, либо удалить. Отметим также, что создавать несколько экземпляров стека очистки в одном потоке не следует — использоваться будет только один из них (последний). Непосредственная работа со стеком очистки ведется с помощью статического класса CleanupStack. Такое разделение очень удобно — не требуется иметь постоянный доступ к указателю на объект стека. Если вы обратитесь к методам CleanupStack в тот момент, когда стек очистки в текущем потоке еще не создан, возникнет паника E32USER-CBase 69.

Как вы помните, работа программы при сбросе продолжается со следующего оператора за ближайшей по стеку вызовов ловушкой. Поэтому система должна уничтожить только те отмеченные в стеке очистки объекты, которые были помещены в него в рамках этой ловушки, ведь с остальными вам еще предстоит работать. Так как в потоке используется один стек очистки, то каждая ловушка при вызове делает в нем специальную отметку-ограничитель. Во время сброса из стека начинают последовательно извлекаться указатели, а объекты, на которые они ссылаются, — уничтожаться. Этот процесс продолжается до тех пор, пока в вершине стека не окажется метка ближайшей ловушки. Таким образом, освобождаются только та память и те ресурсы, контроль над которыми действительно утрачен. Прежде чем в стек очистки будет помещен первый указатель, в нем должна появиться такая отметка — в противном случае будет вызвана паника E32USER-CBase 66. Отсюда вытекает правило: работать с классом CleanupStack можно только в функциях, защищенных ловушками. Опять же, GUI-приложения и серверы являются исключениями, так как там такую отметку ставит система. В дальнейшем мы для краткости будем предполагать, что наши примеры вызываются в рамках ловушки.

Класс CleanupStack имеет ряд перегруженных методов PushL () для добавления указателей на объекты, которые необходимо уничтожить при возникновении сброса, а также метод Pop () для их извлечения из стека. Таким образом, предотвратить утечку памяти в нашем примере мы можем следующим образом.

```
void FuncL()
{
    TInt* i = new (ELeave) TInt;
    CleanupStack::PushL(i);
    SomeLeavingFuncL();
    <...>
    CleanupStack::Pop();
    delete i;
}
```

В данном случае для помещения объекта в стек был использован перегруженный вариант метода `PushL()`, принимающий указатель `TAny*`. Если в функции `SomeLeavingFuncL()` произойдет сброс, то память, занимаемая таким объектом, будет освобождена с помощью метода `Free()` статического класса `User`. Работа метода `User::Free()` заключается в простом освобождении переданной ему по указателю ячейки памяти кучи, деструктор объекта при этом вызван не будет. Класс `CleanupStack` имеет еще два перегруженных метода `PushL()` — один для указателя на C-классы и другой для объекта типа `TCleanupItem`. При помещении в стек C-класса используется вариант метода `PushL()`, принимающий указатель `CBase*`. В этом случае при сбросе объект будет удаляться с помощью оператора `delete`, а значит, будет вызван и его деструктор.

Класс `TCleanupItem` инкапсулирует указатель на произвольный освобождаемый объект и функцию, которая должна его освободить. Это очень гибкий инструмент, позволяющий удалить с помощью стека очистки что угодно и как угодно. Но на практике необходимость в нем возникает крайне редко. Тем не менее вот тривиальный пример использования класса `TCleanupItem` для добавления объекта в стек очистки.

```
void CleanupOperation(TAny* aHeapInt)
{
    delete aHeapInt;
}

void FuncL()
{
    TInt* i = new (ELeave) TInt;
    // Конструктор TCleanupItem принимает указатели
    // на функцию удаления и удаляемый объект
    CleanupStack::PushL(TCleanupItem(CleanupOperation, i));
    SomeLeavingFuncL();
    <...>
    CleanupStack::Pop();
    delete i;
}
```

Созданные в куче массивы C++ фиксированной длины должны удаляться с помощью оператора `delete[]`. Для того чтобы поместить их в стек очистки,

необходимо использовать функцию `CleanupArrayDeletePushL()`. Она автоматически создает объект `TCleanupItem` и добавляет его в стек с указателем на массив и статическую функцию `CleanupArrayDelete::ArrayDelete()`, вызывающую оператор `delete[]` для своего аргумента. Таким образом, в случае возникновения сброса массив будет удаляться правильно. Пример.

```
void FuncL()
{
    // Создание массива из 100 элементов TInt
    TInt* i = new (ELeave) TInt[100];
    // Добавление в стек очистки
    CleanupArrayDeletePushL(i);
    SomeLeavingFuncL();
    // Выталкивание из стека и удаление
    CleanupStack::Pop();
    delete[] i;
}
```

Обратите внимание на то, что в названиях метода `PushL()` и функции `CleanupArrayDeletePushL()` содержится суффикс “L”. Это значит, что они могут вызвать сброс, и справочник SDK даже подскажет нам его наиболее вероятную причину — нехватка памяти для добавления элемента. Однако если сброс произойдет при добавлении указателя на объект, то он уже не будет автоматически удален — это приведет к утечке памяти? Нет. На самом деле при добавлении элемента в стек производится проверка на наличие памяти для следующего указателя. Именно эта проверка и способна вызвать сброс. Таким образом, во время сброса в методе `PushL()` ваш объект уже будет отмечен в стеке. Исключение составляет случай, когда для первого же добавляемого в стек указателя не хватило памяти, но это нонсенс — при таких условиях программа просто не запустится.

В том случае если функция `SomeLeavingFuncL()` завершилась успешно и не привела к сбросу, нам необходимо вытолкнуть объект из стека очистки. Для этого служит метод `Pop()`. Не уничтожайте объект, пока он не извлечен. Если вы выталкиваете объект из стека только за тем, чтобы удалить, то вы можете воспользоваться методом `PopAndDestroy()` — одновременно извлекающим объект из стека и уничтожающим его. Пример.

```
void FuncL()
{
    TInt* i = new (ELeave) TInt;
    CleanupStack::PushL(i);
    SomeLeavingFuncL();
    <...>
    // i извлечен и удален с помощью User::Free(i)
    CleanupStack::PopAndDestroy();
}
```

Рассматриваемые нами примеры довольно просты. На практике в стеке может одновременно содержаться множество указателей. В связи с этим возникают две проблемы — как извлечь несколько элементов одновременно и как избежать ошибок, связанных с очередностью объектов в стеке. Первая задача решается довольно просто: нет необходимости несколько раз подряд вызывать метод `Pop()` или `PopAndDestroy()`. Класс `CleanupStack` содержит варианты этих методов, принимающие количество объектов для извлечения как аргумент. Пример.

```
void FuncL()
{
    TInt* x = new (ELeave) TInt;
    CleanupStack::PushL(x);
    TInt* y = new (ELeave) TInt;
    CleanupStack::PushL(y);
    SomeLeavingFuncL();
    <...>
    CleanupStack::PopAndDestroy(2); //y и x извлечены и удалены
}
```

Если вы попытаете извлечь из стека очистки больше элементов, чем в нем находится, — получите панику.

Вторая проблема сложнее. Иногда разработчик забывает поместить указатель на объект в стек либо не уверен в количестве объектов, которые необходимо извлечь из него. Все методы `Pop()` и `PopAndDestroy()` имеют перегруженный вариант, в котором последним аргументом является указатель на объект. В случае если последний извлекаемый из стека указатель с ним не совпадет — будет вызвана паника `E32USER-CBase 90`. Такая проверка выполняется только при сборке в `DEBUG`-режиме. Пример.

```
void FuncL()
{
    TInt* x = new (ELeave) TInt;
    CleanupStack::PushL(x);
    TInt* y = new (ELeave) TInt;
    CleanupStack::PushL(y);
    SomeLeavingFuncL();
    <...>
    // Убедимся, что последним извлекается именно x
    CleanupStack::PopAndDestroy(2, x);
}
```

Класс `CleanupStack` также содержит отдельный метод для проверки указателя, находящегося на вершине стека — `Check()`. Он работает и в `DEBUG`-, и в `RELEASE`-сборках и в случае несовпадения вызывает такую же панику. Пример.

```
void FuncL()
{
    TInt* x = new (ELeave) TInt;
```

```

CleanupStack::PushL(x);
TInt* y = new (ELeave) TInt;
CleanupStack::PushL(y);
SomeLeavingFuncL();
<...>
// Убедимся, что на вершине стека y
CleanupStack::Check(y);
CleanupStack::PopAndDestroy(2);
}

```

К сожалению, разработчик не всегда имеет под рукой указатель для такой проверки.

При проектировании приложения вам может потребоваться выделить код создания и инициализации объекта в куче в отдельный метод. Такие методы возвращают ссылку на объект в качестве результата. Зачастую нет смысла извлекать ее из стека очистки, так как работа с объектом будет продолжена за пределами метода, и, не исключено, что его тут же придется помещать обратно в стек. В том случае, если в результате выполнения метода в стеке очистки появляются новые элементы, его название должно содержать суффикс “C”. Пример.

```

TInt* FuncLC()
{
    TInt* x = new (ELeave) TInt;
    CleanupStack::PushL(x);
    SomeLeavingFuncL();
    <...>
    return x;
}

void DoStartL()
{
    // Создаем объект, он остается в стеке очистки
    TInt * obj = FuncLC();
    <...> // Работа с obj
    CleanupStack::PopAndDestroy(obj);
}

```

Из механизма расставляемых ловушками отметок-ограничителей вытекает еще одно правило: все, что было помещено в стек очистки в рамках ловушки, должно быть извлечено до выхода из нее. Таким образом, мы можем заключить в ловушку функцию DoStartL(), но никак не FuncLC(). Если возникает необходимость получить созданный в куче объект за пределами ловушки, то его стоит предварительно вытолкнуть из стека очистки, а затем, если необходимо, поместить в него снова. Пример.

```

TInt* FuncL()
{
    TInt* x = new (ELeave) TInt;
    CleanupStack::PushL(x);

```

```
SomeLeavingFuncL();
<...>
CleanupStack::Pop(x);
return x;
}

void DoStartL()
{
    TInt* obj(NULL);
    TRAPD(err, obj = FuncL());
    if (KErrNone == err)
    {
        CleanupStack::PushL(obj);
        <...> // Работа с obj
        CleanupStack::PopAndDestroy(obj);
    }
}
```

В то же время, отмеченный в стеке очистки объект может использоваться во вложенных ловушках без извлечения.

```
void FuncL()
{
    TInt* x = new (ELeave) TInt;
    CleanupStack::PushL(x);
    TRAP_IGNORE(SomeLeavingFuncL(x));
    CleanupStack::PopAndDestroy(x);
}
```

Члены классов никогда не помещаются в стек очистки — достаточно освободить их в деструкторе класса и поместить в него сам класс.

Конструктор класса никогда не должен приводить к сбросу. Он вызывается сразу после размещения объекта в памяти, и, в случае возникновения исключения, эта память будет потеряна, так как указатель на объект еще не помещен в стек очистки. Даже если вы поместите объект в стек очистки в самом конструкторе, в случае сброса это может привести к панике.

Также запрещается использовать методы, способные привести к сбросу в деструкторе (это может привести к вложенным исключениям, а они не поддерживаются). Более подробно о причинах этих ограничений вы можете узнать из статей Джо Стичбури (Jo Stichbury) “Can I Leave in Constructor?” и “Can I Leave in a Destructor?”, упомянутых в конце этой книги.

Подготовка к сертификации ASD

- Понимание того, почему сбросы не должны происходить в конструкторах и деструкторах.
- Знание того, как использовать стек очистки, чтобы сделать код безопасным и избежать утечки ресурсов во время сброса.

- Понимание того, что метод `CleanupStack::PushL()` не приводит к утечке памяти, даже когда сброс происходит в нем самом.
- Знание того, в каком порядке извлекаются объекты из стека и как используются методы `CleanupStack::Pop()` и `CleanupStack::PopAndDestroy()`.
- Умение определять правильное и неправильное использование стека очистки.
- Знание того, как используется метод `CleanupStack::PushL()` и класс `TCleanupItem` для различных объектов, а также метод `CleanupArrayDeletePushL()` для массивов C++.
- Знание назначения суффикса "C" в именах функций.

Т-классы

К категории Т-классов в Symbian C++ относятся фундаментальные типы данных, перечисления, а также ряд классов и структур, ведущих себя подобно простым типам данных. Префикс "Т" в их названиях является сокращением от слова *Type*. Основным назначением этой категории классов является хранение данных. Экземпляры Т-классов могут создаваться как в куче, так и в стеке (большие объекты стараются размещать в куче, чтобы экономить память стека). Они не должны владеть какими-либо данными или ресурсами, требующими дополнительного освобождения. Поэтому членами таких классов могут быть только другие Т-классы (включая фундаментальные типы), а также ссылки и указатели на объекты. При этом класс не должен отвечать за освобождение объектов, на которые ссылаются его члены. Поэтому, для того чтобы уничтожить экземпляр Т-класса, достаточно лишь освободить занимаемую им память. По этой причине классы, относящиеся к категории Т, не должны иметь собственный деструктор.



Технически Т-классы могут иметь деструктор. В Symbian OS 9-й версии он будет вызываться, когда ваш объект выходит за пределы видимости или удаляется с помощью оператора `delete`. Но вот стек очистки вызывать деструктор вашего класса не будет. Поэтому вам придется помещать его туда с помощью `TCleanupItem`. Это не очень удобно, к тому же если такой объект будет использоваться в качестве члена другого Т-класса, то и он будет вынужден объявить деструктор. В более ранних версиях ОС деструктор стековых объектов при их выходе за пределы видимости вообще не вызывается. Поэтому использование собственного деструктора в Т-классах крайне нежелательно.

Членам Т-классов при создании присваиваются случайные значения (кроме случая, когда Т-класс является членом С-класса и при создании его данные обнуляются), поэтому часто для их инициализации используется конструктор. Они также могут инициализироваться с помощью копирующего оператора.

Подобно простым типам данных, T-классы могут передаваться в функции и по значению, и по ссылке (предпочтительнее для объектов большого размера, например, TLex). Иногда, T-классы содержат методы для работы с хранящимися в них данными (чаще всего Set- и Get-методы). Пример.

```
class TMyAge
{
public:
    TMyAge();
    void SetAgeL(const TReal aAge);
    inline TInt Age() const;
    TMyAge* iNext;
private:
    TInt iAge;
};

// Set-метод, способный вызвать сброс
void TMyAge::SetAgeL(const TReal aAge)
{
    if (aAge < 0)
        User::Leave(KErrArgument);
    iAge = aAge;
}

// Get-метод, префикс Get опущен
inline TInt TMyAge::Age() const
{
    return iAge;
}

// Инициализируем данные в конструкторе
TMyAge::TMyAge(): iAge(0), iNext(NULL)
{
}
```

Подготовка к сертификации ASD

- Знание назначения T-классов, а также, какие члены они могут содержать и какие не должны содержать. Знание того, что T-класс не должен иметь деструктор.
 - Знание, какого типа функции могут содержаться в T-классе.
 - Понимание, что объекты T-класса могут создаваться как в стеке, так и в куче.
 - Понимание, что T-классы могут использоваться вместо традиционных структур C/C++.
 - Знание того, что префикс "T" также используется при определении перечислений и структур.
-

С-классы, двухфазное конструирование

С-классы — одни из наиболее распространенных в Symbian C++. Они предназначены для хранения и управления различными видами данных (в том числе, ресурсами, требующими освобождения) и реализуют сложную логику. Литера “С” в их названии происходит от слова `class`. Все С-классы являются наследниками класса `CBase` (объявленного в файле `e32base.h`), что наделяет их следующими характеристиками.

- Класс `CBase` содержит переопределенные операторы `new` и `new (ELeave)`, предварительно обнуляющие выделяемую под объект память кучи. Поэтому все члены С-класса инициализируются нулями.
- Все С-классы должны создаваться только в памяти кучи. Это гарантирует, что члены класса будут по умолчанию инициализированы нулями.
- Класс `CBase` содержит виртуальный деструктор. Как вы помните, стек очистки имеет перегруженный метод `PushL (CBase*)`. Использование виртуального деструктора гарантирует, что при удалении объекта через указатель на его родительский класс будут последовательно вызваны все необходимые деструкторы в иерархии наследования. Таким образом, вы можете безбоязненно помещать любой С-класс в стек очистки — его деструктор обязательно будет вызван в случае сброса.
- В классе `CBase` копирующий конструктор и оператор присваивания находятся в разделе `private`. Поэтому в большинстве случаев С-классы не поддерживают эти возможности и передаются по ссылке либо с помощью указателя. Если же копирующий конструктор или оператор присваивания действительно необходим, его определяют заново.

Вот небольшой пример С-класса, резервирующего область памяти заданного размера.

```
class CClassName : public CBase
{
    ~CClassName();
    CClassName(TInt aSize);

    TInt iSize;
    TAny* iMemory;
};

CClassName::CClassName(TInt aSize): iSize(aSize)
{
    iMemory = User::AllocL(iSize);
}

CClassName::~CClassName()
{
    delete iMemory;
}
```

Обратите внимание, что после освобождения в деструкторе зарезервированной памяти с помощью оператора `delete`, указателю не присваивается `NULL`. Так как деструктор объекта может быть вызван лишь раз, то обращения к нему больше не будет.

Приведенный выше пример также содержит потенциальную опасность — вызов метода, способного привести к сбросу в конструкторе. На практике C-классы очень часто при инициализации обращаются к небезопасным функциям. Мы должны вынести эти функции из конструктора в отдельный метод, который будет вызван нами сразу же после создания экземпляра класса и помещения его в стек очистки. Такой метод C-класса часто называют `ConstructL()`. Пример.

```
CClassName::CClassName(TInt aSize): iSize(aSize)
{
}

void CClassName::ConstructL()
{
    iMemory = User::AllocL(iSize);
}
```

В этом случае безопасное создание объекта класса `CClassName` будет выглядеть следующим образом.

```
CClassName* obj = new (ELeave) CClassName(1024);
CleanupStack::PushL(obj);
obj->ConstructL();
<...> // Работаем с obj
CleanupStack::PopAndDestroy(obj);
```

Вышеописанный прием называют **двухфазным конструированием** (two-phase construction). Первой фазой являются выделение памяти под объект и вызов конструктора класса, второй — вызов метода `ConstructL()`. Существует опасность, что разработчик забудет вызвать вторую фазу конструирования и попытается воспользоваться не полностью проинициализированным объектом. Чтобы этого не произошло, конструктор и метод `ConstructL()` объявляются приватными, а в класс добавляется новый статический метод `NewL()`, выполняющий обе фазы конструирования и возвращающий ссылку на созданный и инициализированный объект.

```
CClassName* CClassName::NewL(TInt aSize)
{
    CClassName* self = new (ELeave) CClassName(aSize);
    CleanupStack::PushL(self);
    self->ConstructL();
    CleanupStack::Pop(); // self;
    return self;
}
```

Аналогично создается и LC-вариант этого метода. На практике в C-классе объявляются оба метода, так как это довольно удобно.

```
CClassName* CClassName::NewLC(TInt aSize)
{
    // I-я фаза конструирования
    CClassName* self = new (ELeave) CClassName(aSize);
    CleanupStack::PushL(self);
    // II-я фаза конструирования
    self->ConstructL();
    return self;
}

CClassName* CClassName::NewL(TInt aSize)
{
    CClassName* self = CClassName::NewLC(aSize);
    CleanupStack::Pop(); // self;
    return self;
}
```

Подавляющее большинство C-классов имеют двухфазный конструктор и хотя бы один из NewX-методов для его использования. Членами C-класса могут быть любые типы данных, в том числе и другие C-классы (так как объект создается в куче). Но, учитывая применяемое в них двухфазное конструирование, в качестве членов разумнее использовать указатели на C-классы. За рядом исключений можно сказать, что вся работа с C-классами ведется через указатели на них. Иногда большие T-классы также предпочитают не создавать в стеке, а объявлять членами C-класса. Таким образом, разработчики экономят память стека за счет кучи.



Очень удобно создавать C-классы с помощью мастера Carbide.c++. Для этого в панели Project Explorer откройте контекстное меню каталогов src или inc вашего проекта и выберите в нем команду New⇒Symbian OS C++ Class. В открывшемся окне мастера введите название класса (без префикса "C") и выберите в качестве шаблона вариант Simple Symbian Style C++ Class. IDE создаст для вас CPP- и H-файлы пустого C-класса, содержащего уже готовый двухфазный конструктор и оба NewX-метода.

Подготовка к сертификации ASD

- Понимание того, что предком любого C-класса является класс CBase.
- Знание назначения C-классов и какие данные они могут содержать.
- Понимание того, что C-классы всегда должны создаваться в куче.
- Знание того, что C-классы могут использовать двухфазное конструирование, а их члены инициализируются нулевыми значениями.
- Понимание возможности правильного удаления C-класса через виртуальный деструктор базового класса CBase.

- Понимание того, что при использовании оператора `delete` в деструкторе для указателей, являющихся членами класса, не нужно затем присваивать им значение `NULL`. Это бессмысленно и неэффективно.
 - Знание того, почему в конструкторе ни при каких обстоятельствах не должен происходить сброс.
 - Знание того, что двухфазное конструирование позволяет избежать создания объектов в неопределенном состоянии.
 - Понимание, почему конструктор и метод второй фазы конструирования `ConstructL()` объявляются в классе с модификаторами доступа `private` или `protected`.
 - Понимание того, что публичные фабричные статические методы `NewL()` и `NewLC()` предоставляются для создания экземпляра класса.
 - Понимание того, как реализуется двухфазное конструирование и как правильно использовать такой класс в качестве базового класса при наследовании.
 - Знание наиболее часто встречающихся C-классов Symbian OS, использующих двухфазное конструирование.
-

R-классы

Классы, входящие в категорию R (от слова *resource*), предназначены для выделения и управления ресурсами, находящимися вне прямого доступа разработчика (принадлежащие системе или другому процессу). Яркий пример — доступ к файлам через класс `RFile`. Работая с таким объектом, может показаться, что вы обращаетесь к файлу напрямую, хотя на самом деле вы лишь шлете запросы к файловому серверу системы, выполняющему все операции над файлами. Реже R-классы используются для сокрытия какой-то функциональности, предоставляя разработчику более простой и понятный интерфейс.

На практике довольно редко возникает необходимость создания собственного R-класса. Исключением является разработка клиентской части в приложениях клиент-серверной архитектуры. Чаще всего вы будете сталкиваться с R-классами, определенными в различных API Symbian OS.

R-классы не имеют общего предка, как C-классы. Большинство R-классов содержит **хендл** (*handle*) на объект, к которому они подключены. В качестве хендла может выступать указатель на объект, идентификатор ресурса либо что-то другое. В любом случае нулевое значение хендла свидетельствует о том, что никакого ресурса объекту R-класса не выделено.

R-классы создаются в стеке, реже — в памяти кучи (обычно как члены C-класса). Они содержат лишь примитивный конструктор, обнуляющий значение хендла при их создании. В R-классах крайне редко используется деструктор. Копирующий конструктор и оператор присваивания чаще всего объявлены приватными, поэтому объекты передаются по ссылке. Обычно R-классы не хранят какие-либо данные, кроме хендла, но могут иметь большое число разнообразных методов.

Незначительная роль конструктора и деструктора объясняется тем, что для инициализации и освобождения выделенного ресурса используются отдельные методы (к тому же инициализация зачастую небезопасна). Это позволяет подключаться к ресурсу лишь тогда, когда это действительно необходимо, и освобождать его как можно раньше. Сам объект R-класса после этого может использоваться повторно. Методы для подключения к ресурсам чаще всего носят название `Open()`, `Connect()`, `Create()` и т.д. Методы для освобождения ресурсов — `Close()`, `Destroy()`, `Free()` ...

Очень важно всегда закрывать выделенные R-классу ресурсы. В противном случае это может привести к утечкам памяти или блокировке объектов. Важно помнить, что деструктора подобного рода классы не имеют, а при выходе за пределы видимости методы освобождения ресурсов автоматически не вызываются.

Для того чтобы избежать потери контроля над выделенными объектами в случае сброса, R-классы также необходимо помещать в стек очистки. Так как деструктора у них нет, то для размещения объекта в стеке используется класс `TCleanupItem`. К счастью, разработчику не нужно каждый раз выполнять лишнюю работу: для наиболее часто встречающихся в R-классах освобождающих ресурсы методов в Symbian C++ определены функции-шаблоны. Их всего три: `CleanupDeletePushL()`, `CleanupClosePushL()` и `CleanupReleasePushL()`. Эти функции автоматически помещают в стек очистки объект `TCleanupItem` с указателями на экземпляр R-класса и метод `Delete()` (либо `Close()` и `Release()` соответственно), вызываемый в случае сброса. Пример.

```
class CFileReaderSample: public CBase
{
public:

    ~CFileReaderSample();
    static CFileReaderSample* NewL();
    static CFileReaderSample* NewLC();

private:

    CFileReaderSample();
    void ConstructL();
    void ReadFileL();
private:
    RFs iFs; // Сессия к файловому серверу
};

void CFileReaderSample::ConstructL()
{
    User::LeaveIfError(iFs.Connect());
}
```

```

CFileReaderSample::~CFileReaderSample()
{
    iFs.Close();
}

void CFileReaderSample::ReadFileL()
{
    RFile f;
    // Открытие файла
    User::LeaveIfError(f.Open(iFs, _L("c:\\myfile.dat"),
EFileWrite|EFileShareAny));
    CleanupClosePushL(f);
    // Методы работы с файлом, способные привести к сбросу
    <...>
    CleanupStack::PopAndDestroy(); // Закрытие файла
    // Обработка прочитанных данных
    // Мы вновь можем открыть файл с помощью f.Open()
}

```

В этом примере опущена реализация методов `NewL()` и `NewLC()`. Я бы мог убрать и их объявления, но хочу, чтобы вы привыкали к этим методам, как к обязательному атрибуту C-класса.

Заметьте, что мы использовали два объекта R-классов: `RFs` — сессия к файловому серверу и `RFile` — объект для работы с файлом через эту сессию. Объект `RFs` можно было бы объявить и использовать в рамках метода `ReadFileL()`, но на практике сессия с файловым сервером часто становится разделяемым ресурсом в рамках всего приложения и постоянно устанавливать и рвать соединение не рекомендуется. Более подробно об этом мы еще поговорим в *главе 6*, раздел “Работа с файловой системой”.

Обратите внимание на то, что я не забочусь о помещении объекта `iFs` в стек очистки. Во-первых, я просто не могу этого сделать: где бы ни был помещен вызов `CleanupClosePushL(iFs)`, мы не можем гарантировать, что по завершении работы этого метода нам не встретится ловушка. Во-вторых, в этом нет необходимости. Если с объектом `CFileReadSample` будут правильно обращаться и поместят в стек очистки, то в случае сброса обязательно будет вызван его деструктор, а значит, и сессия с файловым сервером будет разорвана.

Сразу после того, как файл открывается с помощью метода `Open()` класса `RFs`, объект помещается в стек очистки с помощью функции-шаблона `CleanupClosePushL()`. Таким образом, в случае возникновения сброса при чтении данных из файла, он будет закрыт методом `Close()`. По завершении чтения файла мы могли бы вытолкнуть его из стека очистки и закрыть вручную, но чтобы не делать лишнюю работу, совмещаем эти операции с помощью метода `PopAndDestroy()`. Сразу после этого объект класса `RFs` может быть использован вновь.

В работе Сорина Баска (Sorin Basca) “An Introduction to R Classes”, ссылка на которую имеется в конце книги, вы найдете описание и примеры использования

довольно интересного класса `TAutoClose`, автоматически вызывающего метод `Close()` у вышедшего за область видимости объекта R-класса. На практике он применяется довольно редко и подробно рассматривать его мы не будем.

Подготовка к сертификации ASD

- Знание назначения R-классов (владение ресурсом).
- Понимание того, что объект R-класса может быть создан как в стеке, так и в куче.
- Понимание того, что R-классы создаются и инициализируются (открытие ресурса) отдельно, а также необходимость такого разделения.
- Понимание того, что освобождение (уничтожение) принадлежащего объекту R-класса ресурса и самого R-класса осуществляется отдельно. Представление о негативных последствиях, когда объект R-класса уничтожается раньше, чем вызываются методы освобождения принадлежащего ему ресурса.
- Знание того, как используются методы `CleanupDeletePushL()`, `CleanupClosePushL()` и `CleanupReleasePushL()`.

М-классы, наследование

Как и C++, Symbian C++ поддерживает множественное наследование. Но пользоваться им следует очень осторожно, так как это источник потенциальных ошибок. В более молодых языках программирования (например, Java и C#) эти проблемы решаются с помощью интерфейсов. Symbian C++ имеет свой аналог интерфейсов — М-классы.

Своим названием М-классы обязаны слову *mixin* (примесь). Примеси столь же старая концепция, что и интерфейсы, но вы, возможно, услышите о них впервые. Примесь — это абстрактный класс, применяемый во множественном наследовании для уточнения поведения потомка. Как правило, все содержащиеся в нем методы являются чисто виртуальными. Однако класс-примесь может содержать реализацию виртуальных методов — в этом его отличие от интерфейса. Это допускается в том случае, если все потомки М-класса реализуют схожее поведение, и разработчик хочет избежать напрасного дублирования кода в каждом из них.

М-классы не должны содержать члены-данные, конструктор и перегруженные операторы. Примесь может иметь виртуальный деструктор. Экземпляры М-классов никогда не создаются (это возможно, если класс не абстрактный, но противоречит его назначению).

В Symbian C++ рекомендуется наследовать класс только от одного конкретного базового класса, в то же время, вы можете наследовать его от любого количества классов-примесей. Пример.

```
class CClassName : public CBase, public MSomeClass1,
                  public MSomeClass2
```

Обратите внимание, что примеси перечисляются после базового класса CBase. Это гарантирует, что при помещении объекта в стек очистки будет использоваться правильный перегруженный метод PushL (CBase*). Если М-классы окажутся первыми, то вызван будет PushL (TAny*), и при извлечении объекта из стека с помощью Pop (TAny*) на DEBUG-сборках вы можете получить панику. Более подробно эта проблема описана в статье Джо Стичбури (Jo Stichbury) “Mixin Inheritance and the Cleanup Stack” (ссылка в конце книги). При наследовании примеси всегда перечисляются после базовых классов. Порядок следования самих примесей при этом не важен.

Чаще всего М-классы используются для объявления **методов обратного вызова** (callback) в механизме получения уведомлений. Пример.

```
/*
 * Примесь, позволяющая зарегистрироваться
 * На получение уведомления о звонке будильника
 */

class MClock
{
public:
    virtual void WakeUpL(TAny* aParam);
};

// Класс, вызывающий событие

class CClock : public CActive
{
private:
    MClock* iListner;
public:
    CClock(MClock* aListner);
    TInt Ring(TAny* aParam);
};

CClock::CClock(MClock* aListner) : iListner(aListner)
{
}

TInt CClock::Ring(TAny* aParam)
{
    if (iListner)
    {
        TRAPD(err, iListner->WakeUpL(aParam));
        return err;
    }
    else
        return KErrNotFound;
}
```

// Класс, подписывающийся на уведомление о событии

```
class CListner : public CBase, public MClock
{
private:
    void WakeUpL(TAny* aParam);
};

void CListner::WakeUpL(TAny* aParam)
{
    // Произошло событие
}
```

Таким образом, передав указатель на экземпляр класса CListner в конструктор CClock, мы тем самым подписываемся на уведомление о произошедшем событии. При этом CClock может не знать о классе и свойствах объекта, полученных им в конструкторе. Главное для нас — что он реализует необходимый интерфейс. Подобный способ передачи уведомлений довольно часто встречается в системных API.

В общем случае указатель или ссылка на реализованную в объекте примесь не могут быть использованы для удаления всего объекта (а значит, не могут помещаться в стек очистки), так как содержащийся в них адрес не указывает на его начало. Метод `User::Free()` для такого адреса приведет к панике USER 42. Таким образом, мы не можем уничтожить объект класса CListner через `iListner` в CClock. Для решения этой проблемы в М-классе должен быть объявлен виртуальный деструктор. На практике М-класс не должен владеть реализующим его объектом, и если у вас возникает такая необходимость, то вам, возможно, стоит еще раз подумать над архитектурой приложения.



Если вы пишете программу для Symbian OS 6-8, то использование деструктора в М-классе предполагает, что реализующий примесь объект всегда будет храниться в куче и удаляться с помощью оператора `delete`. Гарантировать такое поведение могут только С-классы, поэтому виртуальный деструктор в М-классе ограничивает возможности его наследования.

Подготовка к сертификации ASD

- Знание назначения М-классов (определение интерфейсов).
 - Понимание правил использования М-классов при множественном наследовании, а также порядка указания С- и М-классов при этом.
 - Знание того, что М-класс не должен содержать члены-данные и конструкторы.
 - Знание, какие функции могут содержаться в М-классе, а также причины, по которым в него может включаться их реализация.
 - Понимание невозможности создания объектов М-класса.
-

Дескрипторы, работа со строками

Для работы со строковыми данными в Symbian C++ вместо null-завершенных строк используются ряд специальных классов — **дескрипторов** (descriptor). Дескриптор, или описатель, содержит как сами данные, так и служебную информацию о них (размер, тип и пр.). Так как в Symbian OS повышенное внимание уделяется надежности работы программ и контролю над ресурсами, то использование C-строк сочли слишком опасным: они часто приводят к утечкам памяти и могут скомпрометировать безопасность системы в целом (переполнение буфера). Поэтому каждый дескриптор “знает” свой размер и вызывает панику категории USER при попытке его переполнить. Из этого также следует, что дескриптору не нужен завершающий символ (например, null), а значит, он может хранить бинарные данные.

Казалось бы, все очень просто. Но помимо безопасности для системы крайне важна и ее производительность. Поэтому, несмотря на то, что дескрипторы проверяют свой размер при добавлении в них данных, они не способны его автоматически увеличивать — это слишком накладно. Таким образом, все дескрипторы в Symbian — фиксированного размера.

Свой отпечаток на работу со строками накладывает и ограниченность ресурсов устройств, на которых функционирует ОС. Размер памяти стека довольно мал, поэтому строки большого размера (более 256 байт) рекомендуется хранить в куче.

Все вышеперечисленное делает работу с дескрипторами настолько непохожей на операции со строками в языках C, C++, C# или Java, что их изучение является настоящим испытанием как для новичков, так и для опытных разработчиков. Чрезвычайно важно понять принципы работы с ними, всю сложность и в то же время “изящность” дескрипторов, — не научившись балансировать между надежностью, производительностью и управлением памятью, вы не сможете проектировать приложения.

Классы дескрипторов.

Изменяемые и неизменяемые дескрипторы

На рис. 5.2 представлена иерархия основных классов дескрипторов Symbian OS. Все они делятся на две группы: изменяемые и неизменяемые. Неизменяемый класс легко отличить по суффиксу “C” в его названии. Под “неизменяемостью” подобных дескрипторов понимается следующее: помещенная в него строка не предназначена для модификации. Тем не менее она в дальнейшем может быть заменена другой строкой. Таким образом, неизменяемый дескриптор является хранилищем для строк-констант, но сам константой не является. Нетрудно заметить, что на самом деле разработчик может редактировать “неизменяемые” данные. Для этого достаточно скопировать их в изменяемый дескриптор, модифицировать и поместить обратно в неизменяемый. Это действительно так, более того — неизменяемые дескрипторы содержат специальный метод для

выполнения подобного трюка. Так зачем же вообще нужны неизменяемые классы дескрипторов, если данные в них все равно можно изменить?

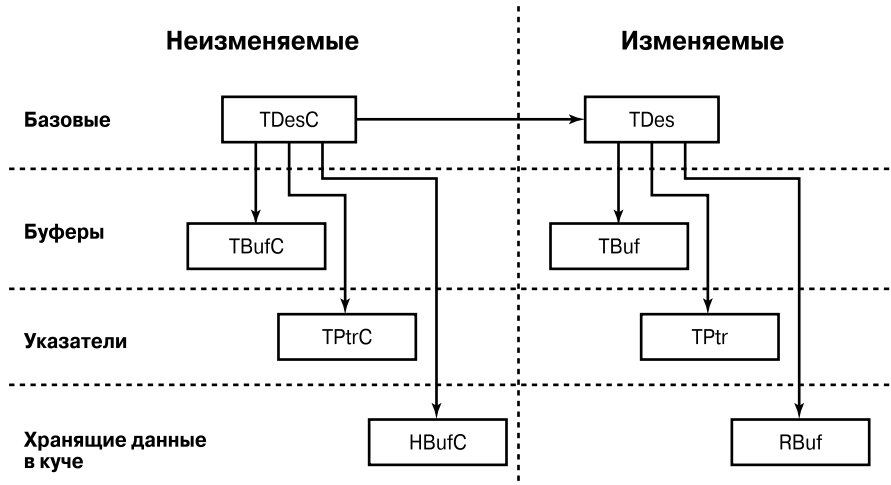


Рис. 5.2. Иерархия классов дескрипторов

На самом деле огромное количество строковых данных в системе и прикладных приложениях являются неизменяемыми: они инициализируются каким-либо значением, а затем используются на протяжении всей работы, не подвергаясь изменениям. Представьте себе: значительная часть текста интерфейса системы, разнообразные всплывающие подсказки, строки диалоговых сообщений — все это по сути константы, которые необходимо считать из внешних файлов ресурсов для соответствующего языка системы. Как мы уже говорили, все дескрипторы имеют ограниченный размер, поэтому они содержат информацию о своем текущем размере строки и максимальном размере, которого она может достичь. Но для константы эти два значения эквивалентны. Поэтому неизменяемый дескриптор имеет на одно информационное поле меньше и занимает в памяти на 4 байта меньше своего изменяемого аналога. Вот из-за этих 4-х байт в Symbian OS и появились неизменяемые дескрипторы. В те времена экономия 4-х байтов на каждом объекте-дескрипторе была очень весомым аргументом для выделения новой группы классов.

Подготовка к сертификации ASD

➤ Знание иерархии наследования дескрипторных классов.

Базовые дескрипторы

Все дескрипторы происходят от класса `TDesC`, при этом все изменяемые дескрипторы наследованы от его прямого потомка `TDes`. Оба этих класса называют базовыми или абстрактными — они не предназначены для непосредствен-

ного хранения данных, и вы не должны создавать объекты этих классов. Схема размещения данных дескриптора в памяти зависит от его типа (класса), но все они начинаются с 4-х байт, в которых 4 бита задают тип дескриптора, а оставшиеся 28 бит — максимальный размер хранимой строки. Таким образом, количество разнообразных типов дескрипторов ограничено $2^4 = 16$ (в действительности используются только 6 — они перечислены на рис. 5.2), а максимальный размер строки 2^{28} байт = 256 Мбайт.

В классе TDesC реализованы все базовые методы, наследуемые его потомками. Класс TDes добавляет реализацию методов, специфичных именно для изменяемых дескрипторов. Все они не виртуальные и не могут перегружаться наследниками, что позволяет избежать добавления указателя на таблицу виртуальных функций в каждый экземпляр дескриптора и тем самым сэкономить еще 4 байта памяти.

Базовый класс TDesC содержит метод Ptr(), возвращающий адрес начала строки в зависимости от типа дескриптора. Таким образом, базовый класс “знает” о том, как размещают данные все его потомки — эта информация в нем закодирована. Это значит, что разработчик не может определить собственный класс дескрипторов, порожденный от TDesC или TDes.

Подготовка к сертификации ASD

- Понимание невозможности создания объектов базовых классов TDesC и TDes.
 - Знание особенностей применяемой в дескрипторах модели наследования, связанных с ней преимуществ (экономия памяти) и ограничений (нельзя создать новый тип дескриптора).
 - Понимание того, что базовые классы TDesC и TDes содержат реализацию всех общих для дескрипторов функций. Тогда как унаследованные от них классы, в основном, добавляют специфичные функции конструирования и присваивания.
-

Символьные дескрипторы

Прежде чем перейти к обсуждению дескрипторов TBuf и TBufC, мне бы хотелось поговорить о том, как в Symbian C++ задаются настоящие строковые константы — иначе нам не построить подходящих примеров. Строковые константы объявляются с помощью макросов _L() и _LIT(). Пример.

```
_LIT(KMyName, "Aleksandr");
```

В данном случае макрос _LIT() позволяет создать так называемый символьный дескриптор (literal descriptor) с именем KMyName. На самом деле символьный дескриптор — это объект класса TLitC, эквивалентный массиву static char[] в C. Такие объекты являются настоящими константами и могут встраиваться в ROM (обратите внимание на префикс “K” в названии KMyName — все именованные символьные дескрипторы должны содержать такой). Строго говоря, TLitC вовсе не дескриптор, так как не наследован от TDesC, но его схе-

ма размещения элементов в памяти идентична дескриптору `TBufC`. Это позволяет использовать такую строку везде, где может применяться `TDesC`. Настоящий константный объект `TDesC` из символьного дескриптора можно получить с помощью оператора `()`. Макрос `_LIT` часто применяют для инициализации дескрипторов, а также задания констант.



Не нужно создавать символьный дескриптор для пустой строки — используйте объявленный в файле `e32cmn.h` дескриптор `_LIT(KNullDesC, "")`.

В отличие от `_LIT()`, макрос `_L()` не объявляет константу, а позволяет получить строку как значение. Казалось бы, это очень удобно — ведь не нужно каждый раз выдумывать уникальное имя, но здесь есть один нюанс: макрос `_L()` хранит данные в ROM в качестве null-завершенной строки, и поэтому для их использования автоматически создается дескриптор-указатель `TPtrC`. Такой способ “дороже” доступа к константе, заданной с помощью `_LIT()`, ровно на 4 байта памяти, что весомо для строк небольшой длины. Поэтому макрос `_L()` рекомендуется использовать только на этапе разработки и тестирования. В большинстве примеров данной книги используется макрос `_L()`, а не `_LIT()` — это сделано для простоты и не должно вводить вас в заблуждение. Несмотря на то, что экономия нескольких байт может показаться несущественной, я настоятельно советую придерживаться данных рекомендаций и избегать необоснованного использования макроса `_L()`.



В Carbide.c++ 2.x вы можете просто набрать в строке `lit` и нажать `<Ctrl+Space>` — IDE превратит ее в шаблон макроса. Познакомьтесь с другими шаблонами автозамены (например, для стека очистки и циклов `for` и `while`), а также задать собственные, можно в окне, которое открывается при выборе команды меню `Window⇒Preferences⇒C/C++\Editor⇒Templates`.

При записи строки в макросах `_L()` или `_LIT()` необходимо бережно обращаться с символом обратный слеш “\”. Он является исключением (escape-символом) для задания управляющих последовательностей и спецсимволов — они перечислены в табл. 5.3.

Таблица 5.3. Управляющие последовательности и спецсимволы

Обозначение	Символ
<code>\\</code>	Обратный слеш
<code>\"</code>	Кавычки
<code>\0</code>	Символ с кодом 0x00
<code>\a</code>	Символ <code></code>
<code>\b</code>	Backspace
<code>\f</code>	FormFeed
<code>\n</code>	Новая строка (NL)
<code>\r</code>	Возврат каретки (CR)
<code>\t</code>	Горизонтальная табуляция
<code>\v</code>	Вертикальная табуляция

На практике часто применяются только `\\`, `\n` и `\t`. Очень полезны управляющие последовательности `\xAAA` (где AAA — код символа) и `\uXXXX` (где XXXX — код символа в шестнадцатеричной форме), предназначенные для вставки произвольного символа. Примеры.

```
TBuf<100> b;

_LIT(KPath, "c:\\data\\data.txt");
b = KPath; // b = "c:\\data\\data.txt", Length() = 16

_LIT(KApp, "My\\u00AE Application\\x169");
b = KApp; // b = "My@ Application@"

_LIT(KRussian, "\\u0420\\u0443\\u0441\\u043A\\u0438\\u0439");
b = KRussian; // b = "Русский"
```

Для того чтобы использовать русский язык в символьных дескрипторах, необходимо в ММР-файл добавить такие строки.

```
OPTION GCCE -finput-charset=Cp1251
OPTION CW -enc system
```

Очень внимательно следует относиться к записи путей файловой системы. Иногда, если вы ошибетесь и запишете один обратный слеш вместо двойного, такой символьный дескриптор все же пройдет этап компиляции. В подобном случае проблемы у вас возникнут уже во время работы программы, непосредственно при обращении к файлу или папке.

Подготовка к сертификации ASD

- Знание методов работы с символьными дескрипторами.
 - Понимание разницы между символьными дескрипторами, созданными с помощью макросов `_LIT()` и `_L()`, а также недостатков последнего.
 - Знание того, что использование макроса `_L()` не рекомендуется.
-

Дескрипторы-буферы TBuf и TBufC

Классы дескрипторов-буферов `TBuf` и `TBufC` одни из самых простых в использовании и часто применяются для работы с небольшими строками. Их схема размещения элементов в памяти показана на рис. 5.3.

TBufC	Тип	Размер	Строка	
	Тип	Размер	Максимальный размер	Строка

Рис. 5.3. Размещение в памяти объектов `TBuf` и `TBufC`

Дескрипторы TBuf и TBufC являются тонкими классами-шаблонами, требующими указания размера строки в виде целого числа TInt при объявлении. Пример.

```
const TInt KMaxNameLength = 20;
// Объявим буфер макс. длины = KMaxNameLength
TBufC<KMaxNameLength> name; // Сейчас он пуст
/* Использовать константу для задания
максимального размера TBuf не обязательно */
TBuf<100> buf;
```

Дескрипторы-буферы могут быть проинициализированы в момент создания с помощью данных любого другого дескриптора (чаще — - символического) или null-завершенной строки.

```
_LIT(KMyName, "Aleksandr");
TBufC<KMaxNameLength> name(KMyName);
TBuf<100> buf(_L("My name is: ")); // Не самый лучший способ
```

Узнать текущую длину хранящейся в дескрипторе строки вам поможет метод Length(), определенный в TDesC. Изменяемым дескрипторам дополнительно доступен метод MaxLength() класса TDes.

В классах TBuf и TBufC определен оператор = и метод Copy(), позволяющие заменить содержащуюся в дескрипторе строку значением любого другого дескриптора. Небольшой пример.

```
// buf.Length() = 20 (KMaxNameLength = 20)
// name.Length() = 12 ("My name is: ")
if (name.Length() >= buf.Length())
    name = buf; // Теперь name и buf содержат одинаковую строку
```

Заметьте, что в данном случае такое копирование возможно лишь потому, что длина текущего значения дескриптора buf оказалась меньше, чем размер неизменяемого дескриптора. Объект buf способен хранить строку длиной до 100 символов, и если бы мы перед копированием не убедились, что текущая длина значения buf не превышает размеров name, то такое действие могло бы вызвать панику USER 11, и приложение немедленно бы аварийно завершилось. Вызов паники может показаться излишне суровой реакцией, но так как операции со строками происходят чрезвычайно часто, в том числе и в ядре, то введение каких-то проверок и механизмов уведомления об ошибках было бы накладным с точки зрения производительности. Поэтому система реагирует именно так, и вам ничего не остается, кроме как внимательно следить за размерами используемых вами строк и дескрипторов.

При объявлении дескриптора-буфера не стоит излишне завышать его максимальный размер. Дескрипторы TBuf и TBufC часто создаются в стеке, и большое количество крупных дескрипторов способно привести к исчерпанию его памяти. Поэтому рекомендуется использовать дескрипторы-буферы для небольших строк: до 128 Unicode-символов или 256 ASCII-символов. Тем не ме-

нее в системных API вы можете встретить использование довольно больших дескрипторов: например, класс `TFileName`, переопределенный от `TBuf<0x100>`. Объект такого класса займет 520 байт. Шестнадцать таких объектов переполняют стек стандартного размера (8 Кбайт в приложениях без GUI). Если вы вынуждены работать с большими дескрипторами-буферами и не уверены, хватит ли вам памяти стека, можете создавать их в памяти кучи с помощью оператора `new`, использовать через указатель и удалять после завершения операции с помощью оператора `delete`. Пример.

```
_LIT(KPath, "c:\\data\\data.txt");
TFileName* filename = new TFileName(KPath);
// Работа с filename
delete filename;
```

Данный пример годится, если до удаления дескриптора не происходит обращения к методам, способным вызвать сброс. Если такие вызовы имеются, то указатель следует помещать в стек очистки.

На практике для выделения большого дескриптора-буфера в куче чаще всего поступают проще — объявляют его членом C-класса. Такой подход не требует работы с указателями и заботы об освобождении занимаемой им памяти. Я рекомендую по возможности поступать именно так.

Надеюсь, я вас не сильно напугал. На самом деле контроль над размерами дескрипторов — не самая сложная задача, и с течением времени вы привыкните бережно относиться к памяти. Ошибки при работе с дескрипторами характерны для новичков и у более опытных Symbian C++-разработчиков встречаются крайне редко.

Класс `TDes` предоставляет изменяемому дескриптору `TBuf` ряд методов для редактирования строки. При работе с ними также следует помнить о максимально допустимых размерах результата. Для краткости мы опустили аргументы и результат функций, большинство из них также многократно перегружено.

- `Append()` — различные варианты этого метода позволяют добавить к текущему значению строку из другого дескриптора или области памяти. Вы также можете воспользоваться оператором `+=`.
- `AppendNum()` — добавление к строке числовых значений (целочисленных и действительных) заданного формата.
- `Replace()` — замена части строки.
- `Insert()` — вставка подстроки в строку.
- `Delete()` — удаление подстроки.
- `Zero()` — очищение содержимого строки.
- `Num()` — эквивалент последовательного вызова `Zero()` и `AppendNum()`.
- `FillZ()` — заполнение всей строки нулевыми байтами (не забывайте, что дескриптор может хранить бинарные данные). После этой операции текущая длина строки будет равна максимальной. Аналогичный метод `Fill()` позволяет заполнить строку символом с произвольным кодом.

- Repeat() — удаление текущей строки и заполнение дескриптора подстроками до максимального размера. Если последняя подстрока не помещается, она будет обрезана.
- SetLength() — меняет текущий размер строки на заданный.
- SetMax() — устанавливает текущий размер строки равным ее максимальной длине.
- Swap() — обмен между данным дескриптором и дескриптором, указанным в качестве аргумента, хранящимися в них строками.
- Justify() — копирует строку в дескриптор и выравнивает ее (по центру или одному из краев), дополнив слева и/или справа символом с указанным кодом до заданной ширины. Аналогичный метод AppendJustify() выполняет подобные действия, добавляя символы к имевшейся строке.

Примеры.

```
TInt size = buf.Length();      // size = 12
TInt max = buf.MaxLength();    // max = 100;

buf.Append(name);              // buf = "My name is: Aleksandr"
buf.AppendNum(max, EHex);      // buf = "My name is: Aleksandr64"

/* EHex - элемент перечисления TRadix, позволяющий получить
   значение 100 в шестнадцатеричном виде: 0x64 */

buf.Num(size);                 // buf = "12"
buf.Zero();                    // buf = ""
// Еще один способ обнулить дескриптор: buf = KErrNullDesC

buf.Justify(_L("Symbian"), 15, ECenter, TChar(0x20));
// buf = "      Symbian      ", TChar(0x20) - задает символ ' '

// Пример конвертирования действительного значения в строку
TBuf<10> ver;

TRealFormat realFormat; //Формат представ. действительных чисел
// Зададим формат на основе текущих настроек локализации
realFormat.iType = KRealFormatFixed;
realFormat.iWidth = 10;
TLocale locale; // Настройки локализации системы
realFormat.iPlaces = locale.CurrencyDecimalPlaces();
realFormat.iPoint = locale.DecimalSeparator();
realFormat.iTriad = locale.ThousandsSeparator();
realFormat.iTriLen = (locale.CurrencyTriadsAllowed() ? 1 : 0);

ver.Num(9.1, realFormat); // ver="9.10" для английской locale

buf.Insert(12, ver);          // buf = "      Symbian 9.10      "
```

Для форматирования содержащейся в дескрипторе строки применяются следующие методы.

- `Trim()` — удаление пробелов на концах строки.
- `TrimLeft()` и `TrimRight()` — удаление пробелов с одного из концов строки.
- `TrimAll()` — аналогичен `Trim()`, за исключением того, что все содержащиеся в строке множественные пробелы заменяются одним пробелом.
- `LowerCase()` и `UpperCase()` — перевод всех символов строки в нижний и верхний регистр соответственно.
- `Capitalize()` — первый символ переводится в верхний регистр, остальные — в нижний.
- `Format()` — чрезвычайно мощный метод, позволяющий отформатировать текст и скопировать его в дескриптор, заменив текущее значение. Подробный список параметров форматирования для этого метода можно найти в справочнике SDK.

Примеры.

```
buf = _L(" SAMPLE   STRING ");
buf.TrimRight();           // buf = " SAMPLE   STRING"
buf.Trim();                // buf = "SAMPLE   STRING"
buf.TrimAll();             // buf = "SAMPLE STRING"
buf.LowerCase();           // buf = "sample string"
buf.UpperCase();           // buf = "SAMPLE STRING"
buf.Capitalize();          // buf = "Sample string"

_LIT ( KFormatString, "%S: %05d%S" );
buf.Format(KFormatString(), &_L("Distance"), 170, &_L("Km") );
// buf = "Distance: 00170Km"
```

Содержимое дескрипторов можно сравнивать. Для этого в классе `TDesC` определены операторы `!=`, `==`, `>`, `<`, `>=` и `<=`. Их работа сводится к вызову метода `Compare()`, в котором идет посимвольное сравнение строк дескрипторов. Два дескриптора считаются равными, если длина содержащихся в них строк одинакова, а сами строки посимвольно эквивалентны. Если одна из строк короче, то она признается меньшей. Если строки равной длины посимвольно неравны, то меньшей признается та, код символа которой в первой неравной паре меньше, например: `"abcd" == "abcd"`, `"abcd" > "ba"`, `"abcd" < "bacd"`.

Класс `TDesC` также предоставляет дескрипторам ряд методов для поиска в них подстроки, отдельного символа или проверки на соответствие заданной маске.

- `Find()` — поиск подстроки в строке. Результатом является позиция, в которой был найден аргумент, либо значение `KErrNotFound` (−1).
- `Locate()` и `LocateReverse()` — поиск символа с заданного символа в прямом и обратном направлении соответственно. Результат работы метода аналогичен `Find()`.

- `Match()` — проверка на соответствие строки дескриптора переданной в качестве аргумента маске. Маска может содержать символ “*” для задания группы неопределенных символов (группа может быть пустой) и “?” для задания неопределенного символа (должен существовать). Замечу, что никаких исключений данный метод не имеет, и просто поискать символы “*” и “?” с его помощью вы не сможете. Результатом является позиция, в которой определено соответствие маски строке, либо значение `KErrNotFound` (–1).

Примеры.

```
_LIT(KNums, "1234567890");
TBuf<100> b(KNums);

TInt pos = b.Find(_L("567")); // pos = 4

TChar ch(0x32); // 0x32 - код символа '2'
pos = b.LocateReverse(ch); // pos = 1

_LIT(KMask, "*0?");
pos = b.Match(KMask); // pos = -1 (KErrNotFound)
```

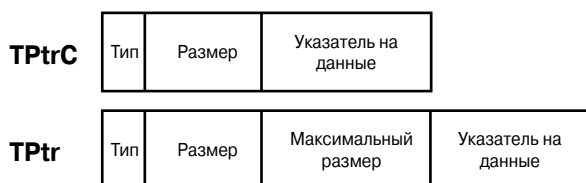
Подготовка к сертификации ASD

- Умение определять правильное и неправильное использование методов редактирования `TDesC` и `TDes`.
 - Знание того, что создание большого числа объектов `TFileName` и других больших дескрипторов в стеке может переполнить его память.
 - Понимание разницы между методами `Size()`, `Length()` и `MaxLength()`.
-

Дескрипторы-указатели

Давайте представим ситуацию, в которой у разработчика возникает задача выделения всех слов предложения (например, для последующей сортировки по алфавиту). Предположим, что мы имеем довольно большой дескриптор, содержащий текст из 100 слов. Многие решали бы такую задачу с помощью массива дескрипторов, но это не самый эффективный способ, особенно в Symbian OS, сражающейся за каждый байт системной памяти. Немного поразмыслив, вы придете к более правильному решению: использовать указатели на расположенные в дескрипторе слова, храня информацию об их размере. Именно такой способ работы со строками предоставляют нам дескрипторы-указатели `TPtrC` и `TPtr`.

На рис. 5.4 изображена схема размещения в памяти объектов дескрипторов-указателей. Как видите, она аналогична размещению `TBuf` и `TBufC` за исключением того, что вместо строки используется указатель на нее. Таким образом, объекты `TPtr` и `TPtrC` могут ссылаться на строки или подстроки, хранимые любыми другими типами дескрипторов. Эти данные могут находиться как в стеке, так и в куче.

Рис. 5.4. Размещение в памяти объектов *TPtr* и *TPtrC*

Объект *TPtrC* можно объявить без инициализации и в дальнейшем задать ему значение с помощью метода *Set()*. Изменяемый дескриптор-указатель *TPtr* всегда требует передачи адреса памяти, на который он указывает, и размера данных в конструкторе. В дескрипторе любого типа присутствует метод *Ptr()*, возвращающий указатель на адрес начала хранящейся в нем строки, который вы можете использовать для инициализации *TPtr*. Примеры.

```
TBuf<100> b(_L("0123456789"));
TPtrC p1;
p1.Set(b); // p1 указывает на строку из b

TPtrC p2(p1); // Теперь и p2 указывает на строку из b

TPtr p3(NULL,0); // Созд. пустой изменяемый дескриптор-указатель
p3.Set((TUint16*)b.Ptr(), b.Length(), b.MaxLength());

// Все то же самое можно проделать в конструкторе
TPtr p4((TUint16*)p3.Ptr(), p3.Length(), p3.MaxLength());
```

На практике объект дескриптора-указателя чаще не создают, а получают в результате работы одного из методов выделения подстроки. Класс *TDesC* предоставляет всем дескрипторам следующие методы, возвращающие константный дескриптор-указатель на подстроку.

- *Left()* — указатель на подстроку, начинающуюся с начала строки и имеющую заданную длину.
- *Right()* — указатель на подстроку заданной длины, выделенной с конца строки.
- *Mid()* — указатель на подстроку указанного размера, начинающуюся с заданной позиции.

Класс *TDes* дополняет эти методы аналогичными функциями с именами *LeftTPtr()*, *RightTPtr()* и *MidTPtr()*, возвращающими модифицируемый указатель. Максимальный размер хранимой в *TPtr* строки в этом случае задается равным размеру подстроки, для которой он получен. Пример.

```
TBuf<100> b(_L("0123456789"));

TPtrC p1(b.Right(5)); // p1 = "56789"
```

```
TPtr p2(b.LeftTPtr(5)); // p2 = "01234", p2.MaxLength() = 5
p2 = p1; // p2 = "56789", b = "5678956789"
```

Заметьте, что дескрипторы-указатели ведут себя точно так же, как обычные дескрипторы. Вы можете сравнивать и выполнять поиск в неизменяемом `TPtrC`, а также имеете полный набор методов для редактирования `TPtr`. Но учтите, что вносимые вами изменения на самом деле применяются к той строке или подстроке, на которую ссылается дескриптор-указатель. Это продемонстрировано в вышеприведенном примере. Обратите внимание, что при копировании данных в дескриптор-указатель их размер не должен превышать его заявленной максимальной длины. В нашем случае оба объекта могут хранить строку до 5 символов длиной. Если вы попытаетесь скопировать в `p2` больший объем данных — это приведет к панике.

Дескрипторы-указатели никогда не владеют памятью, на которую ссылаются, и не отвечают за ее освобождение. Более того, может возникнуть ситуация, когда строка уже удалена, а дескриптор-указатель все еще существует. В этом случае обращение к нему может привести к панике.

Подготовка к сертификации ASD

- Понимание разницы между методами `Copy()` и `Set()`, а также умение правильно выполнять присваивание.
-

Хранение строк в памяти кучи. Дескрипторы `HBufC` и `RBuf`

Для хранения строковых данных большого размера используются дескрипторы кучи: неизменяемый `HBufC` и изменяемый `RBuf`. Они также используются при работе со строками, длину которых невозможно оценить заранее.

Почему `HBufC`? Конечно же, в Symbian нет никаких H-классов. На самом деле `HBufC` порожден от `TDesC`, как и любой другой неизменяемый дескриптор, но не вписывается в концепцию T-классов, так как не может создаваться в стеке (его конструктор объявлен приватным). Поэтому префикс имени класса заменен на “H” — от *heap* (куча).

Схема размещения объектов `HBufC` в памяти аналогична `TBufC`. Так как объекты `HBufC` создаются в куче, то доступ к ним осуществляется только по указателям. Сам дескриптор хранится вместе с данными в выделенной ему ячейке памяти кучи. Новый объект класса `HBufC` создается с помощью статических методов `NewL()` и `NewMaxL()`. Они принимают в качестве аргумента размер буфера (в символах), необходимый для хранения строки. Фактически при этом может выделяться даже больше памяти, чем вы запросили — это зависит от правил выравнивания границ ячейки памяти кучи устройства. Метод `NewMaxL()` дополнительно увеличивает длину строки до максимума, не инициализируя ее.

```

HBufC* h1 = HBufC::NewL(255); // h1->Length() = 0
delete h1;

HBufC* h2 = HBufC::NewMaxL(512); // h2->Length() >= 512
delete h2;

```

Возможность задавать размер буфера дескриптора непосредственно в процессе работы чрезвычайно важна. Зачастую разработчик не может заранее оценить длину строки, которую ему предстоит обработать. В такой ситуации имеет смысл использовать HBufC, даже если в 99% случаев строка будет достаточно малой для хранения в стеке.

Если методу NewL() или NewMaxL() не удалось выделить достаточный объем памяти для создания объекта, произойдет сброс. Не забывайте, что, как и за любой другой созданный в куче объект, именно вы несете ответственность за освобождение памяти выделенной объектам HBufC.

Класс HBufC также предоставляет варианты функций, позволяющие поместить объект в стек очистки сразу после создания.

```

HBufC* h1 = HBufC::NewLC(255);
HBufC* h2 = HBufC::NewMaxLC(512);
<...> // Работа с дескрипторами
CleanupStack::PopAndDestroy(2, h1);

```

Если вы хотите воспользоваться дескриптором HBufC в функции, которая не должна приводить к сбросу, то вам подойдут методы New() и NewMax(). В случае, если при создании объекта произойдет ошибка, их результатом будет NULL, поэтому вам стоит проверять указатель на дескриптор перед началом работы с ним.

```

TInt SomeFunction()
{
    HBufC* h = HBufC::New(100);
    if (!h)
        return KErrNoMemory;
    <...>
    delete h;
    return KErrNone;
}

```

Помещение строки в дескриптор HBufC и чтение данных из него выполняется следующим образом.

```

_LIT(KTest, "abc...");
HBufC* h = HBufC::NewL(100);
*h = KTest; // Заменяем содержимое неизменяемого дескриптора
TBuf<10> b;
b = *h; // Чтение из HBufC
delete h;

```

Дескрипторы всех типов имеют реализованные в TDesC методы Alloc() и AllocL(). С их помощью вы легко можете создать объект HBufC, инициализированный хранящейся в дескрипторе строкой.

```
_LIT(KTest, "abc...");
// KTest() создает TDesC для символьного дескриптора
HBufC* h = KTest().AllocL();
// В h содержится копия строки "abc..."
TBuf<10> b = *h; // b = "abc..."
delete h;
```

Дескрипторы, создаваемые в куче, имеют одно очень важное свойство — они способны изменять размер буфера для хранения строки с помощью методов ReAlloc() и ReAllocL(). Фактически при этом выполняются следующие операции: создается новый дескриптор с буфером требуемого размера, в него копируются данные из старого дескриптора, после чего старый дескриптор уничтожается.

```
_LIT(KTest, "abc");
HBufC* h = HBufC::NewMaxL(100);

TPtr ph(h->Des()); // ph: Length = 100, MaxLength = 100
ph.Repeat(KTest);
//Дескриптор полностью заполнен последовательностью "abccabc.."

h = h->ReAllocL(1000);
ph.Set(h->Des());

// Теперь мы можем добавить еще 900 символов
ph.AppendFill(TChar(0x20), 900); // Дополним строку пробелами
```

Обратите внимание, что HBufC не увеличивает свой размер автоматически — вы все также можете получить панику USER 11, выйдя за его границы. В данном примере мы создаем изменяемый дескриптор-указатель для HBufC, чтобы заполнить его. Дескриптор любого типа имеет метод Des(), позволяющий получить такой указатель. Редактирование содержимого HBufC — довольно часто используемый на практике прием. Это связано с тем, что длительное время в Symbian C++ не было класса изменяемого дескриптора для хранения строк в куче.

Распространенной ошибкой является использование создаваемого методом Des() объекта TPtr в тех случаях, когда вполне можно обойтись неизменяемым дескриптором (при чтении или замене содержимого), так как это — более ресурсоемкая операция, чем простое разыменование указателя HBufC*.

Мы используем для создания дескриптора в куче метод NewMaxL() вместо NewL() для того, чтобы в нем появилась строка ненулевой длины. Если бы мы этого не сделали, то не смогли бы инициализировать ее с помощью Repeat().

В результате работы ReAlloc() дескриптор может полностью изменить свое местоположение в памяти кучи. Поэтому после изменения максимального раз-

мера HBufC дескрипторы-указатели могут потерять ссылку на его данные. Чтобы этого не произошло, необходимо снова назначить их с помощью Set().

Изменяемый дескриптор для хранящихся в куче строк появился лишь в Symbian 8.x. Долгое время шли споры о том, нужен ли аналог HBuf, и каким он должен быть. В результате разработчикам предложили класс RBuf.

Как и любой другой изменяемый дескриптор, RBuf унаследован от TDes, но в тоже время является R-классом. Это означает, что он владеет и управляет внешним ресурсом. Таким ресурсом является объект HBufC. В действительности RBuf инкапсулирует HBufC* и ведет себя аналогично TPtr. Схема размещения объектов RBuf в памяти совпадает с объектами TPtr. Экземпляр RBuf может создать собственный буфер в куче с помощью метода CreateL() либо принять управление над уже существующим дескриптором HBufC или выделенной памятью с помощью метода Assign(). В любом случае RBuf несет ответственность за освобождение ресурса, которым он владеет.

```
HBufC* h = HBufC::NewMaxL(100);
*h = _L("My ");

RBuf r;
r.Assign(h); // Берет на себя управление памятью h
// Последние 2 строки аналогичны RBuf r(h);

r.Append(_L("Name")); // r = "My name"
TBuf<10> b = r; // Чтение данных из RBuf

r.Close(); // Буфер h уничтожается RBuf
```

Объект RBuf освобождает память, которой он владеет, при вызове метода Close(), после чего он может быть использован повторно. Вам не следует забывать о том, что R-классы, владеющие хендлом на открытые ресурсы, в случае обращения к функциям способным вызвать сброс, должны помещаться в стек очистки. Это можно сделать с помощью метода CleanupClosePushL().

```
RBuf r;
r.CreateL(1024); // Создаем RBuf с собственным буфером
r.CleanupClosePushL();
// Вызов небезопасного метода, например r.ReAllocL()
// r.Close() вызывается стеком очистки
CleanupStack::PopAndDestroy(&r);
```

Редактировать объект класса RBuf можно точно так же, как TPtr или TBuf.

Подготовка к сертификации ASD

- Понимание того, что не существует класса HBuf, но в качестве изменяемого дескриптора с динамически создаваемым буфером может использоваться класс RBuf.
- Умение отличать правильные способы создания объекта HBufC.

- Умение использовать класс `RBuf`
- Понимание, когда стоит получать изменяемый дескриптор (`TDes&`) с помощью метода `Des()`, а когда можно ограничиться неизменяемым дескриптором (`TDesC&`), разыменовывая указатель `HBufC*`.

8- и 16-битовые дескрипторы. Кодировки

Итак, мы познакомились с вами с восемью классами дескрипторов. На самом деле их шестнадцать. Все это время для простоты повествования я умалчивал о размере символа в хранимых дескрипторами строках. Теперь мы внесем окончательную ясность в этот вопрос.

Symbian поддерживает два типа строк: состоящие из 8- и 16-битовых символов. Строки 8-битовых символов хорошо подходят для бинарных данных, а также текста в ASCII-кодировке. Строки 16-битовых символов предназначены для хранения Unicode-текста. Для работы с ними определены две группы классов дескрипторов:

- хранящие строки 8-битовых (*narrow*) символов: `TDes8`, `TDesC8`, `TBuf8`, `TBufC8`, `TPtr8`, `TPtrC8`, `HBufC8` и `RBuf8`;
- хранящие строки 16-битовых (*wide*) символов: `TDes16`, `TDesC16`, `TBuf16`, `TBufC16`, `TPtr16`, `TPtrC16`, `HBufC16` и `RBuf16`.

Существует также два набора макросов для задания символьных дескрипторов: `_LIT8()`, `_L8()` и `_LIT16()`, `_L16()`.

Все классы, название которых не содержит указания на размер символа строки (`TDes`, `TDesC`, `TBuf`, `TBufC`, `TPtr`, `TPtrC`, `HBufC` и `RBuf`), называются **нейтральными** (*neutral*), так как размер содержащихся в них символов определяется во время компиляции и зависит от платформы. На самом деле они являются всего лишь псевдонимами для одной из двух вышеперечисленных групп классов, заданными с помощью оператора `typedef`. Во времена Ерос операционная система по умолчанию работала с ASCII-строками. Но Symbian OS использует Unicode (USC-2), поэтому во всех известных мне платформах и версиях SDK классы нейтральных дескрипторов соответствуют 16-битовым аналогам. Маловероятно, что в обозримом будущем это изменится.

Таким образом, все, что мы с вами изучали на примере нейтральных дескрипторов, на самом деле проделывалось с Unicode-дескрипторами. Я советую и впредь при работе с 16-битовыми классами использовать их нейтральные псевдонимы в качестве названий. Эта запись короче и дает некоторые преимущества при портировании на гипотетические будущие платформы.

Что же касается 8-битовых дескрипторов — они абсолютно идентичны Unicode-классам. Вы можете взять любой пример, добавить “8” к именам классов и макросов — и получите готовый пример для работы с ASCII-строками.

Помимо метода `Length()`, возвращающего размер строки в символах, во всех дескрипторах имеется метод `Size()`, позволяющий определить размер

строки в байтах. Как не сложно догадаться, в 8-битовых дескрипторах результат работы этих функций всегда совпадает, а в дескрипторах, порожденных от TDesC16, значение Size() в два раза больше Length(). Эти методы легко перепутать — будьте внимательны!

Единственная проблема, с которой сталкивается разработчик при работе с 8- и 16-битовыми строковыми данными, — это конвертация. Дескрипторы всех типов имеют метод Copy(), позволяющий заменить текущую строку данными из другого источника. Таким источником может быть область памяти или другой дескриптор, в том числе — дескриптор, содержащий символы другого размера. Пример.

```
_LIT8(KText, "ASCII");
TBuf8<10> b8(KText); // b8 = "ASCII"
TInt sz = b8.Size(); // sz = 5

TBuf<10> b16;
b16.Copy(b8); // b16 = "ASCII"
sz = b16.Size(); // sz = 10
b8.Copy(_L("Unicode")); // b8 = "Unicode"
```

Казалось бы, все просто. Но давайте рассмотрим, каким образом происходит такая конвертация. Сначала мы переносим строку "ASCII", состоящую из 8-битовых символов, в Unicode-строку. В 8-битовой строке слово "ASCII" занимает 5 байт и выглядит следующим образом: 0x41, 0x53, 0x43, 0x49, 0x49. В Unicode-строке каждый символ занимает 2 байта. Каким образом метод Copy получает недостающий байт для каждого символа? Очень просто — он добавляет байт 0x00 в конец каждого символа. Таким образом, строка "ASCII" в Unicode будет занимать 10 байт и выглядеть так: 0x41, 0x00, 0x53, 0x00, 0x43, 0x00, 0x49, 0x00, 0x49, 0x00. И это правильно, поскольку латинские символы и в ASCII и в Unicode имеют одинаковый код.

Что же происходит при копировании 16-битовой строки "Unicode" в 8-битовый дескриптор? Метод Copy() просто отбрасывает второй байт в каждом символе и записывает нечетные в новую строку. Таким образом, из 16-битовой строки 0x55, 0x00, 0x6E, 0x00, 0x69, 0x00, 0x63, 0x00, 0x6F, 0x00, 0x64, 0x00, 0x65, 0x00 мы получаем 0x55, 0x6E, 0x69, 0x63, 0x6F, 0x64, 0x65, что опять-таки верно для латинских символов.

Но если вы считали в дескриптор из файла ASCII или Unicode текст, содержащий нелатинские символы, то при подобном конвертировании получите нечитабельный текст. Например, буква “Р” кириллицы в ASCII имеет код 0xD0, а соответствующий Unicode-символ выглядит как 0x20, 0x04. Таким образом, метод Copy() не годится для переноса содержащих нелатинские символы строк между 8- и 16-битовыми дескрипторами.

Для правильного конвертирования данных из ASCII (или любой другой кодировки) в Unicode и обратно служит класс CCnvCharacterSetConverter. При его использовании сначала необходимо вызвать метод PrepareToConvertToOrFromL() с уникальным идентификатором кодировки (в нашем

случае — Windows-1251), а затем выполнить конверсию данных с помощью методов `ConvertToUnicode()` или `ConvertFromUnicode()`. Работа класса требует передачи ему сессии файлового сервера (RFs), более подробно с этим вопросом мы познакомимся в *главе 6*, раздел “Работа с файловой системой”. Для работы приведенного ниже примера необходимо подключить заголовочные файлы `f32file.h`, `charconv.h`, а также библиотеки `efsrv.lib` и `charconv.lib`.

```
RFs fs; // Сессия с файловым сервером
User::LeaveIfError(fs.Connect()); // Подключаемся к серверу
CleanupClosePushL(fs); // Поместим сессию в стек очистки

// Создание объекта CCnvCharacterSetConverter в стеке очистки
CCnvCharacterSetConverter* cnv =
    CCnvCharacterSetConverter::NewLC();

// Идентификатор кодировки Windows-1251
const TInt KCharacterSetIdentifierCodePage1251 = 0x100059D7;
if (cnv->PrepareToConvertToOrFromL(
    KCharacterSetIdentifierCodePage1251, fs)
    != CCnvCharacterSetConverter::EAvailable)
    User::Leave(KErrNotSupported);

// Данные для конвертирования в ASCII и буфер для них в Unicode
TBuf<100> ascii(_L8("\xD0\F3\F1\F1\xEA\E8\E9"));
// "Русский"
TBuf<100> ucs2;

TInt state = CCnvCharacterSetConverter::KStateDefault;
TInt res = cnv->ConvertToUnicode(ucs2, ascii, state);
// ucs2 = \x0420\x0443\x0441\x0441\x043A\x0438\x0439
// ("Русский")

// В res — кол-во несконвертированных символов должно быть 0

CleanupStack::PopAndDestroy(2);
```

Еще один служебный статический класс `CnvUtfConverter` позволяет конвертировать дескриптор из кодировки Unicode (UCS-2) в UTF-7 или UTF-8 и наоборот. Он объявлен в заголовочном файле `utf.h` и требует подключения библиотеки импорта `charconv.lib`. Небольшой пример.

```
TBuf<100> utf8;
CnvUtfConverter::ConvertFromUnicodeToUtf8(utf8, ucs2);
```

Подготовка к сертификации ASD

- Знание того, что дескрипторы Symbian OS могут хранить как текст, так и бинарные данные.

- Знание того, что дескрипторы могут быть 8-битовыми (narrow), 16-битовыми (wide) и нейтральной длины (neutral). Понимать назначение дескрипторов нейтральной длины.
- Понимание того, что дескрипторы не могут динамически изменять свой размер и вызывают панику в случае переполнения.
- Знание того, как конвертируются 8-битовые дескрипторы в 16-битовые и наоборот с помощью метода `Copy()` или класса `CnvUtfConverter`.

Выбор подходящего класса дескриптора

Если вы портируете код с C на Symbian C++, то вам, возможно, пригодится таблица соответствия классов дескрипторов строкам из C (табл. 5.4). Диаграмма, приведенная на рис. 5.5, поможет вам сориентироваться в классах дескрипторов, если вы проектируете приложение с нуля.

Таблица 5.4. Соответствие классов дескрипторов в языке Symbian C++ строкам из языка C

Symbian C++	C++
TLitC	static const char[]
TBuf	char[]
TBufC	const char[]
TPtr	char*
TPtrC	const char*
RBuf	char*
HBufC	const char*

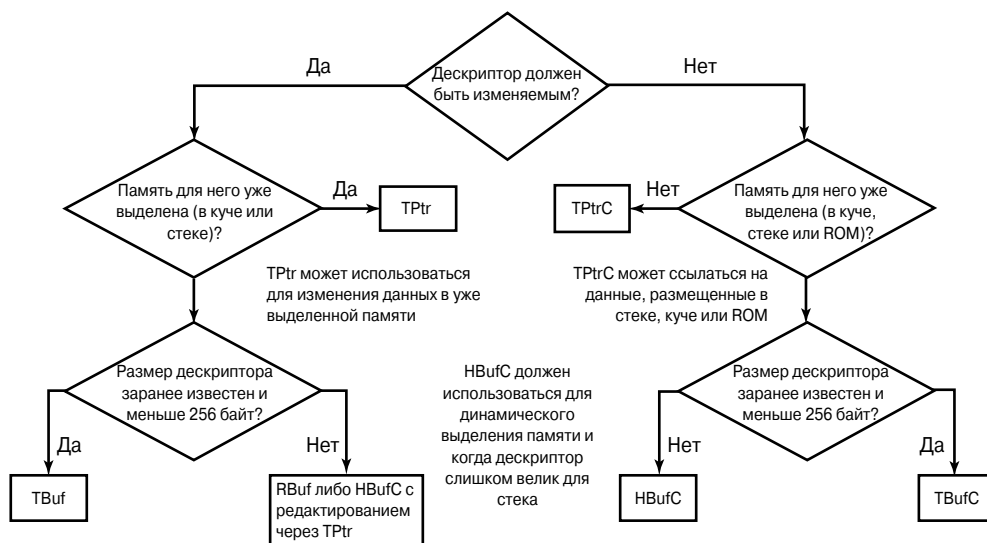


Рис. 5.5. Диаграмма выбора подходящего типа дескриптора

Если дескриптор содержит бинарные данные, вам следует использовать 8-битовый класс (например, `TBufC8`). Если дескриптор предназначен для хранения Unicode-строк, следует выбрать 16-битовый класс (например, `TBufC16`). Во всех прочих случаях следует использовать нейтральные классы дескрипторов.

Дескрипторы как аргументы и результат функций

Дескрипторы могут передаваться в функцию в качестве аргумента и возвращаться из нее в виде результата. Для передачи дескриптора в качестве аргумента рекомендуется воспользоваться базовыми классами. Пример.

```
void Func(const TDesC& aDes1, TDes& aDes2)
```

Использование базовых классов для объявления аргумента позволяет передавать в данную процедуру любые типы порожденных от них дескрипторов. Применение одного из порожденных классов (например, `TPtrC`) также возможно, но ограничивает возможности разработчика и редко бывает обоснованным.

Аргумент функции базового класса дескриптора может передаваться только по ссылке.

Если вы забудете “&”, то исходный код скомпилируется, но данные при передаче в функцию будут теряться. Это происходит потому, что базовые классы не имеют методов хранения данных.

В вышеприведенном примере аргумент `aDes1` доступен только для чтения. При его объявлении мы использовали неизменяемый класс `TDesC`, так как это базовый класс для всех дескрипторов. Помня о том, что даже неизменяемый дескриптор может быть изменен (если данные заменяются полностью либо через `Des()`), мы объявляем аргумент константным. Теперь `aDes1` действительно доступен только для чтения и достаточно общий, чтобы служить для передачи любых типов дескрипторов (даже изменяемых, так как они тоже порождены от `TDesC`).

Для объявления дескриптора-аргумента, доступного как для чтения, так и для записи, мы воспользовались классом `TDes` — базовым для изменяемых дескрипторов. Этот аргумент может использоваться для возвращения результата функции. Пример.

```
void ConcatDesL(const TDesC& aDes1,
               const TDesC& aDes2, TDes& aDes3)
{
    // Процедура, возвращающая в 3-м дескрипторе
    // Результат объединения строк первых двух аргументов
    if (aDes3.MaxLength() > aDes1.Length() + aDes2.Length())
    {
        aDes3 = aDes1;
        aDes3.Append(aDes2);
    }
}
```

```

    }
    else
        User::Leave(KErrBadDescriptor);
    // Если размера третьего аргумента не достаточно - сброс
}

```

Вот небольшой пример использования процедуры `ConcatDesL()`.

```

TBuf<10> buf(_L("12345"));
// Передача буфера и указателя на подстроку этого буфера
// в функцию
ConcatDesL(buf, buf.Left(3), buf);
// buf = "12345123"

// Получим HBufC из копии строки buf
HBufC* hbuf = buf.AllocLC();
// Оставим hbuf в стеке так как ContactDesL может вызвать сброс
_LIT(KTest, "test");
// Передается символьный дескриптор и дескриптор кучи
TBuf<20> res; // Увеличим размер буфера до 20
ConcatDesL(KTest, *hbuf, res);
// res = "test12345123"
// Не забываем освободить память
CleanupStack::PopAndDestroy(hbuf);

```

Как видите, использование базовых классов в качестве аргументов процедуры позволяет нам вызывать ее с любыми типами дескрипторов.

Дескриптор может также возвращаться в качестве результата функции. Пример.

```

const TDesC& CSomeClass::GetDes()
{
    return iDes;
}

```

Обратите внимание, что функция возвращает значение дескриптора, являющегося членом класса `CSomeClass` (об этом нам сигнализирует его имя). Действительно, мы не можем возвращать по ссылке из функции автоматическую переменную. Поэтому следующий код будет неверен.

```

const TDesC& GetDes()
{
    TBuf<100> b (_L("123"));
    return b;
}

```

Но мы можем вернуть указатель на дескриптор, созданный в куче. В этом случае ответственность за освобождение выделенной памяти несет тот, кто воспользовался данной функцией. Это необходимо отразить в документации. В таких случаях следует отказываться от использования базовых классов и возвращать значение `HBufC*`.

```

const HBufC* ConcatDesL(const TDesC& aDes1, const TDesC& aDes2)
{
    HBufC* res = HBufC::NewL(aDes1.Length() + aDes2.Length());
    res = aDes1;
    TPtr p(res->Des());
    p.Append(aDes2);
    // Ответственность за освобождение памяти результата
    // несет вызвавший функцию
    return res;
}

```

На практике помимо методов, возвращающих указатель на `HBufC`, часто встречаются функции, результатом которых является дескриптор-указатель класса `TPtr` или `TPtrC`. При использовании дескрипторов-указателей необходимо помнить, что они не владеют памятью, на которую ссылаются. Поэтому вы не можете вернуть из функции указатель `TPtr` от дескриптора, созданного в стеке. Чаще всего дескрипторы-указатели берутся от аргумента или члена класса. Пример.

```

TPtrC GetHalf(const TDesC& aDes)
{
    return aDes.Left(aDes.Length() \ 2);
}

```

В качестве результата функции можно также использовать дескриптор `TBuf` или `TBufC`, передающийся по значению, но такой подход сопряжен с рядом проблем (ограничивает размер аргументов) и на практике не используется. Пример.

```

const TInt KResSize = 100;
const TBufC<KResSize> ConcatDesL(const TDesC& aDes1,
                                const TDesC& aDes2)
{
    TBuf<KResSize> res;
    // Заполняем res
    return res;
}

```

Подготовка к сертификации ASD

- Умение правильно объявлять функции, аргументами которых являются изменяемые и неизменяемые дескрипторы.
-

Дескрипторы-пакеты `TPkg`, `TPkgC` и `TPkgBuf`

Классы дескрипторов `TPkg`, `TPkgC` и `TPkgBuf` принято рассматривать в отдельности от остальных. Их называют дескрипторами-пакетами, они не служат для хранения строк и являются вспомогательными классами для передачи

данных между процессами и потоками. Дескрипторы-пакеты позволяют оформить некий класс (чаще всего T-класс) в виде дескриптора (некоторые авторы называют этот процесс “дескриптизацией”), после чего данный объект может быть записан в файл или передан в другой процесс.

Несмотря на то, что классы дескрипторов-пакетов носят нейтральные имена — они связаны не с 16-, а с 8-битовым представлением данных. Объекты этих классов являются тонкими шаблонами и принимают в качестве параметра название класса, который в них будет храниться.

Классы TPckg и TPckgC на самом деле являются шаблоном для создания дескриптора-указателя TPtr8 и TPtrC8 соответственно. Они не владеют данными, на которые ссылаются. Класс TPckgBuf порожден от TDes8 и ведет себя аналогично TBuf8. При создании экземпляра класса TPckgBuf переданные ему в конструкторе данные копируются в дескриптор-пакет.

```
class TClass
{
    TInt iInt;
    TBuf<10> iSomeData;
    void SomeFunc();
};

TClass t;
TPckg<TClass> pckg(t); // pckg на самом деле TPtr8 для t
TPckgBuf<TClass> pckgbuf(t); // pckgbuf владеет копией t
```

Извлечь помещенный в дескриптор-пакет класс можно с помощью оператора (). Пример.

```
TClass a = pckgbuf();
```

Работа с дескрипторами пакетами ведется точно так же, как и с другими классами дескрипторов.

Лексический анализатор TLex

Класс TLex не является дескриптором, но мы рассмотрим его в данной главе, так как он используется для работы с дескрипторами. Класс TLex — это мощный лексический анализатор, используемый для разбора содержащихся в строке дескриптора данных. Он предоставляет общие функции разбора строк, подходящие для конвертирования числовых форматов и парсинга синтаксических элементов. Существуют два класса, TLex8 и TLex16, работающих с 8- и 16-битовыми строками соответственно, а также нейтральный псевдоним TLex, эквивалентный TLex16.

При создании объекту TLex можно передать дескриптор, данные которого необходимо разобрать либо присвоить их ему позднее с помощью метода Assign(). Экземпляр этого класса хранит строку, метку, отмечающую текущий анализируемый элемент и указатель на следующий символ, который будет про-

анализирован. Класс `TLex` позволяет находить и извлекать произвольные элементы строки, но наиболее часто применяется при необходимости получить из дескриптора числовое значение. Для этого служит метод `Val()`. Функция `Val()` перегружена для различных знаковых целочисленных типов: `TInt`, `TInt8`, `TInt16`, `TInt32`, `TInt64` — с или без проверки предельного значения; беззнаковых целочисленных типов, записанных в различных системах счисления (десятичной, шестнадцатеричной, двоичной или восьмеричной); и для `TReal`.

```
_LIT(KTest1, "215");
_LIT(KTest2, "0F00");
_LIT(KTest3, "3.14");
TLex l(KTest1);
TInt32 res(0);
l.Val(res); // res = 215
l.Assign(KTest2);
TUint ures(0);
l.Val(ures, EHex); // ures = 3840
TReal rx(0);
l.Assign(KTest3);
l.Val(rx); // rx = 3,14
```

Подробный демонстрационный пример возможностей класса `TLex` можно найти в папке `\Examples\Base\BufsAndStrings\Lexer\` вашей инсталляции SDK.

Подготовка к сертификации ASD

- Знание того, как используется класс `TLex` для конвертирования дескриптора в число и `TDes::Num()` для конвертирования числа в дескриптор.
 - Знание характеристик всех классов дескрипторов.
-

L-классы

Весной 2009 Symbian представила новую концепцию L-классов и выпустила библиотеку `EUserHL` (Euser High Level), содержащую их реализацию. Эта библиотека не входит в состав SDK. Ее дистрибутив можно скачать с сайта Symbian Foundation. Прямую ссылку вы можете найти в *приложении Б*. Данный дистрибутив устанавливается поверх SDK, а также содержит заголовочные файлы, документацию, примеры и саму библиотеку `EUserHL`. В нем вы также найдете SIS-пакет, который должен быть установлен на устройстве для работы с L-классами (так как на устройствах библиотеки `euserhl.dll` нет). Библиотека `EUserHL` доступна для всех SDK, начиная с Symbian 9.1.

Ввиду того, что L-классы появились сравнительно недавно, на практике они пока не применяются и, в целом не ясно, найдет ли эта концепция должную поддержку в сообществе разработчиков.

Итак, что же такое L-классы? Литера “L” в их названии происходит от “**L**iberal with **L**eaving” и указывает на то, что создание, копирование, передача и возвращение по значению, присваивание, а также все манипуляции с операторами данного класса потенциально способны привести к сбросу. Если это не так, подобные исключения должны описываться в документации. Второй особенностью L-классов является то, что они сами обеспечивают управление своей памятью и памятью ресурсов, которыми они владеют. Таким образом, их можно создавать и использовать подобно T-классам, и при этом необязательно помещать L-классы в стек очистки.

На данный момент текущая версия (1.2) библиотеки EUserHL предлагает следующие классы.

- LString — унаследованный от RBuf дескриптор. Единственный на данный момент класс дескриптора, способный автоматически увеличивать свой размер с ростом длины хранящейся в нем строки.
- Группа классов LManagedX (LManagedPtr, LManagedHandle и т.д.) — тонкие шаблоны, позволяющие автоматически уничтожить тем или иным способом помещенный в них объект, как только экземпляр LManagedX выходит за пределы видимости. Годаются только для членов класса.
- Группа классов LCleanedupX — аналогичны группе классов LManagedX и позволяют автоматически уничтожить тем или иным способом помещенный в них объект, как только экземпляр LCleanedupX выходит за пределы видимости. Могут использоваться для всех объектов, безопасная работа которых требует помещения в стек очистки.

Несмотря на то, что использование вышеназванных L-классов сильно упрощает код, я воздержусь от описания их работы в данной книге. Во-первых, они не являются частью SDK, а значит, с течением времени в них могут вноситься изменения. Во-вторых, их использование в ряде случаев идет вразрез с общепринятыми концепциями Symbian C++. Например, объекты LCleanedupX не рекомендуется использовать в функциях, обращающихся к стандартным методам CleanupStack. И это притом, что сам LCleanedupX реализован на основе стека очистки, а значит, без изучения CleanupStack с ним не разобраться. Хотя класс LString выглядит очень привлекательным для новичка, использование этого класса может помешать ему разобраться с обычными дескрипторами. И Symbian C++ непременно вам за это отомстит. Что же касается опытных разработчиков — они 10 лет прекрасно обходились без L-классов и с их точки зрения одна только необходимость предусматривать библиотеку EUserHL на устройство — уже достаточно большой недостаток этой новой концепции. На текущий момент мне не известно, планируется ли интеграция поддержки L-классов в SDK, либо они останутся в виде плагина, предназначенного для новичков и лиц, так и не сумевших в должной мере освоить Symbian C++.

Итак, я оставляю L-классы на ваше самостоятельное изучение. И призываю обращаться к ним только после того, как вы разберетесь с прочими нюансами Symbian C++. Возможно, к этому моменту будущее L-классов будет более определенным.

Динамические массивы

Symbian C++ предоставляет ряд классов для организации динамических массивов различных элементов. Сами массивы хранятся в буфере, размещенном в памяти кучи. Как вы помните, память в куче выделяется в форме ячеек определенного размера. Так как массив динамический, не исключено, что с течением времени для хранения его элементов потребуется увеличить буфер памяти. Вполне вероятно, что свободного места в ячейке для этого не хватит.

Эту проблему можно решать двумя способами: либо менять размер имеющейся ячейки, либо создать новую и связать ее с предыдущей указателями. В Symbian C++ есть классы, реализующие как первый, так и второй метод выделения памяти в кучи (рис. 5.6). Те из них, что всегда хранят данные в одном буфере единственной ячейки памяти кучи, называют массивами с **плоским буфером** (flat buffer). Классы, размещающие элементы массива в связанном списке ячеек, называют массивами с **сегментированным буфером** (segmented buffer).

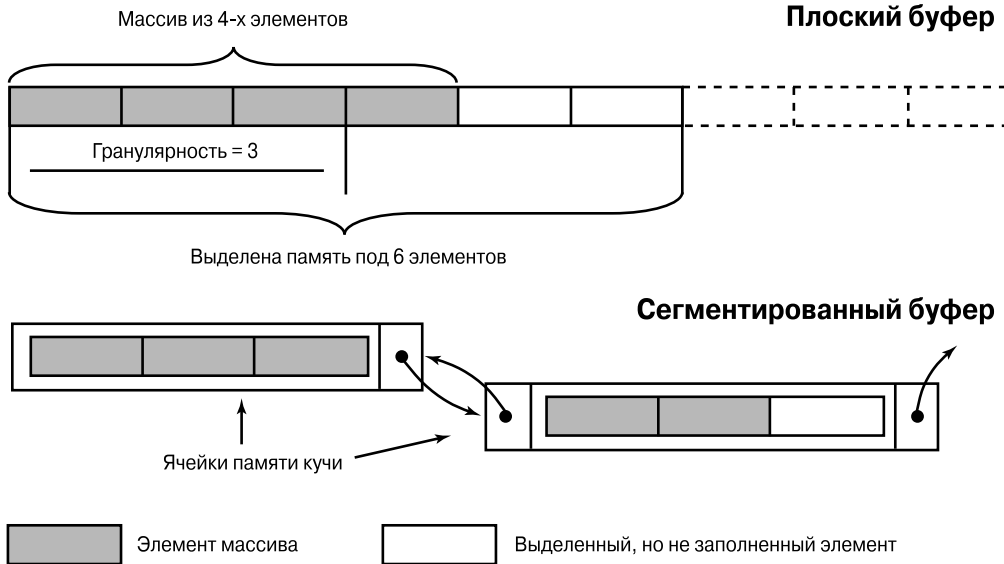


Рис. 5.6. Плоский и сегментированный буферы динамических массивов

Классы, реализующие плоский буфер для хранения данных, позволяют быстрее находить элемент по индексу. Но операции вставки, удаления и добавления элементов в них неэффективны, так как требуют изменения размера буфера или сдвига данных в массиве. Поэтому, если вы планируете часто добавлять или удалять элементы из массива, вам следует предпочесть классы с сегментированным буфером.

Все динамические массивы обладают таким параметром, как **зерно** (granularity), задаваемым в конструкторе класса (см. рис. 5.6). Массив с зерном, равным n , создается пустым, а при добавлении в него первого элемента

увеличивает свой буфер для хранения сразу n элементов. При этом использоваться будет лишь один из них. Впоследствии массив не увеличивает свой буфер до тех пор, пока все выделенные, но неиспользуемые элементы не будут заполнены, после чего вновь выделяется память для n элементов. Таким образом, массив всегда резервирует память для хранения числа элементов, кратного его зерну. Зерно помогает уменьшить количество необходимых операций увеличения буфера при росте числа элементов в массиве. Оптимальное значение зерна выбирается разработчиком в зависимости от конкретной задачи. Слишком маленькое зерно приводит к лишним операциям и трате процессорного времени, слишком большое — к напрасному расходу памяти. На рис. 5.6 показан сегментированный буфер, в одной ячейке которого находится одно зерно. На самом деле это не обязательно — ячейка может хранить столько элементов, сколько в ней поместится, но их количество будет кратно зерну.

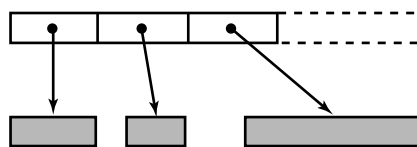
Динамические массивы Symbian C++ различаются также по типу хранящихся в них элементов. Элементы могут быть следующих типов (рис. 5.7).

- Фиксированной длины (например, числа).
- Произвольной длины — элементы, размер которых может меняться. Каждый элемент хранится в собственной ячейке кучи. В буфере же содержатся лишь указатели на такие объекты.
- Упакованные — элементы произвольного размера, находящиеся в буфере массива. В начале каждого элемента имеется поле, хранящее его размер.
- Указатели на объекты C-классов.

Элементы фиксированной длины



Элементы произвольной длины и указатели на объекты C-классов



Упакованные элементы произвольной длины

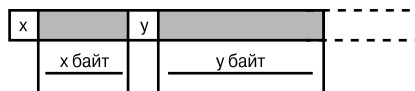


Рис. 5.7. Типы элементов массивов Symbian C++

В Symbian C++ существует восемь C-классов для работы с динамическими массивами (для двух типов буферов и четырех типов элементов). Их обобщенно называют `CArrayX`. Название этих классов можно представить как `CArray<Elem><Buf>`, где `<Elem>` — тип элемента, `<Buf>` — тип буфера.

Параметр `<Elem>` может принимать следующие значения:

- `Fix` — элементы фиксированной длины;
- `Pak` — упакованные элементы произвольной длины;

- Var — элементы произвольной длины;
- Ptr — элементы являются указателями на объекты.

Параметр *<Buf>* может представлять собой либо Flat — для плоского буфера, либо Seg — для сегментированного.

Классы CArrayX являются тонкими шаблонами и принимают название типа элемента во время объявления. Следующий пример демонстрирует создание, заполнение и уничтожение массива беззнаковых целых чисел, а также вычисление среднего арифметического (конечно, это можно было бы сделать и без использования массива).

```
CArrayFixFlat<TUint>* arr_int =
    new (ELeave) CArrayFixFlat<TUint>(11);
// arr_int: Length = 4, Count = 0, Granularity = 11
CleanupStack::PushL(arr_int);

// Инициализация массива случайными числами
for (TInt i = 0; i < 30; i++)
    arr_int->AppendL(Math::Random());

// arr_int: Count = 30, но память выделена для 11*3 = 33
arr_int->Compress(); // Сжимаем буфер массива

TReal sum(0);
for (TInt i = 0; i < arr_int->Count(); i++)
    sum += (*arr_int)[i];

if (arr_int->Count() != 0)
    sum /= arr_int->Count();
// В sum — среднее арифметическое всех элементов

// Уничтожение массива
CleanupStack::PopAndDestroy(arr_int);
```

Для хранения чисел я выбрал класс, оперирующий элементами фиксированной длины в плоском буфере. Массив был инициализирован 30 случайными числами. Для их получения используется метод Random() статического класса Math из файла e32math.h. Обратите внимание, что зерно массива было подобрано не слишком удачно. Если бы мы использовали делитель 30, то избежали бы лишней траты памяти. В идеале, зная заранее, сколько элементов будет в массиве, и что это число не изменится, мы могли бы выбрать зерно 30, и увеличение буфера при инициализации происходило бы лишь один раз. В нашем случае arr_int трижды будет увеличивать свой буфер и зарезервирует место для трех лишних элементов. Для того чтобы освободить выделенную, но не используемую память, во всех классах динамических массивов присутствует метод Compress().

Следующие методы, общие для всех классов `CArrayX`, используются для управления массивом.

- `Count()` — количество элементов в массиве. Зарезервированная для добавления новых элементов память не влияет на это значение.
- `AppendL()` — добавление элемента в массив.
- `InsertL()` — вставка элемента в указанную позицию.
- `ExtendL()` — добавление пустого элемента в массив. Элемент создается конструктором по умолчанию.
- `ExpandL()` — вставка пустого элемента в указанную позицию. Элемент создается конструктором по умолчанию.
- `At()` или оператор `[]` — получение ссылки на элемент с заданным индексом. Будьте внимательны — при изменении состава массива ссылка может утратить актуальность.
- `Delete()` — удаление элемента с заданным индексом.
- `Compress()` — служит для уменьшения размеров буфера за счет памяти, зарезервированной, но не используемой в данный момент.
- `ResizeL()` — изменение размера массива. Если массив уменьшается, то лишние элементы помечаются как неиспользуемые, но размер буфера не изменяется. Если массив увеличивается — к нему добавляются пустые элементы (создаются с помощью конструктора по умолчанию).

Все классы `CArrayX`, кроме `CArrayPtrX`, позволяют получить размер хранящегося по указанному индексу элемента с помощью метода `Length()`. Для `CArrayFixX` метод `Length()` не имеет аргументов, так как размер всех элементов массива одинаков.

Метод `Reset()` позволяет очистить любой динамический массив и освободить память, занимаемую его буфером. При этом в классах, хранящих элементы в ячейках кучи, эта память также освобождается. Он подходит для всех классов, кроме `CArrayPtrX`, содержащего указатели на объекты других классов (чаще всего `C`-классов). Обычно классы `CArrayPtrFlat` и `CArrayPtrSeg` не владеют хранящимися в них объектами. Тем не менее они имеют метод `ResetAndDestroy()`, позволяющий вызвать деструктор у каждого помещенного в массив объекта, а затем освободить память массива.

Поиск элемента в динамическом массиве осуществляется с помощью метода `Find`, принимающего в качестве аргументов ссылку на искомый объект, переменную для хранения результата, а также объект класса `TKeyArrayX`, позволяющий задать правила, по которым сравниваются элементы. Например, следующий код позволяет найти число 100 в целочисленном массиве.

```
TKeyArrayFix key(0, ECmpTUint32); //Сравнение TUint по офсету 0
TInt pos(0); // Здесь будет сохранен результат
if (arr_int->Find(100, key, pos) == KErrNone)
{
```

```

    // В pos — индекс элемента равного 100
}

```

В соответствии с ключом TKeyArrayX может производиться и сортировка массива.

```

TKeyArrayFix key(0, ECmpTUint32);
arr_int->Sort(key); // Сортировка чисел по возрастанию

```

В Symbian C++ определены классы для работы с динамическими массивами дескрипторов: это нейтральные CDesCArrayFlat, CDesCArraySeg и CPtrCArray, соответствующие CDesC16ArrayFlat, CDesC16ArraySeg и CPtrC16Array, а также их 8-битовые аналоги — CDesC8ArrayFlat, CDesC8ArraySeg и CPtrC8Array. Все они происходят от базовых классов семейства CArrayX и используются точно также, за одним исключением: они не являются тонкими шаблонами и не требуют указания типа хранящихся в них элементов при объявлении. Реализация динамических массивов для работы с дескрипторами находится в библиотеке BAFL (Basic Application Framework Library), а их объявление — в заголовочном файле badesca.h.

```

CDesCArrayFlat* arr_des = new (ELeave) CDesCArrayFlat(5);
CleanupStack::PushL(arr_des);

arr_des->AppendL(_L("Test string"));
arr_des->AppendL(_L("Another string"));
arr_des->AppendL(_L("One more string"));

TBuf<30> buf = (*arr_des)[2]; // buf = "One more string"
TKeyCmpText key(ECmpNormal);
arr_des->Sort(key); // Побайтовая сортировка в алфав. порядке
// arr_des = ["Another string", "One more string",
              "Test string"]

TInt pos(0);
// Поиск элемента в массиве
if ( arr_des->Find(buf, pos, key) == KErrNone)
{
    // pos = 1
}

CleanupStack::PopAndDestroy(arr_des);

```

Помимо вышеописанных классов, Symbian C++ предлагает два высокоэффективных R-класса для организации динамических массивов элементов фиксированного размера. Их отличие от семейства CArrayFixX в том, что они не создают объект TPtr8 для доступа к элементам — это повышает производительность. Кроме того, динамические массивы, основанные на R-классах, предлагают безопасные методы добавления элементов, удобные для использования в методах, которые не должны вызывать сброс.

Эффективность подобных массивов достигается за счет некоторых ограничений, накладываемых на элементы, которые могут в них храниться. Их размер не может быть меньше 4 байт и больше 640 байт. Кроме того, он должен быть кратен 4 байтам.

Существует только два R-класса для работы с массивами: `RArray` (аналог `CArrayFixFlat`) и `RPointerArray` (аналог `CArrayPtrFlat`). Их экземпляры часто создаются в стеке либо в качестве члена другого класса. Освобождение занимаемой массивом памяти выполняется с помощью метода `Reset()` или `Close()`. `RPointerArray` имеет метод `ResetAndDestroy()` для вызова деструкторов у хранящихся в нем элементов перед освобождением памяти.

Оба класса являются тонкими шаблонами. На практике класс `RArray` очень часто используется для хранения числовых массивов, поэтому в системе уже определены `RArray<TInt>` и `RArray<TUint>`, предоставляющие более простые методы сортировки и поиска.

```
RArray<TUint> arr_int(11); // Зерно = 11
CleanupClosePushL(arr_int);

// Инициализация массива случайными числами
for (TInt i = 0; i < 30; i++)
    arr_int.AppendL(Math::Random());

arr_int.Compress(); // Сжимаем буфер массива

TReal sum(0);
for (TInt i = 0; i < arr_int.Count(); i++)
    sum += arr_int[i];

if (arr_int.Count() != 0)
    sum /= arr_int.Count();
// В sum — среднее арифметическое всех элементов

arr_int.Sort(); // Сортировка элементов по возрастанию
TInt pos = arr_int.Find(255); // Поиск элемента 255 в массиве

// Уничтожение массива (Вызывается метод arr_int.Close())
CleanupStack::PopAndDestroy(&arr_int);
```

При использовании динамического массива `RPointerArray` следует помещать его в стек очистки с помощью метода `CleanupResetAndDestroyPushL()`, объявленного в файле `\include\mmf\common\mmfcontrollerpluginresolver.h`, либо извлекать его с помощью метода `CleanupStack::Pop()` и вызывать метод `ResetAndDestroy()` вручную.

Поиск и сортировка элементов произвольного типа в R-классах динамических массивов выполняется несколько сложнее. Для этого необходимо реализовать функции для сравнения объектов данного класса (если они не реализованы самим классом) и передать указатель на них при вызове

методов `Sort()` или `Find()` массива. Например, определим следующий класс `TMyPosition` (листинг 5.1).

Листинг 5.1. Класс `TMyPosition`

```
struct TMyPosition
{
    // Класс реализует точку на плоскости
    TInt iX;
    TInt iY;

    inline TMyPosition(const TInt aX, const TInt aY)
    {
        // Конструктор для удобства
        iX = aX;
        iY = aY;
    };
};

TReal GetDist(const TMyPosition &aPos1,
              const TMyPosition &aPos2)
{
    // Вычислим расстояние между точками
    TInt dx = aPos1.iX - aPos2.iX;
    TInt dy = aPos1.iY - aPos2.iY;
    TReal dist(0);
    Math::Sqrt(dist, dx * dx + dy * dy);
    return dist;
}

TBool IsEqualPos(const TMyPosition &aPos1,
                const TMyPosition &aPos2)
{
    // Если расстояние достаточно мало, положим точки равными
    const TInt KCloseEnough = 10;
    return (GetDist(aPos1, aPos2) < KCloseEnough);
}

TInt CompareMyPosition(const TMyPosition &aPos1,
                      const TMyPosition &aPos2)
{
    // Сравнение двух точек
    if (IsEqualPos(aPos1, aPos2))
        return 0; // Равны
    // Иначе сравниваем по координате X (например)
    if (aPos1.iX > aPos2.iX)
        return 1;
    else
        return -1;
}
```

Класс `TMyPosition` содержит информацию о точке на плоскости и объявляет конструктор, удобный для создания объектов такого класса. Размер `TMyPosition` равен 8 байтам, таким образом его объекты могут храниться в динамическом массиве `RArray`. Мы определили вспомогательную функцию `GetDist()`, возвращающую расстояние между двумя объектами `TMyPosition`, а также методы `IsEqualPos()` и `CompareMyPosition()` для определения равенства и сравнения таких объектов. При этом мы положили равными все экземпляры класса `TMyPosition`, расстояние между которыми не превышает значения `KCloseEnough`. Следующий пример демонстрирует использование массива таких объектов.

```
RArray<TMyPosition> arr(5); // Зерно = 5
CleanupClosePushL(arr);

TMyPosition a(100, 100); // Выберем 3 точки на плоскости
TMyPosition b(6, 7);
TMyPosition c(2, 3); // Две из них довольно близки

arr.Append(a); // Инициализация
arr.Append(b);
arr.Append(c);
arr.Compress(); // Сжимаем память буфера
// arr = [(100, 100), (6,7), (2,3)]

TLinearOrder<TMyPosition> order(&CompareMyPosition);
arr.Sort(order); // Сортировка по нашему методу
/*
 * arr = [(6,7), (2,3), (100, 100)]
 * (6, 7) и (2, 3) не изменили взаимное положение,
 * так как достаточно близки (расстояние < KCloseEnough)
 */
TMyPosition d(101, 101); // Такой точки в массиве нет
TIdentityRelation<TMyPosition> ident(&IsEqualPos);
TInt pos = arr.Find(d, ident); // Поиск с учетом близости
// pos = 2, т.к. (101, 101) достаточно близка к (100, 100)
CleanupStack::PopAndDestroy(&arr);
```

В заключение необходимо отметить опасность использования копирующего оператора `=` в классах `RArray` и `RPointerArray`. Как вы помните, R-классы копируются побитово, и полученный путем копирования объект будет содержать хендл на тот же массив что и оригинал. Но при редактировании массива он может изменить свое местоположение в памяти (так как это плоский буфер), и эти изменения будут отражены только в том экземпляре R-класса, через который производились. Проблемы также могут возникнуть при освобождении занимаемой массивом памяти.

Благодаря их высокой производительности и несмотря на многочисленные нюансы в использовании, классы `RArray` и `RPointerArray` рекомендуется

применять вместо классов `CArrayFlatX` везде, где это возможно, например, при работе с числовыми массивами и простыми T-классами.

Подготовка к сертификации ASD

- Знание назначения и характеристик семейств динамических массивов `CArrayX` и `RArray`.
- Знание отличий динамических массивов Symbian OS по типу используемого буфера (плоский или сегментированный), типу хранимых объектов, их размеру (фиксированного или произвольного размера) и владению объектами.
- Знание условий, при которых следует предпочесть массив с сегментированным буфером массиву с плоским буфером.
- Понимание значения зерна массива.
- Умение выбирать подходящий размер зерна массива исходя из того, как он будет использоваться.
- Знание условий, при которых следует использовать массивы семейства `RArrayX` вместо `CArrayX`, а также исключений, при которых классы `CArrayX` применять лучше.
- Знание того, как выполняется сортировка и поиск в динамических массивах.
- Знание того, что `RArray`, `RPointerArray` и все классы семейства `CArrayX` могут быть отсортированы. Знание того, что производительность массивов `RArray`, `RPointerArray` выше.

Массивы фиксированного размера

Для создания массивов фиксированного размера разработчик может воспользоваться стандартными средствами C++. В дополнение к ним в Symbian C++ определен класс-шаблон `TFixedArray`, принимающий при объявлении тип хранящихся в массиве элементов и его размерность. Следующий пример демонстрирует создание такого массива.

```
TFixedArray<TInt, 5> arr;
for (TInt i = 0; i < 5; i++)
    arr[i] = i;
// Либо
TInt data[] = { 0, 1, 2, 3, 4 };
TFixedArray<TInt, 5> arr(&data[0], 5);
```

Класс `TFixedArray` предоставляет следующие методы.

- `At()` или оператор `[]` — получение элемента по индексу. Если индекс неверен в `DEBUG`-сборках будет вызываться паника `USER 133`.
- `Count()` — размерность массива.
- `Length()` — размер элемента массива.

- `Copy()` — копирование элементов в начало массива из указанной области памяти.
- `Begin()` — указатель на первый элемент массива.
- `End()` — указатель на последний элемент массива.
- `Reset()` — обнуление всех элементов.
- `DeleteAll()` — вызов оператора `delete` для всех элементов в массиве. Полезен, если элементами являются указатели на C-классы. После выполнения рекомендуется вызывать метод `Reset()`.

Так как класс `TFixedArray` производит проверку индекса при обращении к элементам, его рекомендуется использовать вместо традиционных массивов C++.

Подготовка к сертификации ASD

- Понимание, почему при работе с фиксированными массивами класс `TFixedArray` предпочтительнее массивов C++.
-

Активные объекты

Активные объекты (active objects, или АО) — традиционно трудная для изучения, но чрезвычайно широко применяющаяся концепция Symbian C++, представляющая собой паттерн программирования для реализации асинхронных вызовов и событийно-ориентированного управления приложением. При работе вы можете обнаружить операции, выполнение которых связано с существенной задержкой. Например, чтение большого файла, отправка SMS или выполнение звонка происходят отнюдь не мгновенно. Для того чтобы, обратившись к этим функциям, выполнение интерактивного приложения не останавливалось в ожидании их завершения, и приложение продолжало реагировать на пользовательский ввод (например, нажатие кнопки “Отмена”), их необходимо реализовать асинхронно. В современных операционных системах, в том числе и Symbian OS, асинхронность достигается за счет **вытесняющей многопоточности** (preemptive multithreading). Но разрабатывать приложения, содержащие несколько потоков выполнения, довольно сложно: возникают проблемы синхронизации их исполнения и конкурирующего доступа к ресурсам. Это вынуждает программиста иметь дело с мьютексами, семафорами и прочими примитивами. Наконец, переключение между потоками является довольно ресурсоемкой операцией.

Symbian C++ предлагает более простой (а значит — и более надежный), изящный и эффективный способ реализации асинхронных вызовов — *активные объекты*. Они оказались настолько удобными, что практически полностью вытеснили прямую работу с потоками не только в сторонних приложениях, но и в подсистемах различного уровня самой операционной системы. Большинство

API Symbian OS наряду с синхронными методами предоставляет их асинхронный вариант для использования в активных объектах.

Для начала попытаемся разобраться с принципами работы активных объектов в несколько упрощенном виде (рис. 5.8). Активный объект предназначен для запуска асинхронной функции и получения результата ее работы. Асинхронная функция отличается от синхронной тем, что лишь запускает выполнение какой-то задачи (запроса), но не дожидается ее завершения. Например, асинхронной функцией может быть метод, запрашивающий код следующей нажатой пользователем кнопки. Разработчик не может предположить, когда ее нажмут, и нажмут ли вообще. Таким образом, асинхронная функция возвращает управление в программу сразу же после вызова, но не может вернуть результат, так как запущенный в ней запрос еще не завершен. Программа еще не получила результата выполнения асинхронного запроса, но она может продолжать свою работу: например, запускать другие асинхронные функции, отображать ход обработки данных в диалоговом окне или вести иную деятельность, благодаря которой и может называться интерактивной. В тот момент, когда асинхронный запрос завершится, в запустившем его активном объекте будет вызван специальный метод-обработчик. В этот метод будет передан результат выполнения задачи, запущенной данной асинхронной функцией, — например код нажатой клавиши.

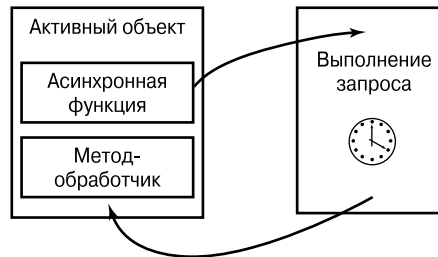


Рис. 5.8. Выполнение асинхронной функции

Как видите, на данный момент все вышеописанное очень похоже на обычную регистрацию на получение уведомления о событии по callback-методу, но это не так. На самом деле это несколько более сложный процесс, позволяющий эмулировать невытесняющую псевдомультитзадачность. В одном потоке может быть создано множество активных объектов. Если бы каждый активный объект требовал для работы отдельный поток, мы бы не получили ни упрощения, ни повышения производительности при событийно-ориентированном программировании. Возможна ситуация, при которой активный объект обрабатывает завершение асинхронного запроса достаточно долго, для того чтобы в течение этого времени завершились еще несколько асинхронных запросов каких-либо других активных объектов из того же потока. Таким образом, нам необходим механизм, гарантирующий, что методы-обработчики этих активных объектов будут вызваны, и определяющий порядок их вызова. Таким механизмом является **планировщик** (active scheduler).

В каждом потоке приложения, создающем хотя бы один активный объект, должен быть только один планировщик (рис. 5.9). Все активные объекты должны регистрироваться в нем в момент создания. Планировщик реализует бесконечный цикл ожидания завершения асинхронных запросов. Как только асинхронный запрос завершает свое выполнение (удачно или с ошибкой), в потоке связанного с ним активного объекта увеличивается значение специального семафора. Это значение постоянно проверяется планировщиком в бесконечном цикле ожидания. Даже если в момент срабатывания семафора исполнялся метод-обработчик одного из активных объектов, рано или поздно он завершится, и планировщик получит управление. Обнаружив ненулевое значение семафора, планировщик ищет в своем списке зарегистрированных активных объектов тот, запрос которого был исполнен, и вызывает его метод-обработчик. Каждый активный объект имеет некоторый приоритет, и список планировщика отсортирован по этому значению. Таким образом, среди всех готовых к обработке активных объектов всегда вызывается метод-обработчик объекта с наивысшим приоритетом.

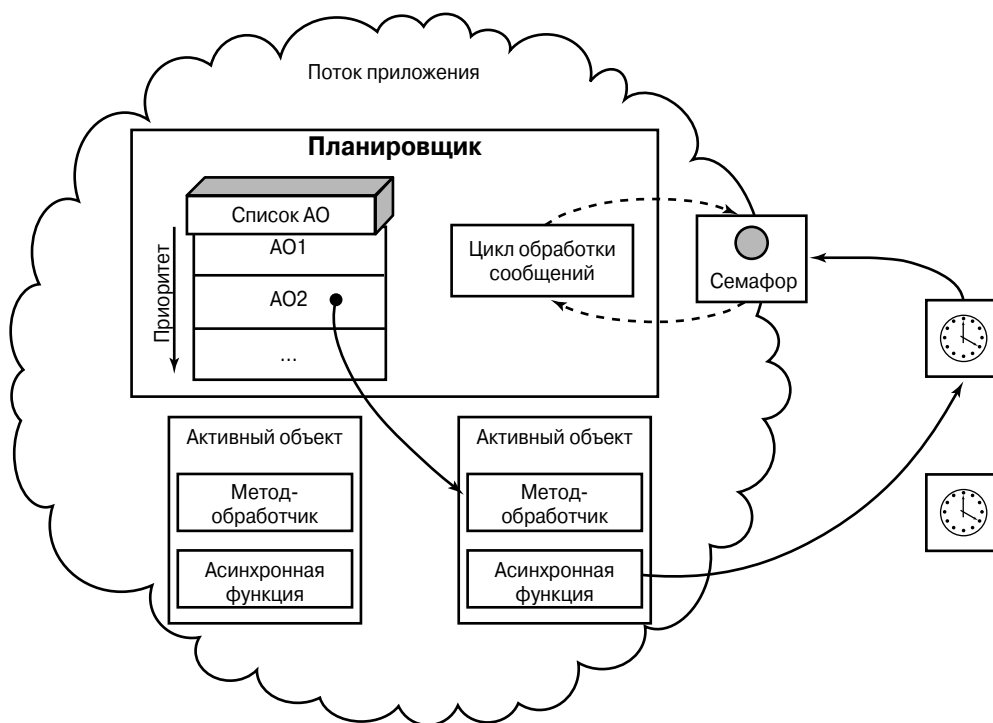


Рис. 5.9. Планировщик активных объектов

Планировщик позволяет реализовать невытесняющую псевдомультитасочность. “Псевдо” — потому что все активные объекты планировщика находятся в одном потоке с ним. Из-за этого же “псевдо” планировщик не может приостановить выполнение одного активного объекта и вызвать метод другого активного

объекта с большим приоритетом — поэтому псевдомультитзадачность “невывес-няющая” (nonpreemptive). Это может привести к некоторым проблемам. Например, если в списке планировщика есть несколько активных объектов с высоким приоритетом и несколько с низким, и при этом асинхронные запросы первых завершаются сравнительно быстро, то до метода-обработчика объектов с низким приоритетом очередь может просто не доходить, поэтому с приоритетами активных объектов следует обращаться аккуратно.

Для полноты картины необходимо сделать несколько замечаний по поводу самих асинхронных функций. Предоставляющие их классы называют **поставщиками асинхронного сервиса** (async. service provider). Реализация выполняемых асинхронно запросов в большинстве случаев находится за пределами потока или процесса, вызывающего асинхронную функцию. Чаще всего это один из служебных серверов Symbian OS. Поэтому поставщиками асинхронного сервиса обычно выступают C-классы и используемые для доступа к ним R-классы. В качестве аргумента асинхронной функции по ссылке передается объект небольшого T-класса (TRequestStatus), являющийся членом активного объекта. Объект класса TRequestStatus содержит только одно поле данных типа TInt, хранящее результат асинхронного запроса. Наличие в функции аргумента этого типа свидетельствует о том, что она выполняется асинхронно и предназначена для использования в активных объектах. Исключением из этого правила является ряд служебных методов классов User и RThread. Поставщик асинхронного сервиса чаще всего предоставляет сразу несколько асинхронных функций для выполнения различных запросов, а также методы для их отмены.

Итак, надеюсь, вы разобрались с принципом работы активных объектов и готовы перейти к их реализации. Этот механизм может показаться довольно сложным, но на самом деле использование активных объектов является тривиальной задачей для разработчика на языке Symbian C++. Вы так часто будете встречаться с ними на практике, что быстро освоите все тонкости работы. А овладев некоторыми хитростями IDE Carbide.c++, вы сможете создавать классы активных объектов за несколько минут.

Планировщик активных объектов реализован в классе CActiveScheduler. Для работы с активными объектами в потоке должен быть создан и установлен один экземпляр этого класса.

```
CActiveScheduler* scheduler = new (ELeave) CActiveScheduler();
CActiveScheduler::Install(scheduler);
```

Планировщик должен быть установлен в потоке до создания активных объектов, так как они в свою очередь обязаны в нем зарегистрироваться. Поэтому инициализацию планировщика выполняют как можно раньше, часто — в точке входа программы (для главного потока). В GUI-приложениях планировщик в главном потоке создается системой автоматически, так как он необходим для работы подсистемы Avkon. Но во всех порожденных потоках планировщик всегда устанавливается вручную.

После установки планировщика работа с ним ведется с помощью статических методов класса `CActiveScheduler`. Это избавляет разработчика от необходимости передачи указателя на него в различные функции (например, конструкторы активных объектов). Не следует устанавливать два и более планировщика в одном потоке — в этом случае использоваться будет только один из них. Получить указатель на текущий планировщик можно с помощью статического метода `Current()`.

Класс `CActiveScheduler` предоставляет статический метод `Add()` для регистрации активных объектов, а также статические методы `Start()`, `Stop()` и `Halt()` для запуска и остановки цикла ожидания завершения асинхронных запросов.

После вызова метода `Start()` последовательный ход выполнения приложения прекращается. С этого момента работать могут только обработчики асинхронных запросов и обработчики ошибок активных объектов. Планировщик выйдет из цикла ожидания только при вызове методов `Stop()` или `Halt()`. Если на момент вызова `Start()` в планировщике не было зарегистрировано ни одного активного объекта или ни один из них не вызвал асинхронный запрос, то поток зависнет. Таким образом, работа событийно-управляемого приложения имеет следующий вид.

```
// Стек очистки уже должен быть инициализирован
// Создание планировщика
CAActiveScheduler* scheduler = new (ELeave) CActiveScheduler();
CleanupStack::PushL(scheduler);
// Установка планировщика в поток
CAActiveScheduler::Install(scheduler);
MainL(); // Создание активных объектов, запуск асинхр. функций
// Запуск цикла ожидания, выполнение обработчиков АО
CAActiveScheduler::Start();
// Завершение работы приложения
CleanupStack::PopAndDestroy();
```

Для того чтобы создать активный объект, необходимо объявить `C`-класс, унаследованный от `CActive` (он в свою очередь порожден от `CBase`). Класс `CActive` содержит несколько чистых виртуальных функций и напрямую использоваться не может. От `CActive` ваш класс унаследует следующее.

- Методы `Priority()` и `SetPriority()`, служащие для получения и изменения приоритета активного объекта в процессе выполнения программы. Приоритет является целым числом и обычно принимает одно из значений перечисления `CActive::TPriority`. Приоритет также необходимо указывать в конструкторе класса `CActive`.
- Метод `IsAdded()` — проверяет, зарегистрирован ли ваш объект в текущем планировщике. Используется редко, так как регистрацию принято выполнять в конструкторе активного объекта.

- Метод `Deque()` — удаляет активный объект из списка планировщика и отменяет текущий асинхронный запрос (если он был запущен).
- Метод `IsActive()` возвращает булево значение, сигнализирующее о том, ожидает ли активный объект завершения какого-либо асинхронного запроса. Всегда проверяйте его результат, прежде чем вызвать асинхронную функцию. Если вы, не дождавшись результатов первого асинхронного вызова, запустите еще один — это приведет к панике. Подобную ошибку называют потерянным сигналом (*stray signal*), так как планировщик не может найти активный объект, обработчик которого должен быть вызван для его обработки. Если по какой-то причине вам необходимо использовать активный объект с незавершенным асинхронным запросом — сначала отмените его.
- Член `iStatus` типа `TRequestStatus`. Этот объект следует передавать в асинхронную функцию. В момент ее выполнения он принимает значение `KRequestPending`. После завершения асинхронного запроса он может содержать либо один из стандартных кодов ошибки, либо иное оговоренное в SDK числовое значение. Значение `iStatus` часто проверяется в обработчике активного объекта, но никогда не меняется вручную.
- Метод `SetActive()` устанавливает состояние ожидания в активном объекте. Он не имеет аргументов и должен вызываться сразу после запуска нового асинхронного запроса, таким образом вы с помощью проверки результата метода `IsActive()` сможете обезопасить себя от его повторного запуска. Состояние ожидания автоматически сбрасывается планировщиком перед вызовом обработчика активного объекта либо самим активным объектом при отмене текущего асинхронного запроса.
- Чистый виртуальный метод `RunL()` — он должен быть реализован порожденным от `CActive` классом. Это и есть многократно упомянутый обработчик асинхронных вызовов. Он не имеет аргументов — данные в него передаются через члены класса или созданные в куче переменные. В методах `RunL` различных активных объектов реализуется вся управляющая логика событийно-ориентированного приложения. Обычно в нем содержатся многочисленные проверки значения `iStatus` (часто с помощью оператора `switch`), на основании которых происходит обработка результата асинхронного запроса, а также может вызваться новая асинхронная функция.
- Виртуальный метод `RunError()`. Так как обработчик `RunL()` активного объекта вызывается планировщиком, то вы не можете отловить возникший в нем сброс. Сброс в методе `RunL()` перехватывается планировщиком, после чего его код передается в качестве аргумента методу `RunError()` вашего класса — в нем вы можете обработать возникшее исключение. Результатом метода `RunError` также является код ошибки. Если он отличен от `KErrNone`, планировщик сочтет, что устранить ошибку вы не смогли, и обработает ее сам. Чаще всего это приводит к панике и аварийному завершению работы приложения. Данный метод не является чистым виртуальным — вы можете не реализовывать его в своем классе. В этом случае будет использована

реализация `RunError()` класса `CActive`, при которой аргумент метода без какой-либо обработки возвращается в качестве его результата. Таким образом, если вы хотите подавить возникший в `RunL()` сброс, вам необходимо реализовать свой вариант метода `RunError()` в классе активного объекта.

- Чистый виртуальный метод `DoCancel()` — в него вы должны поместить код для отмены асинхронного запроса. Этот метод никогда не вызывается напрямую, его вызывает метод `Cancel()`, о котором мы поговорим чуть позже. Нет необходимости размещать в `DoCancel()` проверку `IsActive()` — если активный объект не выполняет асинхронный запрос, метод `DoCancel()` никогда не будет вызван.
- Метод `Cancel()` — используется для отмены текущего асинхронного запроса. В случае, если активный объект не ожидает завершения какого-либо запроса, метод `Cancel()` ничего не предпринимает. Это позволяет использовать его без проверки с помощью метода `IsActive()`. Если же асинхронный запрос был запущен и еще не завершился, то метод `Cancel()` сначала вызывает метод `DoCancel()`, затем дожидается завершения запроса и, наконец, сбрасывает состояние ожидания активного объекта в `EFalse`. Метод `Cancel()` всегда следует помещать в деструктор активного объекта до уничтожения его членов. Это гарантирует, что планировщик не получит “потерянный сигнал”.
- Деструктор `CActive` — автоматически удаляет объект из списка планировщика, поэтому в использовании метода `Deque()` нет необходимости.

Как видите, существует множество правил для использования активного объекта: некоторые методы должны быть обязательно реализованы, некоторые не должны вызываться напрямую, в одних методах необходимо добавлять определенные проверки, в других они излишни и т.д. Все это довольно сложно запомнить. Поэтому я рекомендую вам использовать для создания активных объектов шаблон `Carbide.c++`. С его помощью генерируются заголовочный файл и файл исходного кода типового активного объекта с заданным названием, содержащего обращение к системному таймеру. Его несложно переделать для использования с любой другой асинхронной функцией или даже группой таких функций.

Для того чтобы создать новый класс активного объекта, необходимо воспользоваться командой меню **File⇒New⇒Symbian OS C++ Class**. Она запускает небольшой мастер — в нем вы должны указать проект, в который необходимо добавить новый класс, его название (без префикса “C”), имена файлов для хранения и шаблон класса. В качестве шаблона необходимо выбрать **Active Object Class**. После этого в папке `\src\` вашего проекта появится CPP-файл с кодом нового класса, а в папке `\inc\` — его заголовочный файл. Новый файл исходного кода необходимо добавить в MMP-файл проекта с помощью ключевого слова `SOURCE`. Обычно `Carbide.c++` самостоятельно обнаруживает изменения в папке `\src\` и предлагает внести автоматические изменения в MMP-файл — соглашайтесь.

Для примера рассмотрим класс активного объекта, сгенерированного системой для названия `MyActiveClass` (листинг 5.2 — здесь комментарии были переведены на русский язык).

Листинг 5.2. Файл `MyActiveClass.h`

```
#ifndef MYACTIVECLASS_H
#define MYACTIVECLASS_H

#include <e32base.h> // Для CActive, реализованного в euser.lib
#include <e32std.h>  // Для RTimer, реализованного в euser.lib

class CMyActiveClass : public CActive
{
public:
    // Отмена и уничтожение
    ~CMyActiveClass();

    // Двухфазный конструктор.
    static CMyActiveClass* NewL();

    // Двухфазный конструктор.
    static CMyActiveClass* NewLC();

public:
    // Новые функции
    // Функция для выполнения первого запроса
    void StartL(TTimeIntervalMicroSeconds32 aDelay);

private:
    // Конструктор C++
    CMyActiveClass();

    // Конструктор второй фазы
    void ConstructL();

private:
    // Наследованы из CActive
    // Обработчик
    void RunL();

    // Как отменить запрос
    void DoCancel();

    // Перепишите, чтобы обрабатывать сбросы в RunL().
    // Текущая реализация позволяет планировщику вызывать панику
    TInt RunError(TInt aError);
};
```

```
private:
    enum TMyActiveClassState
    {
        EUninitialized,    // Не инициализирован
        EInitialized,      // Инициализирован
        EError              // Ошибка
    };

Private:
    TInt iState;          // Состояние активного объекта
    RTimer iTimer;        // Предоставляет асинхронный сервис тайминга
};

#endif // MYACTIVECLASS_H
```

В листинге 5.2 отображен заголовочный файл полученного класса CMyActiveClass. Как видите, он наследован от CActive и реализует его чистые виртуальные методы RunL() и DoCancel(). Класс CMyActiveClass также содержит переопределенный метод RunError(), позволяющий разработчику обрабатывать произошедшие в методе RunL() сбросы.

В качестве поставщика асинхронного сервиса в данном случае выступает класс RTimer, объявленный в файле e32std.h. Класс RTimer — один из наиболее простых таймеров Symbian C++ и более подробно будет рассмотрен нами в следующей главе. Он предоставляет ряд асинхронных функций для регистрации на получение уведомления по истечению какого-то периода времени. В данном случае поставщик асинхронного сервиса является членом активного объекта — это распространенная практика. Обычно активный объект не запускает асинхронный запрос сразу же после своего создания, а предоставляет для этого отдельный метод. Таким методом здесь является StartL().

Класс CMyActiveClass, как и многие другие C-классы, содержит двухфазный конструктор, позволяющий безопасно инициализировать члены класса в методе ConstructL(). Наконец, в активном объекте определен тип-перечисление TMyActiveClassState и член класса этого типа — iState. Этот объект позволяет задавать в активном объекте различные состояния и на их основе определять его поведение. В листинге 5.3 приведен текст файла MyActiveClass.cpp, содержащего реализацию объявленных в классе CMyActiveClass методов.

Листинг 5.3. Файл MyActiveClass.cpp

```
#include "MyActiveClass.h"

CMyActiveClass::CMyActiveClass() :
    CActive(EPriorityStandard) // Стандартный приоритет
{
}
```

```

CMyActiveClass* CMyActiveClass::NewLC()
{
    CMyActiveClass* self = new (ELeave) CMyActiveClass();
    CleanupStack::PushL(self);
    self->ConstructL();
    return self;
}

CMyActiveClass* CMyActiveClass::NewL()
{
    CMyActiveClass* self = CMyActiveClass::NewLC();
    CleanupStack::Pop(); // self;
    return self;
}

void CMyActiveClass::ConstructL()
{
    // Инициализация таймера
    User::LeaveIfError(iTimer.CreateLocal());
    CActiveScheduler::Add(this); // Добавление в планировщик
}

CMyActiveClass::~CMyActiveClass()
{
    Cancel(); // Отмена любого запроса, если он выполняется
    iTimer.Close(); // Уничтожение объекта RTimer
    // Удаление различных объектов, если есть
}

void CMyActiveClass::DoCancel()
{
    iTimer.Cancel();
}

void CMyActiveClass::StartL(TTimeIntervalMicroSeconds32 aDelay)
{
    Cancel(); // Отмена любого запроса, на всякий случай
    iState = EUninitialized;
    iTimer.After(iStatus, aDelay); // Установка задержки таймера
    SetActive(); // Говорим планировщику, что запрос активен
}

void CMyActiveClass::RunL()
{
    if (iState == EUninitialized)
    {
        <...> // Делаем что-нибудь, когда RunL() вызван первый раз
        iState = EInitialized;
    }
}

```

```

    }
    else if (iState != EError)
    {
        <...> // Делаем что-нибудь
    }
    iTimer.After(iStatus, 1000000); // Установка таймера на 1 с.
    SetActive(); // Говорим планировщику, что запрос активен
}

TInt CMyActiveClass::RunError(TInt aError)
{
    return aError;
}

```

В конструкторе `CMyActiveClass` вызывается конструктор его родителя `CActive`, в который передается значение приоритета активного объекта. По умолчанию таким значением является `EPriorityStandard` — оно подходит для большинства случаев, и его не следует менять на что-либо иное, не имея на то веской причины. Не следует также думать, что все активные объекты в вашей программе должны иметь разный приоритет — это не так. В качестве величины приоритета чаще всего используют одно из следующих значений перечисления `CActive::TPriority`:

- `EPriorityIdle` — низший приоритет, подходящий для активных объектов представляющих обработку асинхронных процессов в фоновом режиме;
- `EPriorityLow` — низкий приоритет;
- `EPriorityStandard` — обычный приоритет;
- `EPriorityUserInput` — высокий приоритет, подходящий для активных объектов, осуществляющих обработку пользовательского ввода;
- `EPriorityHigh` — наивысший приоритет.

На самом деле в качестве значения приоритета можно передать любое число. Например, указав 25, вы получите приоритет больший, чем `EPriorityHigh` (20). Но это бессмысленно, так как приоритет активного объекта влияет лишь на его порядок в списке планировщика. Поэтому ситуация, при которой все активные объекты потока имеют приоритет `EPriorityHigh`, ничем не отличается от той, в которой все они имеют приоритет `EPriorityIdle`. Приоритет активного объекта — способ выделить его из множества других активных объектов потока. Пяти вышеприведенных значений для этого вполне хватит.

В методе `ConstructL()`, являющимся второй фазой двухфазного конструктора C-класса, выполняется инициализация таймера. Сразу после этого экземпляр активного объекта регистрируется в планировщике.

```
CActiveScheduler::Add(this);
```

Данный код не способен привести к сбросу и мог бы быть помещен в конструктор класса. Тем не менее его предпочитают помещать в метод `ConstructL()`,

причем — в последнюю очередь, после инициализации. Я полагаю, что это позволяет не беспокоить планировщик в том случае, если создание активного объекта не удастся и в двухфазном конструкторе произойдет сброс. Иногда в метод `ConstructL()` также помещают код вызова асинхронной функции. В этом случае вы не должны забывать, что регистрация активного объекта в планировщике должна обязательно проводиться *до запуска* асинхронного запроса.

Обратите внимание на деструктор и метод `DoCancel()` нашего класса. В деструкторе перед освобождением ресурсов (в нашем случае таймер закрывает сессию к системному сервису) вызывается метод `Cancel()` — это обязательное условие уничтожения активных объектов. В том случае, если активный объект ожидает завершения асинхронного запроса, метод `Cancel()` вызовет метод `DoCancel()`, в котором вы должны обратиться к поставщику асинхронного сервиса и отменить выполнение запроса. Класс `RTimer` предоставляет несколько асинхронных функций, но один метод `Cancel()` для отмены выполнения любого асинхронного запроса. После его вызова запрос будет немедленно завершен, а `iStatus` получит значение `KErrCancel`.

Метод `StartL()` позволяет запустить первый асинхронный запрос. Этим запросом является `RTimer::After()` — он завершается через указанный промежуток времени. Обратите внимание, что перед запуском асинхронной функции вызывается метод `Cancel()`. Это гарантирует, что активный объект не запустит асинхронный запрос до завершения предыдущего. Вместо метода `Cancel()` вы можете проверять значение `IsActive()` и, например, возвращать код ошибки из `StartL()`. Сразу после запуска асинхронного метода вызывается метод `SetActive()`, переводящий активный объект в состояние ожидания.

После вызова метода `StartL()` асинхронный запрос завершится через `aDelay` микросекунд, и планировщик вызовет метод `RunL()`. В методе `RunL()` происходит запуск очередного асинхронного запроса, но уже с фиксированным аргументом — 1 секунда.

В методе `StartL()` состояние объекта устанавливается в `EUninitialized`. Это позволяет определить в обработчике `RunL()`, что он вызван после запроса, запущенного из `StartL()`, а не из самого `RunL()`. Механизм состояний используется в активных объектах очень часто, особенно если они обращаются к различным асинхронным функциям (или поставщикам асинхронных сервисов).

Наконец, метод `RunError()`, реализованный в `CMyActiveClass`, позволяет нам перехватить возникшие в `RunL()` сбросы. Но для этого необходимо наполнить его своими проверками, так как в данный момент он ничем не отличается от метода `RunError()` класса `CActive`. Например, если вы хотите подавлять все без исключения сбросы и не желаете их как-либо обрабатывать, просто замените `return aError` на `return KErrNone`.

Как видите, активный объект, созданный для нас средой `Carbide.c++`, вполне годится для выполнения каких-то периодических операций. Впрочем, его можно легко переделать для использования любого другого поставщика асинхронных сервисов.

Использовать объект класса `CMyActiveClass` можно следующим образом.

```
CActiveScheduler* scheduler = new (ELeave) CActiveScheduler();
CleanupStack::PushL(scheduler);
CActiveScheduler::Install(scheduler);

CMyActiveClass* aol = CMyActiveClass::NewLC();
// Аргумент равен 10 секундам
aol->StartL(TTimeIntervalMicroSeconds32(100000000));
CActiveScheduler::Start();
// Удаляем aol и scheduler
CleanupStack::PopAndDestroy(2, scheduler);
```

В данном примере мы создаем один экземпляр активного объекта класса `CMyActiveClass` и сразу же запускаем его асинхронную функцию, после чего начинает свое выполнение цикл ожидания планировщика. Если этот код пошагово выполнять в отладчике `Carbide.c++`, то вы увидите, что после создания объекта `aol` происходит вызов и возвращение из метода `StartL()` (асинхронная функция `iTimer.After()` завершается мгновенно). Но когда программа запустит метод `CActiveScheduler::Start()`, последовательный ход ее выполнения будет остановлен. Через 10 секунд (с таким аргументом был вызван метод `StartL()`) вы внезапно окажитесь в методе `RunL()` класса `CMyActiveClass`. Затем с помощью отладчика вы сможете последовательно пройти по всем его операторам, но, дойдя до конца, последовательный ход выполнения программы будет прерван вновь. И уже через 1 секунду вы снова окажитесь в начале метода `RunL`, так как завершится второй асинхронный запрос. Это будет продолжаться до бесконечности, пока программа не будет остановлена системой. Если же вы в какой-то момент (например, через N срабатываний таймера) вместо очередного асинхронного запроса вызовете в `RunL()` метод `CActiveScheduler::Stop()`, то по завершению `RunL()` отладчик продемонстрирует вам, что `CActiveScheduler::Start()` только что вернул управление, и выполнение программы продолжается со следующего за ним оператора. Отладчик не покажет вам ход выполнения процессов, происходящих в планировщике, — в SDK нет исходного кода этого класса.

С помощью активных объектов можно разбить длительно выполняющуюся операцию на этапы. Это позволяет периодически предоставлять возможность планировщику обработать завершение других асинхронных запросов, некоторые из них могут оказаться более приоритетными (пользовательский ввод). Для этого трюка нам не потребуется ни таймер, ни какой-либо другой поставщик асинхронных сервисов. Рассмотрим подробнее пример, представленный в листингах 5.4 и 5.5.

Листинг 5.4. Объявление класса `CLongTask`

```
#include <e32base.h> // Для CActive, реализованного в euser.lib
class CLongTask : public CActive
```

```

{
public:
    // Отмена и уничтожение
    ~CLongTask();

public:
    // Запуск обработки
    void StartL();

    // Конструктор C++
    CLongTask();

private:
    void RunL(); // Обработчик

    void DoCancel(); // Отмена

private:
    // Состояние процесса: выполняется, завершен и отменен
    enum TLongTaskState
    {
        EWorking, EFinished, ECancel
    };
private:
    TLongTaskState iState;
};

```

Листинг 5.5. Реализация класса CLongTask

```

#include "LongTask.h"

CLongTask::CLongTask() : CActive(EPriorityStandard)
{
    CActiveScheduler::Add(this); // Регистрация в планировщике
}

CLongTask::~~CLongTask()
{
    Cancel();
}

void CLongTask::StartL()
{
    Cancel();

    iState = EWorking; // Начинаем поэтапное выполнение процесса

    // Уведомляем планировщик о вызове асинхронной функции

```

```

    SetActive();

    // Самостоятельно сигнализируем о ее завершении
    TRequestStatus* status = &iStatus;
    User::RequestComplete(status, KErrNone);
}

void CLongTask::RunL()
{
    if (iState == EWorking)
    {
        /*
         * Очередной шаг выполнения какого-либо процесса
         * Для завершения присвойте iState значение Efinished
         */

        // Уведомляем планировщик о вызове асинхронной функции
        SetActive();
        // Самостоятельно сигнализируем о ее завершении
        TRequestStatus* status = &iStatus;
        User::RequestComplete(status, KErrNone);
    }
    else
    {
        // Поэтапное выполнение процесса завершено
    }
}

void CLongTask::DoCancel()
{
    {
        iState = ECancel;
        TRequestStatus* status = &iStatus;
        User::RequestComplete(status, KErrCancel);
    }
}

```

В данном примере мы избавились от двухфазного конструктора, так как не выполняем никакой инициализации, способной привести к сбросу. Регистрация активного объекта в планировщике перенесена в C++-конструктор класса. При решении конкретной задачи необходимость в двухфазном конструкторе может возникнуть вновь.

Обратите внимание на метод `StartL()`. В нем нет вызова асинхронной функции — он имитирует этот вызов. Сначала объект помечается как ожидающий результатов асинхронного вызова с помощью метода `SetActive()`. Затем активный объект сам же и имитирует его завершение с помощью метода `RequestComplete()` статического класса `User`. Данный метод присваивает объекту `TRequestStatus` значение второго аргумента и сигнализирует о завершении асинхронного запроса с помощью семафора текущего потока. Таким

образом, как только программа перейдет к выполнению цикла обработки сообщений планировщика, он проверит семафор потока и обнаружит сигнал о завершении асинхронного метода. После этого планировщик осуществит выбор следующего активного объекта, обработчик которого следует запустить. Им может оказаться другой объект, чей запрос также завершился, но, в конечном итоге, управление вернется к экземпляру `CLongTask`. В методе `RunL()` завершение асинхронного вызова имитируется вновь. Таким образом, объект класса `CLongTask`, не прибегая к методам поставщиков асинхронных сервисов, сам периодически провоцирует вызов своего обработчика и в нем производит очередной этап выполнения какой-либо задачи. Такой прием позволяет реализовать длительно выполняющуюся операцию, не блокируя прочие активные объекты потока.

Метод `RequestComplete()` обычно используется поставщиками асинхронных сервисов для извещения о завершении выполнения вызванной активным объектом операции. В случае если поставщик и активный объект находятся в разных потоках, применяется метод `RequestComplete()` класса `RThread`. Более подробно с классом `RThread` и работой с потоками мы познакомимся в главе 6, раздел “Работа с процессами и потоками”.

Любую асинхронную функцию можно выполнить синхронно с помощью методов `WaitForRequest()` и `WaitForAnyRequest()` статического класса `User`. Это позволяет не прибегать к использованию активных объектов в тех случаях, когда разработчик считает это нецелесообразным. Они также могут пригодиться в тех случаях, когда программа не может продолжать работу без результатов асинхронного вызова.

Следующий пример демонстрирует, как можно реализовать синхронный таймер. Функция `CallBackFuncL()` будет вызываться раз в десять секунд, пока не вернет `EFalse`.

```
TBool CallBackFuncL(TRequestStatus& aStatus);

void PeriodicCallL()
{
    // Интервал 10 секунд
    const TTimeIntervalMicroSeconds32 KDelay = 10000000;
    // Создаем объект-таймер
    RTimer timer;
    timer.CreateLocal();
    CleanupClosePushL(timer);
    // Переменная для хранения результата
    TRequestStatus request;
    TBool cancel(EFalse);
    do
    {
        // Вызов асинхронной функции
        timer.After(request, KDelay);
        // Ждем ее завершения
```

```

    User::WaitForRequest(request);
    // Асинхронный вызов завершен
    cancel = CallBackFuncL(request);
}
while (!cancel);

CleanupStack::PopAndDestroy(&timer);
}

```

Метод `WaitForRequest()` останавливает исполнение потока до тех пор, пока значение переданного ему в качестве аргумента объекта типа `TRequestStatus` не станет отличным от `KRequestPending`. За счет этого достигается синхронность выполнения асинхронного запроса. Будучи приостановленным, поток приложения не может реагировать на какие-либо сообщения. Если в нем реализован интерфейс и обработка пользовательского ввода программы, то она будет выглядеть зависшей.



Наш пример хорош в качестве демонстрации, но для синхронного использования таймера существуют более простые методы `User::At()` и `User::After()`.

Метод `WaitForRequest()` имеет вариант, принимающий ссылки на два объекта `TRequestStatus`. Его выполнение завершится, как только один из них получит отличное от `KRequestPending` значение. Это позволяет приостановить выполнение потока до завершения одного из двух асинхронных запросов. Выяснить, какой из них завершился первым можно, сравнив значения объектов `TRequestStatus`. Этот вариант метода `WaitForRequest()` часто применяют для синхронного вызова асинхронной функции с предельным временем ожидания. Для этого в качестве второго асинхронного запроса используется один из методов класса `RTimer`.

Метод `User::WaitForAnyRequest()` не имеет аргументов и приостанавливает поток до тех пор, пока он не получит сигнал о завершении какого-либо асинхронного запроса.

Иногда обработка результата асинхронного запроса требует предварительного завершения других асинхронных запросов или даже последовательности их вызовов. Использование методов `WaitForX()` в этом случае может оказаться невозможным. Лучшее, что мы могли бы сделать, — еще раз вызвать метод `CActiveScheduler::Start()`. Это позволило бы остановить ход выполнения одного активного объекта, в то же время позволив функционировать остальным. Но так как планировщик у нас один, еще раз вызвать метод `Start()` не представляется возможным.

Для решения этой проблемы существует класс `CActiveSchedulerWait`. Он позволяет реализовать вложенный цикл обработчика текущего планировщика. Объект `CActiveSchedulerWait` может быть объявлен как член C-класса (например, активного объекта) и предоставляет два основных метода `Start()` и `AsyncStop()`. Они оказывают на ход выполнения про-

граммы тот же эффект, что и методы `CActiveScheduler::Start()` или `CActiveScheduler::Stop()`. Очень часто `CActiveSchedulerWait` используют в активных объектах, запуская в методе `StartL()` сразу после вызова асинхронной функции и метода `SetActive()` и завершая в `RunL()`. Это позволяет быть уверенным, что после выполнения метода `StartL()` результат асинхронного запроса уже получен, и допустимо использовать асинхронный объект как синхронный. Например, мы можем сделать синхронным вызов метода `StartL()` класса `CLongTask` и возвращать из него количество этапов, потребовавшихся для завершения операции. Для этого в заголовочном файле необходимо объявить следующие члены класса.

```
TInt iStep;
CAActiveSchedulerWait iWait;
```

А также изменить тип результата функции `StartL()`.

```
TInt StartL();
```

После чего в теле метода `StartL()` после имитации асинхронного запроса добавить следующие строки.

```
iStep = 0;
iWait.Start();
return iStep;
```

Метод `RunL()` в этом случае необходимо переделать следующим образом.

```
void CLongTask::RunL()
{
    if (iState == EWorking && iStep < 10)
    {
        iStep++;
        // Выполнение этапа

        SetActive();
        TRequestStatus* status = &iStatus;
        User::RequestComplete(status, KErrNone);
    }
    else
    {
        // Поэтапное выполнение процесса завершено
        // Остановка CActiveSchedulerWait
        iWait.AsyncStop();
    }
}
```

Дополнительной инициализации или уничтожения объект `CActiveSchedulerWait` не требует.

Подготовка к сертификации ASD

- Понимание разницы между синхронными и асинхронными функциями. Умение распознать примеры тех и других.
 - Знание примеров типового использования активных объектов для выполнения асинхронных запросов без блокирования потока.
 - Понимание разницы между многозадачностью на основе нескольких потоков и на основе активных объектов, а также почему использование АО предпочтительнее в Symbian OS.
 - Понимание значимости приоритета активного объекта.
 - Понимание того, что выполнение обработчика событий (`RunL()`) в активном объекте не может прерываться.
 - Знание наследуемых характеристик активных объектов и функций, которые они должны реализовывать или переопределять.
 - Знание того, как правильно создать и уничтожить активный объект.
 - Понимание роли и характеристик планировщика активных объектов.
 - Знание того, что метод `CActiveScheduler::Start()` должен вызываться только после того, как хотя бы один активный объект запустит асинхронный запрос.
 - Знание того, что типичной причиной сбоев при обработке событий является то, что планировщик не был запущен или был остановлен преждевременно.
 - Понимание того, что от класса `CActiveScheduler` могут наследоваться новые классы, и причин, по которым это происходит.
 - Понимание различных последовательностей выполнения кода при завершении асинхронного запроса и при его отмене с помощью метода `Cancel()`.
 - Знание того, как использовать активные объекты для выполнения задач в фоновом режиме и их поэтапного выполнения.
 - Понимание процессов, происходящих при завершении активным объектом собственного запроса при поэтапном выполнении задачи.
 - Понимание возможных причин паники, вызванной потерянным сигналом (stray signal).
 - Понимание того, что метод `User::After()` блокирует поток до истечения указанного в параметре времени.
 - Понимание того, что типичной причиной блокирования потока может быть неуместное использование метода `User::WaitForRequest()`.
-

ГЛАВА 6

Разработка приложений

Теперь, после знакомства с основами Symbian C++, мы можем перейти к разработке простейших программ и библиотек. В данной главе рассматривается работа с рядом наиболее часто используемых системных API, не являющихся специфичными для мобильных устройств. Информацию об API, оставшихся за рамками данной книги, вы можете почерпнуть из справочника SDK или Википедий порталов Symbian Foundation и Forum Nokia.

Проект Symbian C++ приложения имеет довольно сложную структуру, подробно описанную в *главе 2*. Также далеко не просты в использовании рассматриваемые в *главе 3* инструменты SDK. Поэтому для создания приложений мы будем активно применять средства IDE Carbide.c++. Это позволяет существенно упростить и ускорить процесс разработки.

Приложение Hello World на Symbian C++

Мы приступаем к созданию первого приложения, и решать эту задачу будем с помощью мастера New Symbian OS C++ Project, подробно описанного в *главе 4*, раздел “Создание и импорт существующих проектов”. В качестве типа нового проекта можно выбрать **Empty project for Symbian**. Однако в этом случае в нем будет автоматически создана только папка `\group\` с файлами `bld.inf` и `*.mmr`, и вам придется самостоятельно создавать папки `\src\` и `\inc\`, файлы исходного кода, PKG-скрипт, редактировать MMR-файл и многое другое. Я считаю это неудобным. Гораздо быстрее простое приложение можно создать, выбрав в качестве типа проекта **Basic console application**. При этом вы получите готовую программу, выводящую строку **Hello, world!** в консоль. Ее код легко изменить: например, превратить консольное приложение в выполняющийся в фоновом режиме сервис. Такой подход позволяет сократить время на создание инфраструктуры проекта и избежать при этом возможных ошибок. Поэтому мы не будем создавать свое первое приложение с нуля, а воспользуемся проектом типа **Basic console application**.

Итак, в окне мастера для выбора типа нового проекта укажите **Basic console application** и задайте в качестве его названия **HelloWorld**. После окончания работы мастера новый проект появится в панели **Project Explorer**. Это уже готовое приложение, которое можно собрать, запустить в эмуляторе или поместить в SIS-пакет для установки на устройство.

В папке `\group\` вы обнаружите файл `bld.inf`, содержащий перечень компонентов, формирующих данный проект. В нашем случае таких компонентов всего один — находящийся в той же папке файл `HelloWorld.mmp`. В этом MMP-файле указана служебная информация для сборки исполняемого файла `HelloWorld.exe` (листинг 6.1).

Листинг 6.1. Файл `HelloWorld.mmp`

```
TARGET      HelloWorld.exe
TARGETTYPE   exe
UID          0 0xE6205EE2

USERINCLUDE   ..\inc
SYSTEMINCLUDE \epoc32\include

SOURCEPATH    ..\src
SOURCE        HelloWorld.cpp

LIBRARY       euser.lib
```

Значение `UID3`, равное в приведенном листинге `0xE6205EE2`, у вас может отличаться. Не будучи специально указанным в мастере создания проекта, оно генерируется случайным образом из незащищенного диапазона значений (см. главу 1, раздел “Уникальные идентификаторы в Symbian OS”) в расчете на то, что вероятность нахождения в системе установленной программы с таким же `UID3` крайне мала.

Как видите, при компиляции используется только один файл исходного кода, расположенный в папке `\src\`. Подключаемые заголовочные файлы будут взяты из папки `\inc\` проекта и папки `\epoc32\include\` в SDK. Системная динамическая библиотека `euser` содержит реализацию большинства стандартных API, необходимых для работы приложения: например, стек очистки и планировщик активных объектов. Вряд ли вам удастся создать сколь бы то ни было функциональную программу без ее использования. Поэтому мы можем говорить о ней как о необходимом условии и утверждать, что библиотека `euser` должна подключаться всегда.

Теперь мы перейдем к изучению автоматически сгенерированного исходного кода. Для этого вам необходимо открыть в панели **Project Explorer** файлы `HelloWorld.h` (листинг 6.2) и `HelloWorld.cpp` (листинг 6.3), размещенные в папках `\inc\` и `\src\` соответственно. В предлагаемых здесь листингах исходные комментарии на английском языке переведены на русский.

Листинг 6.2. Файл `HelloWorld.h`

```
#ifndef __HELLOWORLD_H__
#define __HELLOWORLD_H__

// Заголовочные файлы
```

```
#include <e32base.h>

// Прототипы функций

GLDEF_C TInt E32Main();

#endif // __HELLOWORLD_H__
```

Заголовочный файл содержит лишь прототип функции `E32Main()`. На самом деле это объявление не играет роли, и мы вполне могли бы обойтись без `HelloWorld.h` вообще. Но, как я уже говорил, исходный код проекта данного типа часто изменяют в соответствии с новыми задачами, и заголовочный файл может вам понадобиться. Чтобы разработчику не приходилось создавать его самостоятельно, генерируется `HelloWorld.h`.

Вы, возможно, уже обратили внимание на псевдоним `GLDEF_C`. Если вы выделите его и нажмете <F3>, то Carbide.c++ откроет для вас файл с его определением. Оно находится в файле `e32def.h` и выглядит следующим образом.

```
#define GLDEF_C
```

В этом же заголовочном файле вы найдете еще ряд схожих псевдонимов. В дальнейшем они встретятся нам при изучении кода в файле `HelloWorld.cpp`.

```
#define GLREF_D extern
#define GLDEF_D
#define LOCAL_D static
#define GLREF_C extern
#define LOCAL_C static
```

Согласно справочнику SDK¹, эти псевдонимы следует использовать при объявлении глобальных и локальных функций, а также переменных. Выражения с суффиксом “_D” (от data) должны применяться для данных, а с “_C” (от code) — для функций. Считается, что они делают код более понятным, так как, например, ключевое слово `static` может употребляться в нескольких значениях.

Листинг 6.3. Файл `HelloWorld.cpp`

```
// Заголовочные файлы

#include "HelloWorld.h"
#include <e32base.h>
#include <e32std.h>
#include <e32cons.h> // Консоль

// Константы

_LIT(KTextConsoleTitle, "Console");
```

¹ FAQ-0441 от 09.11.2002 г.

214 Symbian C++. Программирование для мобильных телефонов

```
_LIT(KTextFailed, " failed, leave code = %d");
_LIT(KTextPressAnyKey, " [press any key]\n");

// Глобальные переменные

LOCAL_D CConsoleBase* console; // Сюда пишутся все сообщения

// Локальные функции

LOCAL_C void MainL()
{
    //
    // Поместите сюда код своей программы, например следующее
    //
    console->Write(_L("Hello, world!\n"));
}

LOCAL_C void DoStartL()
{
    // Создание планировщика (для работы активных объектов)
    CActiveScheduler* scheduler = new (ELeave) CActiveScheduler();
    CleanupStack::PushL(scheduler);
    CActiveScheduler::Install(scheduler);

    MainL();

    // Удаление планировщика активных объектов
    CleanupStack::PopAndDestroy(scheduler);
}

// Глобальные функции

GLDEF_C TInt E32Main()
{
    __UHEAP_MARK;
    // Создание стека очистки
    CTrapCleanup* cleanup = CTrapCleanup::New();

    // Создание консоли для вывода сообщений
    TRAPD(createError, console = Console::NewL(KTextConsoleTitle,
        TSize(KConsFullScreen, KConsFullScreen)));
    if (createError)
        return createError;

    // Запуск кода приложения в рамках макроса-ловушки TRAP
    TRAPD(mainError, DoStartL());
    if (mainError)
```

```

    console->Printf(KTextFailed, mainError);
    // Ожидание нажатия клавиши перед завершением работы
    console->Printf(KTextPressAnyKey);
    console->Getch();

    delete console;
    delete cleanup;
    __UHEAP_MARKEND;
    return KErrNone;
}

```

Исходный код приложения начинается с блока подключения заголовочных файлов. Первым подключается файл `HelloWorld.h` из папки `\inc\` проекта, затем три системных заголовочных файла. Порядок этих объявлений обычно не важен. Более того, зачастую подключение системных заголовочных файлов переносят в соответствующий CPP-файлу пользовательский заголовочный файл, — так их проще найти.

В файле `e32base.h` содержатся определения большинства базовых классов. Явно или неявно он подключается к файлу исходного кода всегда. В файле `e32std.h` находятся определения большинства стандартных R- и T-классов для работы с массивами, временем, действительными числами, процессами, потоками и т.д. Он, так же как файл `e32base.h`, используется очень часто. Эту пару заголовочных файлов можно подключать всегда. Что касается остальных — бессмысленно запоминать, где находится то или иное определение. Для того чтобы найти объявление какого-либо класса (например, `Console`), следует воспользоваться указателем справочника SDK либо выполнить поиск по заголовочным файлам в папке `\epoc32\include\` с помощью инструмента, запускаемого командой меню **Search⇒System** среды Carbide.c++. В нашем случае нам потребуется определение класса `CConsoleBase`, находящееся в файле `e32cons.h`.

Затем в файле `HelloWorld.cpp` объявляются константы и глобальные переменные. В качестве констант выступают три символьных дескриптора, задаваемых макросом `_LIT()`. Они используются для форматирования строки ввода и вывода для консоли. Глобальная переменная всего одна — указатель на объект класса `CConsoleBase`, реализующий консоль. Это позволяет обращаться к консоли из различных функций, не передавая указатель в каждую из них в виде аргумента.

Теперь мы рассмотрим работу функций `MainL()`, `DoStartL()` и `E32Main()`. Но сделаем это в обратном порядке, нежели их появление в коде. Дело в том, что именно функции `E32Main()` и `DoStartL()` содержат инициализацию классов, необходимых для работы всего приложения. Обычно разработчику нет необходимости в их редактировании, и он работает только с функцией `MainL()`, в которую помещается весь код, реализующий логику приложения. Но для того чтобы разобраться в работе программы, нам нужно начать с функции `E32Main()`.

```
GLDEF_C TInt E32Main()
```

Функция `E32Main()` является точкой входа (entry point) программы. Она должна присутствовать в любом EXE- или DLL-файле. С вызова `E32Main()` начинается выполнение процесса, завершение работы `E32Main()` означает окончание работы программы. Результатом выполнения функции `E32Main()` является код ошибки либо значение `KErrNone`, если сбоев в работе не произошло.

```
__UNEAP_MARK;  
<...>  
__UNEAP_MARKEND;
```

О макросах `__UNEAP_MARK` и `__UNEAP_MARKEND`, использующихся в `E32Main()`, мы уже говорили в начале предыдущей главы. Действительно, точка входа — лучшее место для первичного контроля утечек памяти кучи приложения.

```
CTrapCleanup* cleanup = CTrapCleanup::New();  
<...>  
delete cleanup;
```

Следующим шагом в инициализации программы является создание стека очистки (класс `CTrapCleanup`). Без него невозможно использовать макросы `TRAP/TRAPD`, а значит, и обрабатывать сбросы. Большинство же С-классов при инициализации может вызвать сброс. Поэтому стек очистки должен создаваться в первую очередь.

```
// Создание консоли для вывода сообщений  
TRAPD(createError, console = Console::NewL(KTextConsoleTitle,  
      TSize( KConsFullScreen, KConsFullScreen)));  
if (createError)  
    return createError;  
// Запуск кода приложения в рамках макроса-ловушки TRAP  
TRAPD(mainError, DoStartL());  
if (mainError)  
    console->Printf(KTextFailed, mainError);  
// Ожидание нажатия клавиши перед завершением работы  
console->Printf(KTextPressAnyKey);  
console->Getch();  
  
delete console;
```

Затем в приложении создается экземпляр консоли. Мы отложим рассмотрение ее функциональности и разберемся с тем, почему она создается именно здесь. Как вы помните, помещать объекты в стек очистки можно только в рамках макроса-ловушки, и все они должны быть извлечены из него по завершению работы макроса. Поэтому из функции `E32Main()` вызывается процедура `DoStartL()`, защищенная ловушкой. В ней уже можно работать со стеком очистки и выполнять дальнейшую инициализацию. Решение создать консоль в функции `E32Main()` продиктовано тем, что консоль также используется для вывода информации о произошедших в процедуре `DoStartL()` сбросах. В иных

случаях ее инициализацию логичнее перенести в саму процедуру `DoStartL()` или даже в процедуру `MainL()`.

```
LOCAL_C void DoStartL()
{
    // Создание планировщика (для работы активных объектов)
    CActiveScheduler* scheduler =
        new (ELeave) CActiveScheduler();
    CleanupStack::PushL(scheduler);
    CActiveScheduler::Install(scheduler);

    MainL();

    // Удаление планировщика активных объектов
    CleanupStack::PopAndDestroy(scheduler);
}
```

В процедуре `DoStartL()` продолжается инициализация инфраструктурных объектов приложения. Как видите, здесь создается и подключается планировщик, необходимый для работы активных объектов. На данный момент в `HelloWorld.cpp` активные объекты отсутствуют, и поэтому цикл ожидания планировщика не запускается. Более того, если вы не планируете их использовать, то создание планировщика является излишним — вы можете смело удалить весь код, кроме вызова процедуры `MainL()`, или же просто заменить вызов `DoStartL()` на вызов `MainL()`.

```
LOCAL_C void MainL()
{
    //
    // Поместите сюда код своей программы, например следующее
    //
    console->Write(_L("Hello, world!\n"));
}
```

Локальная процедура `MainL()` содержит пользовательский код. Именно здесь, наконец, и выводится строка "HelloWorld" в консоль. Результат выполнения созданной нами программы в эмуляторе представлен на рис. 6.1.

Как видите, все вышеописанное выглядит довольно сложно во многом потому, что наш автоматически сгенерированный код главным образом предназначен для использования в качестве шаблона. На самом деле аналогичное приложение, выводящее строчку "HelloWorld" в консоль, можно было бы реализовать следующим образом.

```
#include <e32base.h>
#include <e32cons.h>

GLDEF_C TInt E32Main()
{
    CTrapCleanup* cleanup = CTrapCleanup::New();
```

```
if (!cleanup)
return KErrNoMemory;

CConsoleBase * console(NULL);
TRAPD(error, console = Console::NewL(KNullDesC, TSize(
    KConsFullScreen, KConsFullScreen)));

if (!error)
{
console->Write(_L("Hello, world!\n [press any key]\n"));
console->Getch();
}

delete console;
delete cleanup;

return error;
}
```



Рис. 6.1. Выполнение консольной программы HelloWorld в эмуляторе

В заключение, обратите внимание на файл HelloWorld_EKA2.pkg в папке SIS (листинг 6.4). С его помощью можно создать SIS-пакет для установки нашего консольного приложения на устройство. Создание SIS-пакетов с помощью IDE Carbide.c++ было подробно описано в *главе 4*.

Листинг 6.4. Часть файла HelloWorld_EKA2.pkg

```
#{"HelloWorld EXE"}, (0xE6205EE2), 1, 0, 0

%{"Vendor-EN"}
:"Vendor"

"$ (EPOCROOT)Epos32\release\$(PLATFORM)\$(TARGET)\
↳HelloWorld.exe"
-":!\sys\bin\HelloWorld.exe"
```

Как видите, на устройство устанавливается только исполняемый файл и не создаются регистрационные файлы ресурсов. Это означает, что для нашего приложения не будет создана пиктограмма в соответствующем меню устройства, поэтому мы не сможем его запустить. Фактически устанавливать HelloWorld на устройство без некоторой доработки бессмысленно. Более подробно этот вопрос освещается ниже, в разделе “Регистрация программы в меню приложений”.

Консоль

Приложение HelloWorld демонстрирует использование консоли в Symbian C++. Доступ к ней осуществляется посредством базового класса CConsoleBase, предоставляющего общий интерфейс. Класс CConsoleBase содержит множество чистых виртуальных функций, поэтому вы не сможете создать экземпляр этого класса. Для получения конкретной реализации консоли используется статический класс Console, предоставляющий всего один метод-конструктор: NewL(). Результатом работы конструктора NewL() является указатель на объект, порожденный от класса CConsoleBase. Реализация консоли в значительной степени зависит от модели устройства, поэтому SDK не предоставляет конкретного класса для доступа к ней. Итак, создать консоль можно следующим образом.

```
_LIT(KTextConsoleTitle, "Console");
TSize sz(KConsFullScreen, KConsFullScreen);
CConsoleBase* console = Console::NewL(KTextConsoleTitle, sz);
```

Аргументами конструктора NewL() являются название консоли, отображаемое в диспетчере приложений системы, и ее размер (структура класса TSize). Константа KConsFullScreen равна -1 и является специфичной для класса Console. Не следует думать, что объект TSize(-1, -1) всегда соответствует максимальному размеру экрана устройства. Несмотря на то, что в эмуляторе

консоль отображается в диспетчере приложений, на настоящем устройстве это не так. Более того, размеры окна консоли также игнорируются — она всегда выводится на весь экран. Поэтому в действительности передаваемые в `NewL()` аргументы не имеют значения.

Базовый класс `CConsoleBase` предоставляет множество методов для управления консолью, но многие из них не работают либо не имеют никакого эффекта на настоящем устройстве. Поэтому далеко не все из них перечислены ниже.

- Метод `ScreenSize()` — позволяет получить размер консоли. Консоль состоит из n строк, в каждой из которой может содержаться по m символов. В качестве результата выполнения метода `ScreenSize()` возвращается объект `TSize`, содержащий значения m и n .
- Процедура `ClearScreen()` — полностью очищает окно консоли от текста.
- Методы `CursorPos()` и `SetPos()` — позволяют получить и изменить текущее положение курсора в консоли. Позиция $(0, 0)$ соответствует верхнему левому углу.
- Методы `WhereY()` и `WhereX()` — возвращают номер текущей строки и позицию курсора в ней соответственно.
- Процедура `ClearToEndOfLine()` — удаляет текст с текущей позиции курсора до конца строки.
- Метод `Write()` — отображение строки в консоли. Данные выводятся, начиная с текущей позиции курсора. В качестве аргумента передается `Unicode-дескриптор`, содержащий строку. Если размеры консоли не позволяют вывести строку целиком, то ее содержимое смещается на одну строку к началу.
- Метод `Printf()` — вывод отформатированной строки. Первым аргументом является строка-шаблон, в соответствии с которой происходит форматирование. Последующие аргументы содержат форматируемые значения. Работа метода аналогична функции `Format()` класса `TDes`, с той разницей, что результат сразу выводится в консоль, а не помещается в изменяемый дескриптор.
- Метод `KeyCode()` — возвращает код последней нажатой пользователем клавиши.
- Метод `KeyModifiers()` — позволяет получить коды клавиш-модификаторов (например, `<Shift>` или `<Alt>`), которые, возможно, удерживались при последнем нажатии.
- Метод `Read()` — асинхронный, запрос которого завершается при нажатии пользователем клавиши.
- Метод `ReadCancel()` — отмена асинхронного запроса `Read()`.
- Метод `Getch()` — синхронный метод для получения кода нажатой пользователем клавиши. Его работа аналогична нижеприведенной функции.

```
TKeyCode CConsoleBase::Getch()
{
    TRequestStatus status;
    Read(status);
```

```
User::WaitForRequest(status);
return KeyCode();
}
```

Как видите, консоль имеет достаточно большие возможности по отображению информации. С ее помощью можно выводить Unicode-строки в указанной позиции. Этого вполне достаточно даже для отображения псевдографики. Но в то же время методы получения вводимых пользователем данных довольно примитивны и годятся разве что для обработки ответа на сообщение **Press any key to continue....**

Регистрация программы в меню приложений

Как уже отмечалось ранее, созданная нами программа не может быть запущена на устройстве пользователем, так как доступ к ней отсутствует в системном меню приложений. Это очень неудобно, ведь разработчику в результате приходится прибегать к различным трюкам для организации автоматического запуска. Необходимо заметить, что эта проблема не возникает при создании GUI-приложений с помощью шаблонов Carbide.c++. В этом случае все необходимые файлы и пиктограммы уже будут присутствовать в проекте. Тем не менее вам необходимо иметь представление о том, в результате чего пиктограммы приложений появляются в системном меню и что нужно для этого сделать. К тому же создать выполняющийся в фоновом режиме сервис (например, для мониторинга входящих сообщений) из шаблона GUI-приложения довольно сложно — для этого потребуется слишком много изменений, а использовать само GUI-приложение в качестве сервиса слишком накладно — это влечет бессмысленный расход памяти и энергии батареи.

Итак, для того, чтобы зарегистрировать приложение в системном меню, необходимо создать файл ресурсов специального вида. Он является регистрационным и должен быть помещен в папку `\private\10003a3f\import\apps\`. Как вы знаете из главы 1, раздел “Политика безопасности Symbian OS”, эта папка находится в приватном каталоге процесса с SID, равным `0x10003A3F`, а именно, системного процесса `AppArcServer`. Доступ к ней из стороннего приложения возможен только в момент его установки на устройство. В регистрационном ресурсе содержится UID3 и имя исполняемого файла, который должен запускаться из меню приложений. Этого вполне достаточно для того, чтобы его обнаружить — ведь все исполняемые файлы и библиотеки хранятся в каталоге `\sys\bin\` (см. главу 1, раздел “Экранирование данных”). Регистрационный ресурс обычно имеет имя `<MyApplication>_reg.rsc`.

Для того чтобы создать такой файл, выделите в панели **Project Explorer** папку `data` вашего проекта и выберите в ее контекстном меню команду **NewFile**. В качестве имени файла укажите `HelloWorld_reg.rss`. IDE Carbide.c++ создаст для вас пустой файл в заданной папке. Открыв его, введите код, представленный в листинге 6.5.

Листинг 6.5. Файл HelloWorld_reg.rss

```
#include <appinfo.rh>

UID2    KUidAppRegistrationResourceFile
UID3    0xE6205EE2 // Замените на UID3 вашего приложения

RESOURCE APP_REGISTRATION_INFO
{
    app_file="HelloWorld"; //Имя исполняемого файла без ".exe"
}
```

В регистрационном файле запись формата APP_REGISTRATION_INFO всегда должна быть первой. Подключаемый к HelloWorld_reg.rss заголовочный файл appinfo.rh содержит определение этой структуры. В ней содержится довольно большое число необязательных и крайне редко используемых полей, рассмотрение которых займет довольно продолжительное время. Поэтому на данный момент мы ограничимся только тремя.

- `app_file` — имя исполняемого файла без расширения. Эта программа будет запускаться из системного меню.
- `hidden` — может принимать два значения: `KAppNotHidden` (по умолчанию) и `KAppHidden` и позволяет скрыть пиктограмму приложения в главном меню. Вы не сможете контролировать видимость пиктограммы своего приложения после установки программы, так как не получите доступа к регистрационному файлу. Этот параметр может пригодиться для быстрого включения или выключения отображения пиктограммы в системном меню без редактирования PKG-скрипта или MMP-файла (при работе в эмуляторе). Но на самом деле с его помощью разрешается ситуация, при которой регистрационный файл предназначен лишь для извещения системы о каких-то свойствах приложения (мы не рассматриваем такие поля), но сама программа в меню отображаться не должна.
- `launch` — по умолчанию содержит значение `KAppLaunchInForeground`, но может принимать значение `KAppLaunchInBackground` и определяет, будет ли выполняться запущенное приложение в фоновом режиме.

Заметьте, что запись не имеет имени — в нем нет необходимости, так как мы не собираемся использовать ее самостоятельно.

Выражения UID2 и UID3 позволяют переопределить соответствующие идентификаторы в файле HelloWorld.rsc (см. главу 2, раздел “Прочие выражения файлов ресурсов”). Значение UID2 в регистрационном файле ресурсов всегда должно равняться константе `KUidAppRegistrationResourceFile`, определенной в файле appinfo.rh. В противном случае сервер AppArcServer ваш файл проигнорирует. Значение UID3 должно совпадать с UID3 запускаемого исполняемого файла.

Теперь вы должны добавить в MMP-файл информацию о новом файле ресурсов, иначе он не будет компилироваться в файл `HelloWorld.rsc` при сборке проекта. Для этого откройте файл `HelloWorld.mmp`, перейдите на вкладку **Source** и, выделив элемент **Resources** в дереве **Resources**, щелкните на кнопке **Add**. Откроется диалоговое окно **Edit Resource Block**, представленное на рис. 6.2.

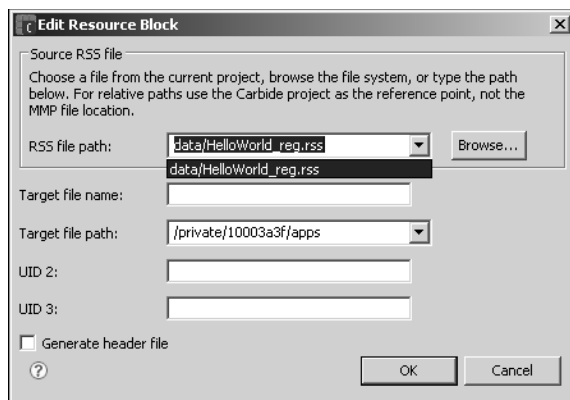


Рис. 6.2. Добавление файла ресурсов в MMP-файл

Выберите в раскрывающемся списке **RSS file path** добавляемый файл ресурсов. На данный момент в нашем проекте он один. Затем введите в поле **Target file path** следующий путь: `/private/10003a3f/apps`. Это относительный путь к файлу от папки эмулятора \$ (EP0CROOT) `Ер0с32\data\z\` для размещения регистрационных ресурсов. Как вы помните, приложение попадает в эмулятор, минуя стадию установки SIS-файла. С помощью параметра **Target file path** указывается, в какую папку эмулятора необходимо поместить скомпилированный файл ресурсов. Обратите внимание, что поля **UID2** и **UID3** пусты — нам незачем их заполнять, так как верные значения уже указаны в самом файле `HelloWorld_reg.rss`. Поле **Target file name** позволяет изменить имя скомпилированного файла, в чем у нас также нет необходимости. Файл `HelloWorld_reg.rss` будет скомпилирован в `HelloWorld_reg.rsc`. Флажок **Generate header file** позволяет сгенерировать заголовочный файл `HelloWorld_reg.rsg`, содержащий идентификаторы всех имеющихся в регистрационном ресурсе структур. Но, так как мы не собираемся считывать из него данные в нашем приложении (да и не можем в силу политики экранирования данных), то данный флажок лучше оставить сброшенным.

Все то, что мы только что проделали на вкладке **Source**, можно выполнить другим способом, — просто добавив в MMP-файл следующие строки.

```
SOURCEPATH ..\data
START RESOURCE HelloWorld_reg.rss
    TARGETPATH \private\10003a3f\apps
END
```

Теперь, собрав проект для целевой сборки WINSW и запустив эмулятор, вы обнаружите в системном меню приложений эмулятора пиктограмму **HelloWorld** с помещенным на нее изображением, выбираемым по умолчанию. При выборе этой пиктограммы будет запускаться наша программа.



Рис. 6.3. Программа HelloWorld в системном меню

Для того чтобы регистрационный файл ресурсов HelloWorld_reg.rsc попал в сборку SIS-файла и был установлен в нужный каталог, необходимо в скрипт пакета HelloWorld.pkg добавить следующую строку.

```
"$(EPOCROOT)Epoc32\data\z\private\10003a3f\apps\  
HelloWorld_reg.rsc"  
-":\private\10003a3f\import\apps\HelloWorld_reg.rsc"
```

Изменение подписи пиктограммы

Как видите, все довольно просто. Вышеперечисленных действий вполне достаточно, чтобы вручную запустить приложение в эмуляторе или устройстве. Но, возможно, вам захочется изменить подпись под пиктограммой в меню. Этого можно достичь нижеописанным способом.

Нам понадобится добавить в проект еще один файл ресурсов и создать в нем запись структуры LOCALISABLE_APP_INFO. В отличие от нашего проекта, в шаблоне проекта GUI-приложения эта запись автоматически создается в главном файле ресурсов. Главным для GUI-приложения называется файл ресурсов, полное имя которого возвращает метод ResourceFileName() класса CEikApplication, и он загружается при запуске программы автоматически. Мы не будем в данный момент рассматривать архитектуру GUI-приложений. Достаточно сказать, что по умолчанию главный файл ресурсов находится в папке \resource\apps\ и имеет такое же имя, как и исполняемый файл программы. Для единообразия с шаблоном GUI-приложения мы создадим в нашем проекте аналогичный файл.

Итак, создайте в папке \data\ проекта пустой файл с именем HelloWorld.rss, затем подключите его к MMP-файлу, указав в качестве значения параметра **Target file path** путь \resource\apps и установив флажок **Generate header file** (нам понадобится файл HelloWorld.rsg). Теперь во время сборки проекта будет компилироваться файл ресурсов HelloWorld.rsc, а затем генерироваться его заголовочный файл HelloWorld.rsg. В самом файле HelloWorld.rss необходимо добавить строки, приведенные в листинге 6.6.

Листинг 6.6. Файл HelloWorld.rss

```
#include <appinfo.rh>

RESOURCE LOCALISABLE_APP_INFO r_localisable_app_info
{
    // Краткое название приложения
    short_caption = "Hello World";
}
```

Заголовочный файл `appinfo.rh`, подключенный в самом начале файла ресурсов, содержит определение структуры `LOCALISABLE_APP_INFO`. Формат структуры `LOCALISABLE_APP_INFO` довольно сложен, и разобраться в нем, не имея представления об архитектуре GUI-приложений, мы не сможем. На данный момент нам вполне достаточно знать лишь о назначении поля `short_caption`. Оно позволяет задать краткое название приложения. Краткое название отображается в случае, если для вывода на экран обычного названия не хватает места. В частности, именно оно используется в качестве подписи к пиктограмме программы в меню приложений. Заметьте, что наш ресурс имеет имя `r_localisable_app_info`. Вы можете использовать любое другое имя, но оно должно быть обязательно — только именованные ресурсы попадут в сгенерированный RSG-файл.

Теперь нам необходимо уведомить систему о том, что у нас имеется реализация структуры `LOCALISABLE_APP_INFO`, в каком файле ресурсов она реализована, и как ее в нем найти. Для этого в регистрационный ресурс `HelloWorld_reg.rss` необходимо внести изменения, как показано в листинге 6.7.

Листинг 6.7. Измененный файл HelloWorld_reg.rss

```
#include <appinfo.rh>
#include "HelloWorld.rsg"

UID2    KUidAppRegistrationResourceFile
UID3    0xE6205EE2 // Замените на UID3 вашего приложения

RESOURCE APP_REGISTRATION_INFO
{
    app_file="HelloWorld"; //Имя исполняемого файла без ".exe"
    localisable_resource_file = "\\resource\\apps\\HelloWorld";
    localisable_resource_id = R_LOCALISABLE_APP_INFO;
}
```

Как видите, мы проинициализировали в ресурсе структуры `APP_REGISTRATION_INFO` два поля значениями, отличными от значений по умолчанию. Именно они позволят системе найти и прочитать данные записи `LOCALISABLE_APP_INFO`.

- `localisable_resource_file` — определяет путь и имя файла ресурсов, содержащего запись структуры `LOCALISABLE_APP_INFO`. Расширение не указывается, так как система может найти подходящий локализованный вариант этого файла (см. главу 2, раздел “Локализация и компиляция файла ресурса”).
- `localisable_resource_id` — идентификатор записи структуры `LOCALISABLE_APP_INFO` в файле, определенном в поле `localisable_resource_file`.

В качестве значения поля `localisable_resource_id` используется псевдоним `R_LOCALISABLE_APP_INFO`, определенный в заголовочном файле ресурса. Его определение находится в автоматически генерируемом файле `HelloWorld.rsg`, который мы подключаем вместе с файлом `appinfo.rh`. Название идентификатора нам известно заранее, так как оно является именем записи ресурса, все буквы в котором переведены в верхний регистр (см. главу 2, раздел “Идентификаторы ресурсов”). В MMP-файле проекта ресурс `HelloWorld.rss` должен быть объявлен до объявления файла `HelloWorld_reg.rss`, иначе при первой сборке компилятор не обнаружит еще не сгенерированный файл `HelloWorld.rsg`.

Запустив эмулятор, вы можете убедиться, что подпись пиктограммы приложения в меню изменилась с **HelloWorld** на **Hello World**. Для того чтобы новый файл ресурсов попал в SIS-пакет и был установлен на устройство, в PKG-скрипт необходимо добавить следующую строку.

```
"$(EPOCROOT)Epoc32\data\z\resource\apps\HelloWorld.rsc"
-":\resource\apps\HelloWorld.rsc"
```



Разместить запись структуры `R_LOCALISABLE_APP_INFO` в самом регистрационном файле ресурсов нельзя.

Изменение пиктограммы

На данный момент в системном меню для нашего приложения отображается пиктограмма, выбираемая по умолчанию. Разработчик может заменить ее, поместив содержащий требуемое изображение файл на устройство и подключив его в структуре `R_LOCALISABLE_APP_INFO`. Сделать это довольно сложно, во многом не из-за необходимости внесения большого количества изменений в проект, а из-за множества разнообразных нюансов, связанных с созданием самой пиктограммы.

В Symbian 9.x используются пиктограммы в формате SVG-T, упакованные в файл формата MIF. Формат SVG (от англ. Scalable Vector Graphics — масштабируемая векторная графика) является подмножеством расширяемого языка разметки XML и предназначен для описания двухмерной векторной и смешанной векторно-растровой графики. Это значит, что файл с расширением `.svg` вы можете открыть в любом текстовом редакторе и обнаружите там большое количество разнообразных элементов, похожих на теги языка HTML. Векторную графику,

в отличие от растровой, можно масштабировать без потери качества. Поскольку Symbian OS используется на устройствах с экранами различных размеров и разрешений, масштабируемость изображений для этой операционной системы является крайне важным свойством. Действительно, если вам удастся создать красивую пиктограмму для Symbian 9.1 — она без каких-либо изменений будет одинаково хорошо смотреться и на других устройствах под управлением Symbian 9.x. Формат SVG-T (SVG Tiny) является спецификацией SVG для мобильных устройств: в нем отсутствуют возможности, которые не могут быть переданы в силу аппаратных ограничений. Разработкой форматов SVG занимается консорциум W3C².

Для создания изображений в формате SVG могут использоваться современные редакторы векторной графики. Сообщество Forum Nokia для этих целей рекомендует бесплатный редактор Inkscape³ с открытым исходным кодом (существуют версии для Windows, Linux и Mac OS). На практике же многие разработчики пытаются не нарисовать пиктограмму с нуля, а конвертировать имеющийся растровый файл в формат SVG. Это действительно можно сделать, но нужно учесть следующее. Большинство подобных конверторов переводит изображение в формат SVG, а не SVG-T. Формат SVG допускает использование растровых вставок, а SVG-T — нет. Чаще всего подобные редакторы не превращают растровую графику в векторную, а просто помещают растровую вставку в файл .svg. Такая пиктограмма на мобильном устройстве *отображаться не будет*. Вы можете открыть SVG-файл в текстовом редакторе и поискать в нем элементы `image`, содержащие base64-кодированные изображения в формате PNG. Если они в нем присутствуют, то такая пиктограмма нам не подойдет. Поставляющийся вместе с SDK конвертер `svg2svgt` не решает эту проблему. С его помощью можно лишь удостовериться, что изображение соответствует спецификации SVG-T, а растровые вставки при этом из файла будут удалены. Поэтому, прежде чем менять пиктограмму в приложении, убедитесь, что новая пиктограмма полностью удовлетворяет стандарту SVG-T.

Перед использованием пиктограмма в формате SVG-T должна быть упакована в MIF-файл (Multi Image File). В таком файле могут храниться несколько SVG- или XML-изображений, доступ к которым осуществляется по индексу. Для генерации MIF-файла используется утилита `mifconv`, входящая в состав SDK. Здесь мы должны учесть еще один нюанс. Так как формат SVG является подмножеством языка XML, то закодированные в соответствии с ним изображения, по сути, являются текстовыми файлами и могут иметь довольно большой размер. Для сжатия SVG-файла может использоваться входящая в SDK утилита `svgtbinencode`. Но поддержка SVG-T изображений в бинарной форме появилась лишь в Symbian 9.2, и до S60 3 FP1 SDK утилита `mifconv` по умолчанию подпрограмму `svgtbinencode` не вызывала. Это привело к нарушению обратной совместимости: пиктограммы приложений, собранных с помощью S60 3 FP1 SDK и выше, не отображались на устройствах под управлением Symbian 9.1. Реша-

² <http://www.w3.org/Graphics/SVG/>

³ <http://www.inkscape.org>

ется данная проблема добавлением ключа \x при вызове утилиты mifconv, что отключает бинарное кодирование SVG-файлов.

Итак, предположим, что вы создали пиктограмму (или нашли *.svg файл в SDK) для вашего приложения, соответствующую требованиям стандарта SVG-T, и назвали ее helloworld.svg. Создайте в проекте папку \gfx\ и поместите туда этот файл. Мы не собираемся создавать из него MIF-файл вручную, а воспользуемся возможностью добавления собственных сборочных скриптов. Для этого в папке group создайте файл HelloWorld_icons.mk и поместите в него текст, представленный в листинге 6.8.

Листинг 6.8. Файл HelloWorld_icons.mk

```
# Корректировка пути для старой целевой платформы WINS
ifeq (WINS,$(findstring WINS, $(PLATFORM)))
ZDIR=$(EPOCROOT)epoc32\release\$(PLATFORM)\$(CFG)\Z
else
ZDIR=$(EPOCROOT)epoc32\data\z
endif

# Папка для размещения полученного mif файла
TARGETDIR=$(ZDIR)\resource\apps
# Имя получаемого mif файла
ICONTARGETFILENAME=$(TARGETDIR)\helloworld.mif
# Папка с исходными изображениями
ICONDIR=..\gfx

do_nothing :
    @rem do_nothing

MAKMAKE : do_nothing
BLD : do_nothing
CLEAN : do_nothing
LIB : do_nothing
CLEANLIB : do_nothing

RESOURCE : $(ICONTARGETFILENAME)

$(ICONTARGETFILENAME) : $(ICONDIR)\helloworld.svg
# Сборка mif файла
    mifconv $(ICONTARGETFILENAME) \
        /c32 $(ICONDIR)\helloworld.svg

FREEZE : do_nothing
SAVESPACE : do_nothing

RELEASESABLES :
    @echo $(ICONTARGETFILENAME)

FINAL : do_nothing
```



Чтобы избежать вышеописанной проблемы с нарушением обратной совместимости MIF-файлов, следует при вызове `mifconv` перед ключом `/s32` добавить ключ `/x`. В шаблоне GUI-проекта `.mk`-файл со скриптом для сборки MIF-файла создается автоматически.

Для того чтобы наш скрипт вызывался при сборке проекта, необходимо в файл `\group\bld.inf` добавить строку `gnumakefile HelloWorld_icons.mk` в секцию `PRJ_MMPFILES`. Затем нужно регенерировать все вспомогательные скрипты, использующиеся для сборки проекта (см. главу 2, раздел “Сборка проекта”). Очистите проект с помощью команды меню **Project⇒Clean** в Carbide.c++. Затем, выделив папку `group` в панели **Project Explorer**, выберите команду **Open Command Window** в ее контекстном меню. В открывшемся окне консоли введите и выполните команду **bldmake clean** — это удалит MAKE-сборщики первого уровня (см. главу 3, раздел “Очистка проекта”).

Теперь SVG-пиктограмма будет автоматически упаковываться в MIF-файл. Он будет сохраняться в папке `\resource\apps\`, путь к которой здесь задан относительно каталога целевой платформы.

Для установки пиктограммы на устройство в PKG-скрипт следует добавить следующую строку.

```
"$ (EPOCROOT) Eroc32\data\z\resource\apps\HelloWorld.mif"
-":\resource\apps\HelloWorld.mif"
```

Все что нам осталось — это объявить в ресурсе `R_LOCALISABLE_APP_INFO`, где именно находится наша пиктограмма. Для этого запись `r_localisable_app_info` файла `HelloWorld.rss` должна быть изменена так, как показано в листинге 6.9.

Листинг 6.9. Измененный файл `HelloWorld.rss`

```
#include <appinfo.rh>

RESOURCE LOCALISABLE_APP_INFO r_localisable_app_info
{
    // Короткое название приложения
    short_caption = "Hello World";

    caption_and_icon =
    CAPTION_AND_ICON_INFO
    {
        // Обычное название приложения
        caption = "Hello World";
        // Количество пиктограмм
        number_of_icons = 1;
        // Файл с пиктограммами
        icon_file = "\\resource\\apps\\HelloWorld.mif";
    };
}
```

Как видите, мы инициализировали новым значением поле `caption_and_icon` в структуре `LOCALISABLE_APP_INFO`. Это поле имеет тип `STRUCT` и предназначено для хранения записи формата `CAPTION_AND_ICON_INFO`. В ней определены следующие поля:

- `caption` — обычное название приложения;
- `number_of_icons` — число пиктограмм;
- `icon_file` — файл, содержащий пиктограммы.

После перекомпиляции ресурсов и сборки проекта в меню приложений эмулятора будет отображена новая пиктограмма (рис. 6.4).



Рис. 6.4. Новая пиктограмма приложения



Так как пиктограммы кешируются в системе, вам необходимо перезапустить эмулятор или настоящее устройство для того, чтобы вместо прежней пользовательской пиктограммы стала отображаться новая.

Именованье исполняемых файлов, смена идентификаторов

Все исполняемые файлы и библиотеки в Symbian OS устанавливаются в каталог `\sys\bin\` на одном из логических дисков. Если во время установки дистрибутива вашего приложения выяснится, что в данном каталоге уже есть файл с таким же именем — она завершится сообщением об ошибке. Поэтому, во избежание конфликта имен (в англоязычной литературе — “naming clash”), рекомендуется включать в название файла значение его идентификатора UID3. Таким образом, вместо `HelloWorld.exe` мы должны были бы использовать что-то вроде `HelloWorld_E6205EE2.exe` (в вашем случае UID3 может иметь другое значение).

Как вы помните, значение UID3 в большинстве случаев принимается за значение SID, а значит, должно быть уникальным в системе. Таким образом, мы заменяем две проблемы (конфликт имен файлов и значений SID) одной.

Какова же вероятность совпадения UID3? Во время тестирования приложению можно присвоить значение идентификатора из незащищенного диапазона (см. главу 1, табл. 1.2 “Диапазоны значений уникальных идентификаторов” в разделе “Идентификаторы VID и SID”). Это значение выбирается случайным образом. Еще лучше — зарегистрироваться на сайте Symbian Signed и запросить значение из защищенного диапазона. Это можно сделать бесплатно, а результат запроса возвращается мгновенно. Сайт Symbian Signed имеет базу выданных идентификаторов и никогда не выдает одно и то же значение дважды. Таким

образом, используя значение UID3 из защищенного диапазона, вы полностью решите проблему совпадения идентификаторов. На практике чаще всего UID3 запрашивают у Symbian Signed не из желания минимизировать риски при установке (они и так ничтожны), а для того чтобы в перспективе приложение могло пройти процесс сертификации.

Как видите, вероятность совпадения UID3 невелика. Несмотря на это, неопытные разработчики регулярно сталкиваются с этой проблемой. Чаще всего это связано с использованием демонстрационных примеров: разработчики собирают и пробуют их, забывая удалить с устройства, а затем, используя примеры в качестве шаблона, создают свои приложения и не могут установить, потому что забыли поменять UID3.

Иногда разработчику приходится менять идентификатор UID3 своего приложения. Обычно это связано с необходимостью использовать значение из защищенного диапазона вместо незащищенного или наоборот. Чаще всего к этому приводит неверная оценка требуемых программе защищенных возможностей. Например, вы планировали подписать ее сертификатом self-signed, а выясняется, что для вызова необходимого API потребуется декларация обращения к защищенной возможности NetworkControl. Это в свою очередь вынуждает проходить сертификацию, а значит, UID3 должен быть из защищенного диапазона. В общем случае, неважно, из-за чего вам приходится менять UID3, достаточно знать, что такая необходимость вполне может возникнуть, а вместе с ней — и необходимость изменить имя исполняемого файла.

Carbide.c++ не предоставляет инструментов для рефакторинга проекта после изменения этих значений. Разработчику придется делать все вручную. Я намеренно использовал не соответствующее рекомендациям имя исполняемого файла, чтобы показать сложность вносимых изменений. Итак, для замены UID3 и имени файла HelloWorld.exe нам придется:

- внести изменения в файл HelloWorld.mmp (ограничиться PKG-скриптом не удастся);
- изменить UID3 файла HelloWorld_res.rss и исправить в нем ссылку на исполняемый файл;
- заменить имена файлов в PKG-скрипте, изменить пути для установки файлов в приватный каталог приложения (\private\E6205EE2\).

Если бы наше приложение использовало GUI и архитектуру Avkon, то пришлось бы дополнительно менять UID3 и в ряде заголовочных файлов, а также переименовывать файл главного ресурса. При использовании UI Designer значение UID3 потребуется изменить в файле application.uidesign. Наконец, имя исполняемого файла и особенно значение UID3 могут встречаться в коде, созданном самим разработчиком. Изменения могут потребоваться и в проектах сопутствующих приложений или библиотек.

Как видите, чем сложнее проект, тем труднее менять в нем эти значения. Поэтому я настоятельно советую вам следующее.

1. Заранее побеспокойтесь об именах файлов, определитесь с методом сертификации программы и выберите подходящее значение UID3.
2. Если эти параметры пришлось менять — выполните их поиск по проекту с помощью команды меню **Search**⇒**Search** в окне Carbide.c++. Практика показывает, что их использование можно обнаружить в самых неожиданных местах.
3. Не поленитесь собрать и протестировать приложение после внесения этих изменений. Если вы что-то забыли, то, скорее всего, ошибка будет критической.

Автозапуск при запуске системы

В Symbian 9.x появилось специальное API Startup List Management для организации автозапуска приложения при включении устройства. В предыдущих версиях системы этого также можно было достичь, но с помощью довольно сложных манипуляций (запуск из загружающегося при старте системы распознавателя, см. раздел “Распознаватели”, ниже в этой главе).

Запуск приложений осуществляет специальный системный процесс App Installer, стартующий при загрузке системы. Для того чтобы зарегистрировать вашу программу в списках подлежащих запуску приложений, необходимо во время инсталляции поместить специальный регистрационный ресурс в папку `\import\` приватного каталога процесса App Installer. Из этого следует, что зарегистрировать программу на автозапуск можно *только в момент ее установки*.

Кроме того, имя файла регистрационного ресурса должно быть в формате: `[PID].rsc`, где `PID` — идентификатор SIS-пакета (указывается в PKG-скрипте). Процесс App Installer, получив PID пакета, проверяет сертификат, которым он подписан. Если это сертификат `self-generated`, то система проигнорирует ваш регистрационный файл. Это означает, что для работы механизма автозапуска необходимо подписать пакет с помощью DevCert или сертифицировать его. Регистрационный ресурс содержит информацию только об одном исполняемом файле, который должен быть запущен, и этот файл обязательно должен быть в том же пакете. Пакет должен быть типа SA (SISAPP).

Для того чтобы создать регистрационный ресурс, добавьте в папку `\data\` вашего проекта файл `HW_startup.rss`. Мы не можем использовать необходимое имя файла сразу, так как оно содержит квадратные скобки — это приводит к ошибкам в сборочных скриптах. В полученный вами файл ресурса необходимо поместить текст, представленный в листинге 6.10.

Листинг 6.10. Файл startup.rss

```
#include <startupitem.rh>

RESOURCE STARTUP_ITEM_INFO
```

```
{
executable_name = "!:\\sys\\bin\\HelloWorld.exe";
}
```

Заголовочный файл `startupitem.rh` содержит определение структуры `STARTUP_ITEM_INFO`. Она включает только два поля.

- `executable_name` — имя запускаемого исполняемого файла. “!” используется вместо литеры диска, если разработчик не знает, на какой диск будет установлена программа. В этом случае будет выполнен поиск файла на дисках в соответствии с правилами системы (см. раздел “Сессия файлового сервера” ниже в этой главе).
- `recovery` — действие в случае ошибки при запуске исполняемого файла. SDK предлагает только одно значение для этого поля: `EStartupItemPolicyNone` (ничего не предпринимать). Оно же является значением по умолчанию, поэтому мы не задаем его явно.

В регистрационном файле ресурсов может содержаться несколько записей структуры `STARTUP_ITEM_INFO` для разных исполняемых файлов, но записей других структур в нем быть не должно.

Для того чтобы `HW_startup.rss` компилировался вместе с проектом, в MMP-файл следует добавить его описание. При этом не требуется указывать никаких значений, отличных от значений по умолчанию. Параметры **Target file path** и **Target file name** не нужны, поскольку в эмуляторе API Startup List Management не работает, а переименовать файл мы можем и в PKG-скрипте. Заголовочный файл к файлу `startup.rss` также не требуется — мы все равно не сможем обратиться к нему, так как он будет помещен в приватную папку другого процесса. Таким образом, запись в MMP-файле будет выглядеть так.

```
SOURCEPATH ..\data
START RESOURCE HW_startup.rss
END
```

Теперь наш скомпилированный RSC-файл будет создаваться в соответствующей целевой сборке папке SDK. Остается лишь оформить его установку на устройство. Откройте PKG-скрипт вашего проекта и найдите в нем значение PID. Оно находится в следующей строке.

```
#{"HelloWorld EXE"}, (0xE6205EE2), 1, 0, 0
```

В моем проекте идентификатор PID равен `0xE6205EE2`, следовательно, имя помещаемого на устройство регистрационного файла будет `[E6205EE2].rsc`. В вашем проекте PID может быть иным. При создании проекта с помощью шаблона в качестве PID автоматически создаваемого PKG-скрипта используется UID3 из единственного MMP-файла. На самом деле значение PID может быть не равно ему и вообще никак с ним не связано. Как только вы выяснили идентификатор вашего будущего SIS-файла, можете добавить в него установку регистрационного ресурса с помощью следующей строки в PKG-скрипте.

```
"$(EPOCROOT)Epos32\Data\HW_startup.rsc"  
-":\private\101f875a\import\[E6205EE2].rsc"
```

Создание библиотек

При разработке сложных приложений зачастую удобнее реализовывать часть функций в библиотеках. Это могут быть как статические библиотеки, подключаемые к исполняемому файлу на этапе компиляции (LIB), так и разделяемые динамические библиотеки (DLL), подгружаемые при запуске программы.

Библиотеки позволяют повторно использовать ранее созданные функциональные блоки и делают код более понятным. Они удобны для разделения работ по реализации проекта между группами разработчиков. С их помощью можно предоставить готовую функциональность третьей стороне, не открывая исходного кода.

Разделяемые динамические библиотеки к тому же позволяют экономить память и место на диске, частично обновлять и расширять функциональность приложения.

Статически связываемые библиотеки (LIB)

Статические библиотеки (static library) являются объектными файлами с расширением `.lib`, содержащими реализацию некоторых функций, и подсоединяются в момент компиляции исходного файла. Находящийся в них код размещается в исполняемом файле, поэтому их не нужно помещать в дистрибутив приложения.

Для того чтобы создать статическую библиотеку, воспользуйтесь шаблоном **Basic static library** в соответствующем окне мастера New Symbian OS C++ Project. Для примера назовем наш будущий проект **StaticLib**. После его создания, откройте файл `StaticLib.mmp` и обратите внимание на следующие строки.

```
TARGET      StaticLib.lib  
TARGETTYPE   lib  
UID          0
```

Как видите, в качестве значения `TARGETTYPE` использовано `lib`. Именно это выражение используется сборщиком для определения типа выходного файла. Оно же должно влиять на `UID1`, но в статических библиотеках его нет, так как они используются только во время сборки исполняемого файла на ПК и на устройство под управлением Symbian OS не копируются. По той же причине в них отсутствуют идентификаторы `UID2` и `UID3`, поэтому ключевое слово `UID` задает для них нулевые значения. В статических библиотеках также не требуется декларировать доступ к защищенным возможностям.

В файле `StaticLib.h` вы найдете объявление, а в `StaticLib.cpp` — реализацию пустой функции.

```
TInt DummyFunction()
{
    //
    // Вставьте сюда свой код
    //
    return 0;
}
```

Вы можете разместить в ней свой код, добавить несколько новых функций или классов, главное — не забывайте создавать заголовочные файлы с их прототипами и объявлениями. Обратите внимание, что в библиотеке не содержится точка входа `E32Main()` — она не нужна. Если вы соберете проект, в каталоге целевой платформы SDK появится файл `StaticLib.lib`. Вам не нужно заботиться о том, в какой папке он появился. Главное — знать его имя: оно совпадает с указанным в MMP-файле.

Для того чтобы использовать код из статической библиотеки в создаваемой другим проектом программе, скопируйте в него все необходимые заголовочные файлы. Например, скопируйте `StaticLib.h` в папку `\inc\` проекта `HelloWorld`. Теперь вы можете подключить с его помощью директивы `#include "StaticLib.h"` к любому файлу исходного кода и обращаться в нем к функции `DummyFunction()`.

Далее необходимо подключить саму статическую библиотеку к исполняемому файлу. Для этого откройте файл `HelloWorld.mmp` и на вкладке **Libraries** в разделе **Static Libraries** щелкните на кнопке **Add**. В появившемся списке доступных библиотек выберите значение **StaticLib.lib**. Если такой библиотеки там не окажется, запишите ее имя вручную и добавьте в файл `HelloWorld.mmp`. В конечном итоге в тексте MMP-файла должна появиться следующая строка.

```
STATICLIBRARY StaticLib.lib
```

Как только статическая библиотека подключена к MMP-файлу, вы можете его компилировать. Библиотеки, собранные для различных целевых платформ, хранятся в разных папках. Следует помнить, что при сборке исполняемого файла для определенной целевой платформы (например, GCCE) потребуется соответствующая версия статической библиотеки. Поэтому она также должна быть предварительно собрана для нужной целевой платформы.

Обычно при распространении не входящих в SDK библиотек предоставляется несколько их вариантов для различных платформ (WINSCW, GCCE, ARMV5). В сопроводительную документацию включается информация об установке, а также защищенных возможностях, декларация которых потребуются при обращении к той или иной функции.

Разделяемые динамические библиотеки (DLL)

Динамические библиотеки создаются аналогично статическим, за тем исключением, что в соответствующем окне мастера `New Symbian OS C++ Project` необходимо будет выбрать шаблон проекта **Basic dynamically linked library**.

Все динамические библиотеки в Symbian OS принято делить по типу предоставляемого интерфейса на статические и полиморфные. Для использования **библиотек со статическим интерфейсом** (static interface dll, не путайте со статическими библиотеками!) к программе подключаются заголовочные файлы, содержащие объявление экспортируемых библиотекой функций и методов. **Библиотеки с полиморфным интерфейсом** (polymorphic interface dll или polymorphic dll) экспортируют только одну функцию, возвращающую указатель на объект конкретного класса, наследованного от определенного абстрактного интерфейса. Таким образом, для работы с полиморфной DLL достаточно располагать заголовочным файлом с объявлением этого интерфейса. С помощью полиморфных DLL удобно реализовывать механизм расширения функциональности за счет подключаемых модулей, и они широко используются в системе.

На одном из шагов мастера вам будет предложено ввести идентификаторы UID2 и UID3 новой библиотеки. Для библиотек со статическим интерфейсом UID2 должен быть равен KSharedLibraryUid (0x1000008D), а UID3 является уникальным идентификатором. Такие DLL всегда статически связываются с исполняемым файлом. Во время запуска приложения происходит загрузка необходимых ему для работы статически связанных библиотек, а также выполняется проверка их идентификаторов UID3. Если библиотека имеет идентификатор, отличный от того, с которым она использовалась для компоновки исполняемого файла, — она не будет загружена, и программа не сможет запуститься.

В полиморфных библиотеках UID2 является идентификатором реализуемого в них интерфейса. Например, для создания DLL-модуля, добавляющего поддержку нового протокола в системный сервер сокетов, необходимо использовать в качестве UID2 значение KUidProtocolModule (0x1000004A) и изменить расширение имени файла библиотеки на .prt. Данные о том, какие именно идентификаторы UID2 нужно использовать для того или иного вида плагинов, содержатся в справочнике SDK. Если вы используете полиморфные DLL в своем приложении, то можете выбрать свое собственное значение UID2. UID3 служит для уникальной идентификации конкретной реализации интерфейса и выбирается разработчиком. Как правило, полиморфные DLL являются динамически связываемыми и загружаются с помощью класса RLibrary во время работы приложения.

При разработке и использовании DLL не следует забывать о правилах платформы безопасности Symbian OS (см. главу 1, раздел “Защищенные возможности в библиотеках”).

С помощью шаблона **Basic dynamically linked library** создается библиотека со статическим интерфейсом и UID2, равным 0x1000008D. Предположим, вы создали проект с именем **DynamicLib**. В этом случае обратите внимание на следующие строки содержащегося в нем MMP-файла.

```
TARGET      DynamicLib.dll
TARGETTYPE  dll
UID         0x1000008D 0xE2EC91A6
```

Как видите, библиотека создается с расширением `.dll`, а не `.lib`. Ключевое слово `TARGETTYPE` определяет значение `UID1` и должно быть `dll`. В полиморфных DLL, реализующих системные интерфейсы для расширения функциональности служб Symbian OS, расширение имени файла и `UID1` могут меняться в соответствии с требованиями, приведенными в документации SDK.

В файле `DynamicLibDllMain.cpp` определены функция для вызова паники и старая точка входа `E32Dll()` для сборки под старые SDK (для совместимости). Функция точки входа `E32Main()`, используемая в Symbian 9.x, отсутствует за ненадобностью, но вы всегда можете ее добавить.

В файлах `DynamicLib.h` (листинг 6.11) и `DynamicLib.cpp` (листинг 6.12) содержатся определение и реализация предоставляемой библиотекой функциональности. Рассмотрим файл заголовка подробнее. Как видите, содержащийся в нем `C`-класс инкапсулирует дескриптор для хранения строки (до 15 символов). В нем, помимо стандартных конструкторов и деструктора, содержатся методы для получения дескриптора-указателя на текущую строку, а также добавления символа в нее и удаления его из строки. Наконец, метод `Version()` возвращает структуру типа `TVersion`, хранящую версию класса (или библиотеки). Обратите внимание на то, что прототипы всех публичных методов содержат выражение `IMPORT_C`. Оно означает, что реализация данной функции находится в статически подключаемой библиотеке. Таким образом, файл `DynamicLib.h` может быть скопирован в проект приложения и использован для доступа к классу. Реализацию функций, не содержащих в объявлении `IMPORT_C`, система будет искать в коде исполняемого файла, а не в библиотеках импорта, что приведет к ошибке при сборке. Встраиваемые (`inline`) функции никогда не помечаются как импортируемые, так как раскрываются во время препроцессинга исходного файла. Для доступа к публичным членам-данным реализованного в DLL класса также не требуется никаких дополнительных маркеров.

Листинг 6.11. Файл `DynamicLib.h`

```
#include <e32base.h> // CBase
#include <e32std.h>  // TBuf

// Константы

const TInt KDynamicLibBufferLength = 15;
typedef TBuf<KDynamicLibBufferLength>
        TDynamicLibExampleString;

// Определение класса

class CDynamicLib : public CBase
```

```

    {
public:
    // Функции для создания
    IMPORT_C static CDynamicLib* NewL();
    IMPORT_C static CDynamicLib* NewLC();
    IMPORT_C ~CDynamicLib();

public:
    // Новые демонстрационные функции
    IMPORT_C TVersion Version() const;
    IMPORT_C void ExampleFuncAddCharL(const TChar& aChar);
    IMPORT_C void ExampleFuncRemoveLast();
    IMPORT_C const TPtrC ExampleFuncString() const;

private:
    // Функции для создания
    CDynamicLib();
    void ConstructL();

private:
    // Данные
    TDynamicLibExampleString* iString;
};

```

Листинг 6.12. Файл DynamicLib.cpp

```

#include "DynamicLib.h" // CDynamicLib
#include "DynamicLib.pan" // Коды паники

// Методы класса

EXPORT_C CDynamicLib* CDynamicLib::NewLC()
{
    CDynamicLib* self = new (ELeave) CDynamicLib;
    CleanupStack::PushL(self);
    self->ConstructL();
    return self;
}

EXPORT_C CDynamicLib* CDynamicLib::NewL()
{
    CDynamicLib* self = CDynamicLib::NewLC();
    CleanupStack::Pop(self);
    return self;
}

CDynamicLib::CDynamicLib()
{

```

```

    }

void CDynamicLib::ConstructL()
{
    // Создание дескриптора-буфера в куче
    iString = new (ELeave) TDynamicLibExampleString;
}

EXPORT_C CDynamicLib::~CDynamicLib()
{
    delete iString;
}

EXPORT_C TVersion CDynamicLib::Version() const
{
    // Номер версии демонстрационного API
    const TInt KMajor = 1;
    const TInt KMinor = 0;
    const TInt KBuild = 1;
    return TVersion(KMajor, KMinor, KBuild);
}

EXPORT_C void CDynamicLib::ExampleFuncAddCharL(const
                                                TChar& aChar)
{
    __ASSERT_ALWAYS(iString != NULL,
                    Panic(EDynamicLibNullPointer));

    if (iString->Length() >= KDynamicLibBufferLength)
    {
        User::Leave(KErrTooBig);
    }

    iString->Append(aChar);
}

EXPORT_C void CDynamicLib::ExampleFuncRemoveLast()
{
    __ASSERT_ALWAYS(iString != NULL,
                    Panic(EDynamicLibNullPointer));

    if (iString->Length() > 0)
    {
        iString->SetLength(iString->Length() - 1);
    }
}

EXPORT_C const TPtrC CDynamicLib::ExampleFuncString() const

```

```

{
    __ASSERT_ALWAYS(iString != NULL,
        Panic(EDynamicLibNullPointer));
    return *iString;
}

```

Файл `DynamicLib.cpp` (см. листинг 6.12) содержит реализацию вышеописанного класса. Реализация всех функций, объявление которых содержало выражение `IMPORT_C`, начинаются с `EXPORT_C`, так они помечаются как экспортируемые и попадают в библиотеку импорта при сборке DLL. Существует правило: все экспортируемые методы и функции должны быть помечены выражением `EXPORT_C` в реализации и `IMPORT_C` в объявлении.



В этом примере дескриптор-буфер создается в куче. Это не имеет под собой серьезных оснований и сделано исключительно в демонстрационных целях. Чтобы создать буферный дескриптор в куче, достаточно объявить его членом C-класса.

После компиляции в соответствующей целевой сборке папке SDK будет создан файл `DynamicLib.dll`, но этого недостаточно. Чтобы использовать полученную DLL в приложении, необходимо сгенерировать для нее библиотеку импорта (import library). Библиотеки импорта содержат информацию об экспортируемых из DLL функциях, их атрибутах и порядке. Они имеют расширение `.lib` (как статические библиотеки), если получены для бинарного интерфейса ABIv1, либо `.dso`, если получены для ABIv2. Разработчику не нужно беспокоиться о корректном выборе расширения для имени файла библиотеки импорта при ее подключении с помощью MMP-файла — пишите всегда `.lib`, утилиты SDK достаточно умны, чтобы использовать вариант `.dso` при необходимости.

Есть два способа получить библиотеку импорта в процессе сборки DLL: простой и правильный. **Простой способ** заключается в использовании ключевого слова `EXPORTUNFROZEN` в MMP-файле. Его можно добавить в текст файла `DynamicLib.mmp` либо установить флажок **Export unfrozen** на вкладке **Options** в окне его редактора. Эти действия указывают сборщику проекта, что вы желаете поместить в библиотеку импорта *все* экспортируемые функции без исключения. Недостаток этого способа в том, что вы не контролируете порядок следования этих функций, а в некоторых случаях это недопустимо. Например, если вы компилируете новую версию уже существующей и используемой в приложениях библиотеки, то вам придется перекомпилировать все подключающие ее исполняемые файлы, так как вы не можете гарантировать того, что в новой версии порядок экспортируемых функций остался прежним. Тем не менее использование флажка **Export unfrozen** может быть оправдано, если вы только приступили к разработке DLL и еще нет необходимости заботиться об обратной бинарной совместимости.

Правильный способ заключается в использовании DEF-файла (см. главу 3, раздел “Заморозка проекта, def-файлы”). Если в проекте еще не создан DEF-файл, то вам необходимо скомпилировать его, а затем заморозить. Для этого

воспользуйтесь командой меню **Project⇒Freeze exports** в главном окне Carbide.c++. Теперь в панели **Project Explorer** вы увидите новую папку с именем бинарного интерфейса (**EABI** или **BWINS**). В ней должен присутствовать полученный DEF-файл. Если в панели **Project Explorer** такая папка не появилась — обновите содержимое панели, нажав <F5>. Если и это не помогло, значит, либо ваша библиотека не экспортирует функций, либо она не была успешно собрана (проверьте наличие ошибок в консоли).

Если в проекте присутствует DEF-файл, то при сборке должна генерироваться библиотека импорта. Но если вы сгенерировали его впервые, то вам необходимо регенерировать сборочные скрипты. О том, как это сделать рассказывалось в *главе 3*, раздел “Очистка проекта”. После этого следует вновь собрать проект. Таким образом, процедура получения первой библиотеки импорта для DLL будет выглядеть так.

1. Сборка проекта (компиляция DLL).
2. Заморозка проекта (создание DEF-файла).
3. Очистка проекта (регенерация всех сборочных скриптов).
4. Сборка проекта (компиляция DLL, создание библиотеки импорта в соответствии с DEF- файлом).

В дальнейшем каждый раз, когда вы удаляете или добавляете в библиотеку DLL новую экспортируемую функцию, необходимо будет выполнить следующее.

1. Сборка проекта (компиляция DLL).
2. Заморозка проекта (обновление DEF-файла).
3. Сборка проекта (компиляция DLL, обновление библиотеки импорта в соответствии с новым DEF-файлом).



Для целевых сборок WINSCW и GCCE\ARMv5 создаются разные DEF-файлы (в папках **BWINS** и **EABI**, соответственно). Постарайтесь не запутаться и не забыть обновить DEF-файл для GCCE после длительной работы с WINSCW.



Внимательно следите за сообщениями в консоли при сборке проекта. В случае, если в библиотеке имеются экспортируемые функции, а в DEF-файле они отсутствуют, или, если сборщик по какой-то причине не генерирует библиотеку импорта, в консоль будут выведены соответствующие предупреждения.

Получить библиотеку импорта для уже скомпилированной DLL, не имея ее исходного кода, невозможно. Поэтому динамические библиотеки, предназначенные для использования в сторонних проектах, распространяются вместе с **.dso-** и **.lib-**файлами. В дистрибутив готового приложения, естественно, библиотеки импорта не включаются.

Теперь, когда мы располагаем и DLL, и библиотекой импорта, мы можем продемонстрировать ее использование в проекте HelloWorld. Для этого скопируйте

файл `DynamicLib.h` в папку `\inc\` проекта `HelloWorld` и подключите в MMP-файле библиотеку импорта `DynamicLib.lib`. В результате в любом файле исходного кода проекта вы теперь сможете обратиться к классу `CDynamicLib`. Пример.

```
#include "DynamicLib.h"

LOCAL_C void MainL()
{
    CDynamicLib* obj = CDynamicLib::NewLC();
    TBuf<10> buf;
    for (TInt i = 0; i < 10; i++)
    {
        buf.Num(i);
        obj->ExampleFuncAddCharL(TChar(buf[0]));
    }

    buf = obj->ExampleFuncString();
    // buf == "0123456789"
    CleanupStack::PopAndDestroy(obj);
}
```



Опытные разработчики предпочитают размещать все необходимое для создания библиотеки в одном проекте с исполняемым файлом (если эту DLL использует только одна программа). Это позволяет получить доступ к коду DLL в отладчике `Carbide.c++` непосредственно из процесса приложения. Для этого необходимо скопировать MMP-файл библиотеки в проект программы, а затем перенести в него содержимое папок `\inc\` и `\src\`. Обычно `Carbide.c++` автоматически предлагает добавить MMP-файл в файл `bld.inf` проекта и задает вопрос, к какому из MMP-файлов проекта следует подключить новые файлы исходного кода. Если этого не произошло, настройте все вручную. Главное — убедитесь, что MMP-файл библиотеки в `bld.inf` упоминается *раньше*, чем MMP-файл использующей ее программы. После этого вам потребуется выполнить все описанные выше действия для получения библиотеки импорта.

Для того чтобы изучить создание и работу с динамически связываемыми библиотеками, мы преобразуем имеющийся у нас проект `DynamicLib`. Для этого достаточно в исходном коде библиотеки удалить все маркеры `IMPORT_C` и `EXPORT_C`, добавив следующую функцию.

```
EXPORT_C CDynamicLib* GetInstance()
{
    return CDynamicLib::NewL();
}
```

Функция `GetInstance()` будет единственной экспортируемой из нашей DLL. Объявлять ее прототип в файле `DynamicLib.h` не следует — мы будем

обращаться к ней по номеру (ординалу). В проекте HelloWorld обновите файл `DynamicLib.h`, удалите библиотеку импорта `Dynamic.lib` из ММР-файла и измените метод `MainL()` следующим образом.

```
#include "DynamicLib.h"

LOCAL_C void MainL()
{
    // Объект для работы с библиотеками
    RLibrary lib;
    // Загрузка динамически связываемой библиотеки
    User::LeaveIfError(lib.Load(_L("DynamicLib.dll"));
    CleanupClosePushL(lib);

    // Получение функции с ординалом 1 (GetInstance)
    TLibraryFunction func = lib.Lookup(1);
    User::LeaveIfNull((TAny*) func);
    // Получение результата функции
    CDynamicLib* obj = (CDynamicLib*) func();

    TBuf<10> buf;
    for (TInt i = 0; i < 10; i++)
    {
        buf.Num(i);
        obj->ExampleFuncAddCharL(TChar(buf[0]));
    }

    buf = obj->ExampleFuncString();
    // buf = "0123456789"
    CleanupStack::PopAndDestroy(2, &lib);
}
```

Не правда ли, все просто? Чтобы сделать нашу библиотеку действительно полиморфной, остается лишь переделать объявление класса `CDynamicLib` с использованием подходящего М-класса в качестве интерфейса или базового С-класса.

```
// Определение интерфейса
class CMyLib: public CBase
{
public:
    virtual TVersion Version() const = 0;
    virtual void ExampleFuncAddCharL(const TChar& aChar) = 0;
    virtual void ExampleFuncRemoveLast() = 0;
    virtual const TPtrC ExampleFuncString() const = 0;
};

class CDynamicLib : public CMyLib
{
```

```
<...> // Здесь объявление CDynamicLib
}
```

Желательно также придумать уникальный идентификатор для нашего интерфейса и использовать его в качестве UID2.

Теперь в проекте HelloWorld мы можем заменить класс CDynamicLib на CMyLib и вообще убрать объявление класса CDynamicLib. Я использовал порожденный от класса CBase C-класс вместо M-класса, для того чтобы не переделять работу со стеком очистки в приложении. Такой подход довольно удобен, так как позволяет создать несколько DLL с различной реализацией базового класса CMyLib и использовать их в программе, не имея каких-либо заголовочных файлов этих библиотек. При этом библиотеку для нашей программы могут создать даже сторонние разработчики, достаточно предоставить им описание класса CMyLib. Именно так в пользовательских приложениях реализуются простейшие плагины.

Динамическая загрузка библиотек осуществляется с помощью класса RLibrary. Он предоставляет ряд перегруженных методов Load() для открытия, загрузки и подключения DLL-файла, а также метод Close() для его выгрузки. Для определения файла, который должен быть загружен, необходимо указать его имя. Разработчик может дополнительно передать путь (или список путей), по которому следует искать файл, а также значения идентификаторов, проверяемых при загрузке. Примеры.

```
// Загрузка DynamicLib.dll, поиск в стандартных папках
lib.Load(_L("DynamicLib.dll"));

// Загрузка DynamicLib.dll, поиск в \sys\bin\ на всех дисках
lib.Load(_L("DynamicLib.dll"), _L("\\sys\\bin"));

// Загрузка DynamicLib.dll, поиск в c:\sys\bin\ и e:\sys\bin\
lib.Load(_L("DynamicLib.dll"),
        _L("c:\\sys\\bin;e:\\sys\\bin"));

// Задание трех идентификаторов библиотеки
// KNullUid означает любое значение
const TUid KMyInterfaceUid = {0x1000008D};
TUidType uids(KNullUid, KMyInterfaceUid, KNullUid);
// Загрузка DynamicLib.dll с проверкой идентификаторов
lib.Load(_L("DynamicLib.dll"), uids);
```

Так как библиотеки на ROM-диске хранятся несколько отличным способом, для загрузки DLL с диска Z:\ существует отдельный метод LoadRomLibrary().

После того как нужная библиотека была найдена и загружена, разработчик может получить указатель на одну из ее экспортируемых функций. Для этого служит метод Lookup(), аргументом которого является номер (ординал)

функции. Нумерация функций начинается с единицы. Метод `Lookup()` возвращает указатель на функцию без аргументов, результатом которой является `TInt` (см. определение типа `TLibraryFunction`). Если вы знаете, что прототип экспортируемой с данным номером функции выглядит иначе, выполните необходимое приведение типов.

Другие методы класса `RLibrary`:

- `FileName()` — служит для получения имени загруженной библиотеки;
- `Type()` — возвращает три идентификатора `UID` загруженной библиотеки;
- `GetRamSizes()` — позволяет получить текущий объем памяти, занимаемый кодом и неизменяемыми данными библиотеки, при этом для `DLL`, находящейся в `ROM`, всегда возвращаются нулевые результаты.

Изменяемые глобальные данные в `DLL`

Под **изменяемыми глобальными данными** (`modifiable global data`) понимаются любые непостоянные глобально определенные переменные, непостоянные статические переменные, определенные в функциях, а также константы с нетривиальным конструктором. В документации по Symbian OS их чаще называют **WSD** (`writable static data`).

```
TBufC<20> fileName; // Изменяемые глобальные данные

const TPoint point(0, 1); // Константы с конструктором
static const TChar KExclamation('!');

void SomeFunction()
{
    static TInt iCount; // Статические переменные
    <...>
}
```

Поддержка `WSD` в разделяемых библиотеках появилась лишь в ядре `EKA2` (Symbian 8.0b, Symbian 8.1b, Symbian 9.x) для облегчения портирования кода с других платформ. Чтобы включить ее, необходимо добавить ключевое слово `EPOCALLOWDLLDATA` в `MMP`-файл библиотеки. При компиляции под целевую платформу `WINSCW`, а также при сборке для устройства в некоторых версиях `SDK` этот режим включен по умолчанию.

Использовать `WSD` не рекомендуется, так как на это тратится слишком большое количество ресурсов. Существует также множество ограничений на использование библиотек с `WSD` на устройстве и в эмуляторе. Например, в эмуляторе такая библиотека может быть загружена не более чем одним приложением в каждый момент времени. Более подробно обо всех нюансах реализации и использования библиотек с `WSD`-данными вы можете узнать из документации, приведенной в конце книги.

Рекомендуется избегать использования WSD при создании DLL и заменять их при портировании существующих библиотек. Для этого используются следующие приемы:

- перемещение глобальных переменных в классы (там, где это возможно);
- использование статически связываемых библиотек вместо DLL;
- размещение всех WSD-данных в одном классе, экземпляр которого создается при инициализации библиотеки, а указатель на него помещается в Thread Local Storage (TLS);
- использование клиент-серверной архитектуры.

Более подробную информацию о них вы можете почерпнуть из дополнительной документации.

Подготовка к сертификации ASD

- Знание и понимание характеристик статических и полиморфных DLL.
 - Знание того, что значения UID2 используются для различия статических и полиморфных DLL, а также между различными типами плагинов.
 - Понимание того, что функции разделяемых библиотек должны экспортироваться, чтобы быть доступными для использования другими исполняемыми файлами.
 - Знание того, что Symbian OS не позволяет выполнять поиск функции по имени в динамически связываемых библиотеках — только по ординалу.
 - Знание того, что WSD не поддерживается в DLL, используемых в системах на ядре EKA1, и не рекомендуется на ядре EKA2.
 - Знание базовых приемов при портировании, позволяющих избежать использования WSD.
-

Работа с процессами и потоками

Как и в любой другой современной операционной системе, в Symbian OS существуют процессы и потоки. В потоках выполняется код приложений. Для обеспечения многозадачности их выполнение планируется системой на основании приоритетов. Каждый поток имеет некоторый приоритет, заданный числом. Процесс является контейнером потоков и обеспечивает безопасность их исполнения. Прямой доступ к памяти одного процесса из другого невозможен. Обмен информацией между ними осуществляется с помощью специальных механизмов. Процессы также имеют приоритет, но он служит лишь для вычисления абсолютных приоритетов содержащихся в них потоков.

При запуске приложения создается новый процесс и один поток в нем. Этот поток называется главным, и завершение его выполнения означает окончание выполнения процесса. В главном потоке могут создаваться дополнительные по-

токи — их максимальное число ограничено размером доступной процессу памяти (128 потоков либо 13 потоков с собственной кучей)⁴.

Symbian C++ предоставляет разработчику ряд классов для работы с процессами и потоками. Мы подробно рассмотрим их использование, так как без этого изучение IPC и клиент-серверной архитектуры не возможно.

Работа с процессом осуществляется с помощью класса `RProcess`. При создании экземпляра `RProcess` он автоматически подключается к текущему процессу приложения. Его хендл в этот момент равен `KCurrentProcessHandle`. Таким образом, разработчик может управлять работой собственной программы. Помимо этого, класс `RProcess` позволяет подключиться к процессу другого приложения с помощью метода `Open()`. Каждый процесс в системе имеет числовой идентификатор и символьное имя. Имя процесса по умолчанию совпадает с именем его исполняемого файла, но может быть изменено программно. Следующий пример демонстрирует открытие процесса по имени.

```
RProcess pr; // Сейчас можно работать с текущим процессом
if (pr.Open(_L("MyProcessName")) == KErrNone)
{
    // Процесс с именем MyProcessName существует и открыт
    <...> // Работа с процессом MyProcessName
}
```

Идентификатор назначается процессу автоматически при создании и является 64-битовым числом. Для хранения идентификатора используется простой класс `TProcessId`. Следующий пример демонстрирует открытие процесса по идентификатору.

```
TProcessId id(uid); // Значение uid необходимо знать заранее
if (pr.Open(id) == KErrNone)
{
    // Процесс с идентификатором uid существует и открыт
    <...> // Работа с процессом MyProcessName
}
```

Естественно, в большинстве случаев значение идентификатора процесса разработчику неизвестно. Точное написание имени также может вызывать сомнения. В этом случае процесс может быть найден по шаблону, которому должно соответствовать его имя с помощью класса `TFindProcess`.

```
TFindProcess fp(_L("MyProc*"));
TFullName proc_name; // Дескриптор-буфер
if (fp.Next(proc_name) == KErrNone)
{
    // В proc_name имя, соответствующее шаблону
    // Открываем процесс по имени
    User::LeaveIfError(pr.Open(proc_name));
}
```

⁴ FAQ-1392

В конструктор класса `TFindProcess` передается шаблон, которому должно соответствовать имя процесса (по умолчанию равен `"*"`). Этот шаблон также можно задать с помощью метода `Find()`. Метод `Next()` служит для нахождения следующего подходящего имени процесса. Их может быть несколько, но в вышеприведенном примере мы надеемся, что в системе есть всего один соответствующий шаблону процесс либо он первый среди всех подходящих процессов. Единственным аргументом метода `Next()` является дескриптор, в который помещается найденное полное имя процесса. Если новое удовлетворяющее условиям имя не может быть найдено, то результатом метода будет код ошибки, а в дескриптор будет помещена пустая строка. Перебрать все запущенные в системе процессы можно следующим образом.

```
TFindProcess fp;
TFullName proc_name; // Дескриптор-буфер
while (fp.Next(proc_name) == KErrNone)
{
    // Открываем следующий процесс
    // Или просто выводим его имя
}
```

Все варианты метода `Open()` имеют второй аргумент типа `TOwnerType`, представляющий собой следующее перечисление:

- `EOwnerProcess` (по умолчанию) — хендл открытого процесса может быть использован во всех потоках приложения;
- `EOwnerThread` — хендл открытого процесса может быть использован только в текущем потоке приложения.

Помимо метода `Open()`, класс `RProcess` предоставляет метод `Create()`, позволяющий запустить исполняемый файл, а затем открыть его процесс. Пример.

```
pr.Create( _L("MyApp"), KNullDesC );
```

Первым аргументом является полное имя исполняемого файла. Если опущено расширение — оно полагается равным `.exe`. Если отсутствует путь к файлу — выполняется его поиск в стандартных каталогах. Вторым аргументом задает параметр, передаваемый в главный поток запускаемого процесса. Последним аргументом является объект типа `TOwnerType`, назначение которого такое же, как и в методе `Open()`. Мы вновь позволили ему принять значение по умолчанию. Существует также вариант метода `Create()`, позволяющий указать значения, которым должны соответствовать идентификаторы `UID` исполняемого файла (используются так же, как в методе `Load()` класса `RLibrary`).

Теперь, когда нам известно множество способов подключения объекта `RProcess` к процессу, пришло время подробно рассмотреть операции, которые можно с ним выполнять.

- `FileName()` — получение полного имени исполняемого файла, породившего процесс.
- `Id()` — идентификатор `TProcessId` процесса.
- `RenameMe()` — позволяет задать имя процесса.
- `GetMemoryInfo()` — получение размера и адресов расположения в памяти различных секций (кода, данных и пр.).
- `HasCapability()` — проверка наличия у процесса доступа к определенной защищенной возможности.
- `Type()` — получение значений трех идентификаторов `UID` исполняемого файла процесса.
- `SecureId()` — получение идентификатора `SID` процесса.
- `VendorId()` — получение идентификатора `VID` процесса.
- `Priority()` и `SetPriority()` — получение и изменение приоритета процесса. Разработчик может установить приоритет процесса равным `EPriorityLow`, `EPriorityBackground`, `EPriorityForeground` либо `EPriorityHigh`. Попытка изменить его на `EPriorityWindowServer`, `EPriorityFileServer`, `EPriorityRealTimeServer` или `EPrioritySupervisor` приведет к панике — столь высокие значения может задавать только система. При изменении приоритета процесса абсолютные значения всех входящих в него потоков пересчитываются.
- `Resume()` — помечает главный поток процесса как доступный для исполнения (используется в случае, если выполнение потока было остановлено).
- `Rendezvous()` — см. ниже.
- `RendezvousCancel()` — см. ниже.
- `Kill()` — завершение работы процесса. Для использования требуется доступ к защищенной возможности `PowerMgmt`.
- `Terminate()` — завершение работы процесса. Используется вместо `Kill()`, чтобы подчеркнуть, что процесс прекращает работу аварийно. Для использования требуется доступ к защищенной возможности `PowerMgmt`.
- `Panic()` — вызывает в процессе панику с указанной категорией и кодом ошибки. Для использования требуется доступ к защищенной возможности `PowerMgmt`.
- `ExitType()` — позволяет узнать, завершил ли работу процесс, и, если это так, причину его завершения (`Kill`, `Terminate` или `Panic`).
- `ExitCategory()` — возвращает категорию паники, в результате которой был завершен процесс.
- `ExitReason()` — возвращает код паники, в результате которой был завершен процесс.

Класс `RProcess` предоставляет асинхронную функцию `Logon()`, с помощью которой можно получить уведомление о завершении (нормальном или аварийном) работы процесса. В объекте `TRequestStatus` возвращается тип

завершения процесса (его также можно получить из метода `ExitType()`). Метод `LogonCancel()` позволяет отменить запущенный с ее помощью асинхронный запрос. Следующий код демонстрирует синхронное использование данных методов.

```
RProcess pr;
// Подключение к стороннему процессу с помощью Open()
// или Create()
<...>
CleanupClosePushL(pr);
TRequestStatus rs;
pr.Logon(rs);
// Синхронно ждем завершения асинхронного запроса
User::WaitForRequest(rs);
// В rs должен быть код завершения процесса
if (rs != KErrCancel && rs != KErrNoMemory)
{
    switch(pr.ExitType())
    {
        case EExitKill:
            // Завершен с помощью Kill()
            break;

        case EExitTerminate:
            // Завершен с помощью Terminate()
            break;

        case EExitPanic:
            {// Завершен из-за возникшей паники
                TBuf<50> buf;
                buf = pr.ExitCategory();
                buf.Append(_L(" "));
                buf.AppendNum(rs.Int());
                // Или AppendNum(pr.ExitReason());
                // В buf вся информация о панике,
                // например "KERN-EXEC 3"
            }
            break;

        default:
            {
                // Не завершился? Что-то тут не так
                User::Panic(_L("Something wrong"), rs.Int());
            }
    }
}
CleanupStack::PopAndDestroy(&pr);
```

Еще одним асинхронным методом класса `RProcess` является `Rendezvous()`. Посланный с его помощью запрос возвращается либо при завершении процесса, либо если в нем вызван перегруженный метод `Rendezvous()` с целочисленным аргументом. Таким образом процесс уведомляет о достижении какого-то определенного состояния. `Rendezvous` можно перевести с английского как “рандеву, встреча”. В этих терминах работу методов `Rendezvous()` можно представить следующим образом. Один или несколько процессов подключаются к процессу *X*, используя объекты класса `RProcess`, и посылают с помощью метода `Rendezvous()` асинхронный запрос на уведомление о том, когда процесс *X* будет готов к встрече. В процессе *X* в определенный момент выполняется следующий код.

```
RProcess pr; // Подключен к текущему процессу
const TInt KReason(200);
pr.Rendezvous(KReason); // Готов к встрече
```

Подобным образом процесс *X* уведомляет всех интересующихся (если они есть), что к встрече он готов. В этот момент все асинхронные запросы `Rendezvous()` завершаются с кодом `KReason`. Асинхронный запрос может быть также отменен из вызвавшего его процесса с помощью метода `RendezvousCancel()`.

Использование метода `Rendezvous()` можно часто наблюдать в приложениях клиент-серверной архитектуры. С его помощью, запустив сервер, клиент ожидает уведомления о том, что сервер закончил инициализацию инфраструктурных объектов и готов к установлению рабочей сессии. Более подробно мы рассмотрим эту процедуру в следующем разделе.

При вызове метода `Close()` закрывается только хендл на процесс. На работу самого процесса это никак не влияет.

Работа с потоками в Symbian C++ ведется с помощью класса `RThread` и во многом аналогична работе с процессами. При создании объекта `RThread` он автоматически подключается к текущему потоку. Для подключения к другим потокам необходимо воспользоваться методом `Open()`. Как и процессы, поток определяется либо по имени, либо по уникальному идентификатору. Если имя не известно, найти нужный поток можно с помощью класса `TFindThread`. Работа с `TFindThread` ведется точно так же, как и с классом `TFindProcess`.

Создать новый поток в процессе можно с помощью метода `Create()` следующим образом.

```
// Главная функция нового потока
TInt ThreadFunction(TAny* aParams)
{
    <...> // Код, выполняющийся в отдельном потоке
    return KErrNone;
}
```

```

    }

    // Запуск нового потока
    void StartNewThread()
    {
        RThread thread;
        const TInt KStackSize = 0x8000; // 8 Кб - стек
        User::LeaveIfError(thread.Create(
            _L("UniqueThreadName"), // Имя потока
            ThreadFunction, // Главная функция потока
            KStackSize, // Размер стека потока
            NULL, // Используем кучу текущего потока
            NULL // Аргумент главной функции
        ));

        thread.Resume(); //Поток помечен как готовый для выполнения
        thread.Close();
    }

```

Первым аргументом метода `Create()` является имя создаваемого потока. Оно должно быть уникальным, в противном случае вы получите ошибку `KErrAlreadyExists`. Затем указывается главная функция потока: с ее вызова начинается его выполнение. Функция должна иметь такой вид.

```
TInt foo(TAny*);
```

Затем задаются размеры стека и кучи потока (как вы помните, для главного потока программы они определяются в ММР-файле проекта). Размер стека обычно равен 8 Кбайт, но для работы с элементами пользовательского интерфейса `Avkon` может потребоваться до 20 Кбайт. На практике GUI всегда создают в главном потоке, поэтому 8 Кбайт вам должно вполне хватить. Размеры кучи указываются несколько сложнее. Класс `RThread` содержит два варианта метода `Create()`. В одном из них достаточно указать минимальный и максимальный размер памяти кучи потока. Его следует использовать всегда, когда вы хотите создать поток с собственной кучей. Во втором необходимо передать указатель на объект класса `RAllocator`, предоставляющий доступ и средства управления кучей. Мы не будем рассматривать работу с классом `RAllocator` в рамках этой книги. Если возникает необходимость создать поток, разделяющий память кучи с другим потоком, то чаще всего им является текущий поток. Для того чтобы новый поток использовал память кучи текущего потока, достаточно передать `NULL` вместо указателя на объект `RAllocator` в методе `Create()`.

Следующий аргумент метода `Create()` позволяет передать в новый поток указатель через единственный параметр его главной функции. Как и в методах `RProcess::Open()`, `RProcess::Create()`, у метода `RThread::Open()` последним аргументом является объект типа `TOwnerType`, определяющий область использования хендла потока. В вышеприведенном примере он опущен.

По завершению работы метода `Create()` новый поток уже создан, но не запущен. В этот момент вы можете, например, подписаться на уведомление о за-

вершении его работы с помощью метода `Logon()` или изменить его приоритет. Для того чтобы запустить поток, необходимо пометить его как готовый к исполнению с помощью метода `Resume()`. После этого поток может вызваться планировщиком Symbian OS, когда ему в соответствии с приоритетом будет выделено процессорное время. Остановить выполнение потока можно с помощью метода `Suspend()`. Класс `RProcess` не содержит методов `Resume()` и `Suspend()`, так как процесс не является единицей планирования в механизме многозадачности системы.

Класс `RThread` предоставляет для работы с потоком все те же методы, что и `RProcess`, а также некоторые дополнительные.

- `Resume()` — помечает поток как готовый к выполнению.
- `Suspend()` — приостановка выполнения потока. Поток будет игнорироваться системным планировщиком до вызова метода `Resume()`.
- `HandleCount()` — возвращает количество открытых хендлов в текущем потоке и процессе.
- `StackInfo()` — получение информации об используемом потоком стеке.
- `RequestCount()` — позволяет получить значение семафора, используемого планировщиком активных объектов. Отрицательная величина означает, что поток ожидает завершения как минимум одного асинхронного запроса.
- `RequestComplete()` — сигнализирует о завершении асинхронного запроса через переданный по указателю объект `TRequestStatus`. Объект `RThread` должен быть подключен к текущему потоку. Применяется поставщиками асинхронных сервисов, используется в случае, если подписчик принадлежит другому потоку или процессу.
- `RequestSignal()` — позволяет послать сигнал через семафор потока, аналогично `RequestComplete()` с `NULL`-указателем на `TRequestStatus`. Может применяться только для потоков в текущем процессе.



Асинхронный метод `Logon()` обеспечивает отслеживание завершения работы только одного потока. Для того чтобы наблюдать сразу за всеми потоками системы, в Symbian имеется класс `RUndertaker`. С его помощью можно реализовать аналог включаемых файлом `errrd`-уведомлений или утилиту для ведения журнала сбоев в системе.

Метод `Close()` закрывает лишь хендл на поток, после чего для продолжения работы с объектом класса `RThread` необходимо вновь открыть существующий или создать новый поток. Поток завершается в тот момент, когда вся программная логика в нем выполнена либо он остановлен с помощью методов `Kill()`, `Terminate()` или `Panic()`.



Завершение работы потока требует некоторого времени. В этот период система освобождает выделенные ему ресурсы, закрывает все установленные им сессии и рассылает уведомления о его завершении (асинхронные запросы метода `Logon()`). Поэтому, если вы завершаете, а затем

вновь создаете поток с тем же именем, не забывайте, что он все еще может быть зарегистрирован в системе. В этом случае вы получите ошибку `KErrAlreadyExists`.

Синхронизация потоков

Главной проблемой разработчика при работе с несколькими потоками является их синхронизация, — например, для разграничения доступа к общим ресурсам. Для этого в Symbian C++ имеются стандартные объекты синхронизации, возможно, знакомые вам по другим языкам программирования. Все они создаются на стороне ядра системы и могут использоваться для управления потоками, принадлежащими различным процессам. Разработчику доступны мьютексы (mutex), условные переменные (condition variable), семафоры (semaphore) и критические секции (critical section).

Мьютексы

Мьютекс (от mutual exclusion — взаимное исключение) — это простой объект, который может принадлежать *только одному* потоку. Он позволяет потокам выполнять над собой лишь две операции — захват и освобождение, а также содержит счетчик, равный нулю, когда мьютекс свободен, и больше нуля, когда мьютекс занят. Захват увеличивает значение счетчика, освобождение — уменьшает. Свободный мьютекс может захватить любой поток, после чего он может увеличивать (сколько угодно) и уменьшать (до нуля) его счетчик. Но если мьютекс уже захвачен одним потоком, то любой другой поток, попытавшийся завладеть им, будет приостановлен до тех пор, пока мьютекс вновь не станет свободным. Если освобождения мьютекса дожидаются несколько потоков, то, как только его счетчик станет нулевым, им завладеет поток с наивысшим приоритетом.

Работа с мьютексами в Symbian C++ реализуется с помощью класса `RMutex`. Он предоставляет два метода для создания нового мьютекса. С помощью `CreateGlobal()` создается именованный мьютекс, доступ к которому может получить любой поток. Для этого достаточно найти его по имени, используя объект класса `TFindMutex`. Работа с `TFindMutex` абсолютно аналогична работе с вышеописанными классами `TFindProcess` и `TFindThread`, поэтому подробно рассматривать ее мы не будем.

Второй метод для создания мьютекса — `CreateLocal()`, он позволяет получить неименованный мьютекс, доступ к которому возможен только по хендлу и только из потоков текущего процесса.

Оба создающих мьютекс метода имеют аргумент типа `TOwnerType`, позволяющий определить владельца открытого объекта `RMutex`. Значениями могут быть:

- `EOwnerProcess` (по умолчанию) — объект может быть использован во всех потоках приложения;
- `EOwnerThread` — объект может быть использован только в текущем потоке приложения.

Для того чтобы открыть существующий мьютекс по заданному имени, необходимо воспользоваться методом `Open()`. Как только мьютекс открыт, разработчик может воспользоваться следующими методами.

- `IsHeld()` — проверяет, принадлежит ли мьютекс текущему потоку.
- `Wait()` — захват мьютекса, счетчик увеличивается. Если мьютекс не свободен и не принадлежит текущему потоку (`IsHeld() == EFalse`), то метод вернет управление только тогда, когда мьютекс будет освобожден другим потоком, и им удастся завладеть (учитывая приоритеты всех ожидающих потоков). Если мьютекс свободен или уже принадлежит текущему потоку (`IsHeld() == ETrue`), то метод `Wait()` завершается сразу же.
- `Signal()` — освобождение мьютекса, счетчик уменьшается. Если счетчик стал нулевым — мьютекс становится свободным, а метод `IsHeld()` начинает возвращать значение `EFalse`.

Не забывайте закрывать объекты `RMutex` с помощью `Close()`. Мьютекс будет удален системой, когда будет закрыт последний подключенный к нему экземпляр объекта `RMutex`. Небольшой пример.

```
_LIT(KMutexName, "MyTestMutex");

TFindMutex find(KMutexName);
RMutex mutex;

if (mutex.Open(find) != KErrNone)
    User::LeaveIfError(mutex.CreateGlobal(KMutexName));
CleanupClosePushL(mutex);

mutex.Wait();
<...> // Доступ к разделяемому ресурсу
mutex.Signal();

CleanupStack::PopAndDestroy(&mutex);
```

Условные переменные

Условные переменные используются вместе с мьютексами и позволяют потоку заблокировать свое исполнение до наступления определенных условий. Они реализуются классом `RCondVar` и, подобно мьютексам, могут создаваться глобально (для доступа из любого процесса) или локально (для доступа лишь из потоков текущего процесса). Класа для поиска условных переменных нет, поэтому разработчик должен знать точное имя глобальной условной переменной, чтобы подключиться к ней.

В классе `RCondVar` определены следующие методы.

- `Wait()` — ожидание выполнения условия. В качестве аргумента передается мьютекс, захваченный текущим потоком. Этот мьютекс освобождается, и выполнение потока приостанавливается, до тех пор пока не будет вызван метод

`RMutex::Signal()`. После чего переданный в качестве аргумента мьютекс захватывается вновь и метод возвращает управление.

- `TimedWait()` — аналогичен методу `Wait()`, но позволяет задать интервал времени, по истечению которого поток прекращает ждать сигнал от условной переменной.
- `Signal()` — сигнализирует условной переменной о том, что необходимые условия могли наступить. Наиболее приоритетный из ожидающих этого события потоков помечается как готовый к выполнению.
- `Broadcast()` — аналогичен методу `Signal()`, но позволяет разблокировать сразу все ожидающие сигнала условной переменной потоки.

Разобраться с принципами работы условных переменных лучше всего на практике. Предположим, что нам необходимо синхронизировать работу двух потоков. Оба они используют однонаправленную очередь сообщений, доступ к которой может получить лишь один поток в каждый момент времени. Очевидно, что работу с очередью сообщений следует синхронизировать с помощью мьютекса. Поставим перед собой задачу оптимизировать чтение сообщений из очереди. Для этого считывающий сообщения поток (клиент) будет ожидать до тех пор, пока в очереди не появится, по крайней мере, n сообщений, после чего считает сразу все. Остановку клиента мы реализуем с помощью условной переменной. Пример.

```
/*
 * Вспомогательная процедура для инициализации объектов
 */

void InitMutexAndCondVarL(RMutex& aMutex, RCondVar& aCondVar)
{
    _LIT(KMyMutexText, "MyQueueMutex");
    if (aMutex.OpenGlobal(KMyMutexText) != KErrNone)
    {
        User::LeaveIfError(aMutex.CreateGlobal(KMyMutexText));
    }

    _LIT(KMyCondText, "MyQueueEvent");
    if (aCondVar.OpenGlobal(KMyCondText) != KErrNone)
    {
        User::LeaveIfError(aCondVar.CreateGlobal(KMyCondText));
    }
}

/*
 * Пример инициализации объектов
 */

RMutex mutex;
RCondVar condvar;
```

```

InitMutexAndCondVarL(mutex, condvar);

/*
 * Функция для чтения сообщений на стороне клиента
 */

void ReadMessagesL()
{
    mutex.Wait(); // Получаем исключительный доступ к очереди

    // Проверка всегда защищена мьютексом
    while (!<очередь содержит n сообщений>)
    {
        condvar.Wait(mutex);
        // 1. Освобождаем мьютекс и ожидаем сигнала
        // 2. После сигнала мьютекс захватывается снова
        // 3. Проверяем условие while,
        //     если оно не выполнено - возвращаемся к шагу 1
    }

    // В очереди могут быть n или более сообщений
    <считываем все сообщения из очереди>

    // Освобождение мьютекса
    mutex.Signal();
}

/*
 * Функция для записи 1 сообщения на стороне сервера
 */

void WriteOneMessage()
{
    mutex.Wait(); // Получаем исключительный доступ к очереди

    <Пишем одно сообщение в очередь>

    // Посылаем сигнал о том, что условие могло измениться
    condvar.Signal();

    // Освобождаем мьютекс
    mutex.Signal();
}

```

Рассмотрим подробнее вышеприведенный исходный код. Функция `InitMutexAndCondVarL()` позволяет создать или открыть мьютекс и условную переменную. Она должна использоваться как на стороне клиентского, так и на стороне серверного потока. Благодаря этому мы можем быть уверены, что

подключились к нужным именованным объектам. В тот момент, когда клиент вызывает метод `ReadMessagesL()`, ограничивающий доступ к очереди мьютекс захватывается клиентским потоком. После этого можно проверить количество сообщений в ней. Предположим что их недостаточно много — в этом случае происходит вызов метода `Wait()` условной переменной. Переданный ему в качестве аргумента мьютекс немедленно освобождается, выполнение потока приостанавливается, но работа метода не завершается до тех пор, пока для данной условной переменной не будет вызван метод `Signal()`. Спустя некоторое время серверный поток обратится к методу `WriteOneMessage()` для записи очередного сообщения в очередь. Для этого он захватывает мьютекс, выполняет необходимые операции для записи сообщения, вызывает метод `Signal()` условной переменной и освобождает мьютекс. При вызове метода `Signal()` клиентский поток будет помечен как готовый к исполнению. Как только ему будет отведено процессорное время, продолжится работа метода `Wait()` условной переменной, и в нем будет предпринята попытка вновь захватить мьютекс. После получения мьютекса работа метода `Wait()` будет завершена, и клиентский поток вновь сможет проверить, достаточно ли сообщений накопилось в очереди. Заметьте, что метод `Signal()` вызывается при каждой записи сообщения, так как мы исходим из того, что сервер не имеет информации о значении n , более того, значение n может меняться клиентом на основе некоторых критериев (например, очередь заполняется слишком медленно). Если проверяемые условия не выполнены — клиентский поток вновь переходит к ожиданию их изменения с помощью условной переменной. В противном случае происходит чтение всех сообщений из очереди и освобождение мьютекса.

Семафор

Семафор во многом схож с мьютексом и отличается лишь тем, что позволяет не одному, а сразу нескольким потокам (заранее известному количеству) обратиться к нему и не быть заблокированными. Семафор содержит счетчик, который инициализируется при создании некоторым положительным числом n . Он так же как и мьютекс, допускает две операции: захват и освобождение, но на этот раз захват уменьшает, а освобождение увеличивает значение счетчика семафора. До тех пор, пока значение счетчика не отрицательно, подключенные к семафору потоки могут уменьшать его и не быть заблокированными. Таким образом, максимум n потоков могут одновременно пройти семафор. Если счетчик стал отрицательным, все попытавшиеся захватить его потоки приостанавливаются и помещаются в очередь. При освобождении семафора, в случае, если значение его счетчика было отрицательным, и независимо от того, каким оно стало, из очереди извлекается и разблокируется один поток с наибольшим приоритетом.

Работа с семафором ведется с помощью класса `RSemaphore` и очень похожа на работу с мьютексом. Отличие заключается в следующем:

- при создании разработчик должен указать положительное число для инициализации счетчика семафора;
- метод `Wait()` имеет перегруженный вариант, позволяющий задать интервал времени, на протяжении которого поток будет ожидать освобождения семафора;
- метод `Signal()` имеет перегруженный вариант, позволяющий задать величину, на которую увеличится счетчик семафора;
- поиск именованных семафоров осуществляется с помощью класса `TFindSemaphore`, используемого аналогично классу `TFindMutex`.

Пример.

```
_LIT(KSemaphoreName, "MyTestSemaphore");
const TInt KCount = 2;
const TInt KTimeOut = 10000000; //10 сек.

TFindSemaphore find(KSemaphoreName);
RSemaphore sem;

if (sem.Open(find) != KErrNone)
    // Создаем семафор с n = 2
    User::LeaveIfError(sem.CreateGlobal(KSemaphoreName,
                                      KCount));
CleanupClosePushL(sem);

// Если поток вынужден ожидать освобождения
// семафора более 10 сек. - произойдет сброс
User::LeaveIfError(sem.Wait(KTimeOut));
<...> // Доступ к разделяемому ресурсу
sem.Signal();

CleanupStack::PopAndDestroy(&sem);
```

Критическая секция

Как и мьютекс, **критическая секция** позволяет обеспечить исключительный доступ нескольких потоков к одному разделяемому ресурсу. Но создаваться и использоваться критические секции могут только потоками одного процесса — это позволяет обеспечить некоторую оптимизацию производительности (минимизация обращений к ядру, подобно фьютексам в Linux). Управление критическими секциями в Symbian C++ выполняется с помощью класса `RCriticalSection`, порожденного от `RSemaphore`. Работа с ним ведется точно так же, как и с локальными мьютексами, за исключением того, что вместо метода `IsHeld()`, использовавшегося для проверки принадлежности мьютекса текущему потоку, класс `RCriticalSection` предлагает метод `IsBlocked()`, возвращающий значение `ETrue`, если какой-либо поток уже находится в критической секции.

Так как критические секции создаются неименованными, найти и открыть новый хендл на них невозможно. Поэтому потоки должны либо использовать один объект `RCriticalSection`, либо несколько объектов с одним хендлом (создаются с помощью копирующего конструктора).



За рамками нашего обсуждения примитивов синхронизации остался класс `RFastLock`, реализующий так называемый “быстрый семафор”.

Межпоточное взаимодействие

Межпоточное взаимодействие (Inter Thread Communication, или ИТС) — механизм передачи данных из одного потока в другой. В случае если потоки являются частью одного процесса, то они находятся в одном адресном пространстве и могут осуществлять доступ к объектам друг друга по указателям. Обычно указатель на объект, служащий для обмена информацией между потоками, передается в качестве аргумента главной функции нового потока.

Гораздо сложнее механизмы, которые могут использоваться для передачи данных между потоками в различных процессах. Такой вид взаимодействия называется межпроцессным и рассматривается в следующем разделе. Следует помнить, что все межпроцессные механизмы также являются и межпоточными, но обратное неверно.

Подготовка к сертификации ASD

- Умение определять правильность утверждений о потоках и процессах в Symbian OS.
 - Знание роли и характеристик примитивов синхронизации `RMutex`, `RCriticalSection` и `RSemaphore`.
-

Межпроцессное взаимодействие

Как вы знаете, прямой доступ к памяти одного процесса из другого в Symbian OS невозможен, тем не менее он очень часто бывает необходим. Для передачи данных между потоками различных процессов и уведомлений о событиях используются специальные механизмы. Их часто обобщенно называют методами **межпроцессного взаимодействия** (Inter Process Communication или IPC).

Разделяемые области памяти

Наиболее простой и примитивный способ обмена данными между потоками — выделение области памяти, к которой они оба имеют доступ. Такой подход часто используется при создании драйверов и приложений, требующих высокой

производительности при обмене данными. К сожалению, использование разделяемой области обычно приводит к необходимости решать проблемы синхронизации доступа и уведомления о событиях.

Память в Symbian OS выделяется непрерывными **участками** (chunks) в адресном пространстве. В архитектуре ARMv5 каждый процесс может иметь до 16-ти таких участков (в ARMv6 это ограничение снято). Главный поток процесса при работе требует от 3 до 4 участков памяти (для стека, кучи, кода, а иногда для WSD-области подключаемых DLL). В случае, если создаваемые в приложении потоки выделяют собственную кучу, на них тратится еще по одному участку памяти⁵. В большинстве случаев разработчик не сталкивается с проблемой исчерпания возможности выделять и подключать к процессу новые участки памяти.

Для работы с участками памяти используется класс RChunk. С его помощью могут создаваться глобальные именованные и локальные неименованные (доступные только для процессов одного потока по хендлу) области памяти. Найти глобальную область памяти можно с помощью класса TFindChunk. При создании задается текущий и максимальный размер выделяемой памяти (выравнивается до размера страницы). Получить ее адрес можно с помощью метода Base(). Память освобождается, когда закрывается последний хендл на нее. Деструкторы размещенных в ней объектов при этом не вызываются.

Считывать и записывать данные в подобные области памяти можно с помощью классов RMemReadStream и RMemWriteStream, которые будут более подробно рассматриваться ниже, в разделе “Потоки данных”.

Так как предоставляемых классом RChunk методов очень много, а используется этот класс довольно редко, мы не будем подробно останавливаться на нем и ограничимся лишь небольшим примером.

```
RChunk chunk;
// Размеры участка памяти
const TInt KChSize = 0x4000;
const TInt KChMaxSize = KChSize*3;
_LIT(KChName, "MySampleChunk");
// Открытие на запись либо создание именованного участка
if (chunk.OpenGlobal(KChName, EFalse) != KErrNone)
    User::LeaveIfError(chunk.CreateGlobal(KChName,
                                         KChSize, KChMaxSize));
CleanupClosePushL(chunk);

// Инициализация потока для записи данных
RMemWriteStream ws(chunk.Base(), chunk.Size());
CleanupClosePushL(ws);
<...> // Запись данных
// Освобождение ресурсов
CleanupStack::PopAndDestroy(2);
```

⁵ SDN FAQ-1392

Очереди сообщений

Для обмена данными между двумя потоками более удобны очереди сообщений. Для их использования необходимо объявить класс, инкапсулирующий передаваемые данные. Максимальный размер передаваемых данных — 256 байт. Пример.

```
struct TMyMessage
{
    TInt iMyIntData;
    TBufC<10> iMyDesData;
};
```

Затем с помощью класса-шаблона RMsgQueue, определенного в заголовочном файле e32msgqueue.h, создается именованная (доступная для всех потоков) или локальная неименованная (доступная только для потоков текущего процесса) очередь на некоторое количество слотов.

```
_LIT(KMsgQueue, "MyMsgQueue");
// Очередь сообщений TMyMessage
RMsgQueue<TMyMessage> queue;
// Открываем именованную очередь
if (queue.OpenGlobal(KMsgQueue) != KErrNone)
    // Создаем очередь на 10 слотов сообщений
    queue.CreateGlobal(KMsgQueue, 5);
```

После этого разработчику станут доступны следующие методы для организации работы с очередью.

- MessageSize() — возвращает размер сообщений, которые будут передаваться в очереди.
- Send() — отправка сообщения в очередь. Результатом работы будет значение KErrNone, если отправка прошла успешно, либо KErrOverflow, если очередь переполнена.
- SendBlocking() — отправка сообщения в очередь. Метод завершает работу только после того, как сообщение будет отправлено. Если очередь заполнена, то вызвавший метод поток будет ждать освобождения слота.
- Receive() — чтение сообщения из очереди и освобождение слота. Результатом работы будет значение KErrNone, если сообщение действительно было получено, либо KErrUnderflow, если очередь пуста.
- ReceiveBlocking() — чтение сообщения из очереди и освобождение слота. Метод завершает работу только после того, как сообщение будет прочитано. Если очередь пуста, то вызвавший метод поток будет ждать появления в ней сообщения.
- NotifyDataAvailable() — асинхронный метод, позволяющий получить уведомление о появлении сообщений в очереди. Запрос может быть отменен с помощью метода CancelDataAvailable().

- `NotifySpaceAvailable()` — асинхронный метод, позволяющий получить уведомление о появлении свободного слота для отправки нового сообщения. Запрос может быть отменен с помощью метода `CancelSpaceAvailable()`.

Как видите, методы класса `RMsgQueue` довольно просты. Мы не будем подробно рассматривать пример работы с ним. Заметьте, что в классе `RMsgQueue` нет метода для чтения сообщения из очереди без его удаления из нее. Поэтому реализовать передачу данных сразу для нескольких клиентских потоков с его помощью затруднительно.

Очередь очищается, а занимаемая ею память освобождается в тот момент, когда закрывается последний хендл на нее.

Механизм уведомлений Publish & Subscribe

Для передачи данных большому числу клиентов используется механизм Publish & Subscribe (P&S, публикации и подписки). Он основан на использовании такого объекта, как **свойство** (property). Этот объект создается на стороне ядра, идентифицируется по паре значений *<категория/ключ>* и может хранить данные определенного типа. Поток (при наличии соответствующих прав) может создавать и удалять свойства, помещать в них данные, подписываться на уведомления об изменении этих данных и считывать их. Обычно создает и изменяет значение свойства один поток, называемый **издателем** (publisher). Все остальные потоки, следящие за изменениями этого свойства и считывающие его данные, называют **подписчиками** (subscriber). Свойства существуют до тех пор, пока не будут удалены либо до перезагрузки ОС.

Для работы со свойствами используется класс `RProperty`, определенный в заголовочном файле `e32property.h`. Новые свойства создаются с помощью метода `Define()`, при этом указывается категория, ключ, тип и (необязательно) предварительно выделяемый объем памяти.

Значением категории чаще всего выбирается `SID` процесса. В этом случае для создания свойства не потребуется наличия доступа к каким-либо защищенным возможностям. Если же категория отлична от `SID`, то ее значение должно быть меньше величины `KUIdSecurityThresholdCategoryValue (0x10273357)`, а процесс должен иметь доступ к защищенной возможности `WriteDeviceData`. Чаще всего в этом случае выбирают категорию `KUIdSystemCategory (0x101F75B6)`. Если условия создания свойства с заданной категорией не выполняются, то метод `Define()` вернет значение `KErrPermissionDenied`.

Ключ является целым беззнаковым числом, он необходим для создания нескольких свойств в одной категории.

В качестве типа нового свойства могут использоваться следующие значения из объявленного в классе `RProperty` перечисления `TType`.

- `EInt` — свойство хранит целое число. По умолчанию оно инициализируется нулем.

- `EByteArray` или `EText` — свойство хранит массив байт. Максимальный размер данных не должен превышать константы `RProperty::KMaxPropertySize (512)`.
- `ELargeByteArray` или `ELargeText` — свойство хранит массив байт. Максимальный размер данных не должен превышать константы `RProperty::KMaxLargePropertySize (65535)`.

В случае, если тип размещаемых в свойстве данных отличается от `EInt`, разработчик может в последнем аргументе метода `Define ()` указать объем памяти, который должен предварительно выделяться для их хранения.

Ниже приводятся примеры определения трех свойств различного типа.

```
const TUid KMyAppSID = {0xE6205EE2};
const TUint KMyPropKey = 0;

RProperty pr;
pr.Define(KMyAppSID, KMyPropKey, RProperty::EInt);
pr.Define(KMyAppSID, KMyPropKey+1, RProperty::EText);
pr.Define(KMyAppSID, KMyPropKey+2,
          RProperty::ELargeText, 1000);
```

Существуют перегруженные варианты метода `Define ()`, позволяющие с помощью объектов типа `TSecurityPolicy` задать политику безопасности для записи и чтения данных из свойства. Проверка может осуществляться по `SID`, `VID` и наличию доступа к определенным защищенным возможностям процесса, поток которого обращается к свойству.

Подключиться к свойству (открыть хендл) можно с помощью метода `Attach ()`, принимающего в качестве аргументов категорию, ключ, а также (необязательный параметр) объект типа `TOwnerType`, регламентирующий область действия хендла и уже обсуждавшийся выше в разделе “Работа с процессами и потоками”.

```
User::LeaveIfError(pr.Attach(KMyAppSID, KMyPropKey+2));
```

Интересной особенностью является то, что подключиться можно и к еще не определенному свойству — ошибки не произойдет. В ряде случаев это удобно. Например, в клиент-серверной архитектуре клиентское приложение может подключаться к свойству при запуске, а сервер, осуществляющий определение и запись данных в это свойство, может стартовать намного позднее (по первому запросу клиента).

Подключившись к свойству, вы можете использовать ряд перегруженных `Get ()` и `Set ()` методов. Они позволяют считывать или записывать данные в свойство. В качестве данных может быть целое число либо дескриптор (8- или 16-битовый). Имеются также ряд перегруженных методов, позволяющих дополнительно указать категорию и ключ — их можно использовать без предварительного подключения к свойству, но выполняются они несколько медленнее.

Помимо операций чтения и записи, разработчик может подписаться на уведомление об изменении данных в свойстве. Для этого служит асинхронный ме-

тод `Subscribe()`. Запрос завершается даже в том случае, если записываемые в свойство данные совпадают с уже хранящимися в нем. Если изменение данных осуществляется достаточно часто, то на момент вызова обработчика `RunL()` асинхронного объекта значение свойства может поменяться несколько раз. Для того чтобы быть уверенным, что вы не пропустите ни одно из записанных в него значений, необходимо повторно вызывать в обработчике метод `Subscribe()` до чтения данных из свойства, а не после. В противном случае вы получите не следующее помещенное в свойство значение, а первое после последнего чтения. Это особенность реализации свойств на стороне ядра. Отмена асинхронного запроса `Subscribe()` осуществляется с помощью метода `Cancel()`.

Для удаления свойства используется метод `Delete()`, при этом асинхронные запросы всех подписчиков завершаются с кодом `KErrNotFound`. Для того чтобы удалить свойство, процесс потока должен иметь необходимые для доступа к нему права.



Еще один метод IPC, появившийся лишь начиная с Symbian 9.3, — конвейеры. Работа с ними осуществляется с помощью класса `RPipe`. На момент написания книги этот класс не входит в состав SDK и документация по нему отсутствует.

Подготовка к сертификации ASD

- Знание механизмов IPC, использование которых наиболее предпочтительно в Symbian OS (клиент-серверное взаимодействие, механизм Publish & Subscribe, очереди сообщений). Умение выбрать наиболее подходящий механизм для предложенной задачи.
 - Умение использовать механизм Publish & Subscribe для получения и подписки на уведомления об изменениях системных свойств. Понимание роли платформы безопасности в организации защиты свойств от вредоносных манипуляций.
-

Клиент-серверная архитектура приложений

Множество сервисов Symbian OS реализуется с помощью серверов. Например, приложение, считывающее данные из файла, на самом деле обращается к специальному файловому серверу, который осуществляет чтение и передает прочитанные данные программе. Таким образом, клиент-серверная архитектура также является методом IPC, но ее возможности куда шире, чем простой обмен данными и сигналами. Серверы могут управлять доступом к разделяемым ресурсам, реализовывать сложную логику и политику безопасности. Кроме того, клиент-серверная архитектура часто используется при создании приложений, функционирующих в фоновом режиме. Например, при разработке спам-фильтра SMS-сообщений вы можете реализовать всю логику по работе с коммуникационными сервисами в виде сервера без GUI, который будет постоянно

выполняться на устройстве. В то время как для его настройки и управления можно создать полноценное GUI-приложение-клиент с богатым интерфейсом и анимацией. Клиент будет запускаться только тогда, когда к нему обратиться пользователь. По завершению работы клиент отправляет необходимые управляющие команды серверу и выгружается из памяти. В результате экономятся ресурсы и минимизируется энергопотребление устройства.

Общие сведения

По длительности работы различают два вида серверов: **постоянные** (persistent) и **временные** (transparent). Постоянные серверы запускаются при старте системы и завершают свое выполнение вместе с ней. Временные серверы запускаются при обращении к ним первого клиента и выгружаются после отключения последнего клиента (обычно с некоторой задержкой). Иногда встречаются решения, сочетающие оба этих принципа поведения.

Сервер реализуется специальным активным объектом и всегда запускается в отдельном от клиента потоке (а в большинстве случаев — в другом процессе). Для связи клиента и сервера используется **сессия** (session) — объект на стороне ядра системы, создание которого инициируется клиентом (рис. 6.5). Работа с сессией ведется через R-класс на стороне клиента и C-класс на стороне сервера. Установив сессию, клиент может создать запрос, в результате чего на сервер отправляется сообщение, содержащее код команды и несколько простых параметров (числа или указатели). На стороне сервера имеется метод, проверяющий код команды в поступившем сообщении и на основании этого обрабатывающий переданные в нем параметры. Когда сервер сигнализирует через сообщение об окончании его обработки, запрос клиента завершается. Клиент может посылать как синхронные, так и асинхронные запросы (сервер в этом случае становится поставщиком асинхронного сервиса).

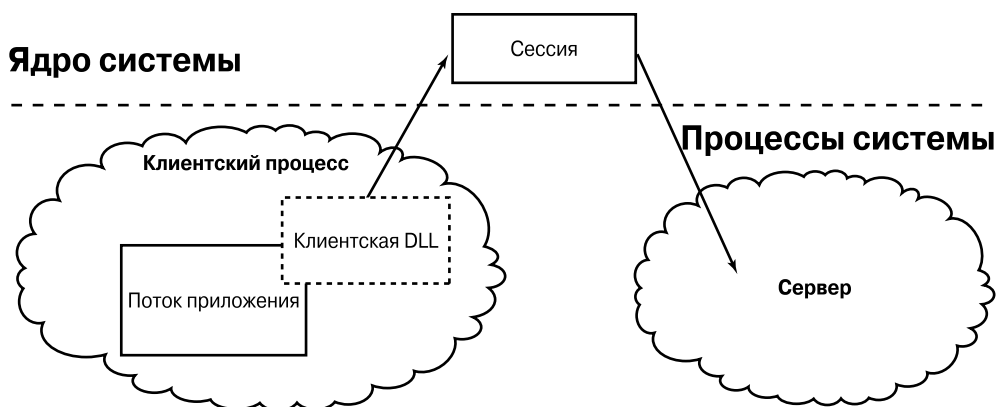


Рис. 6.5. Клиент-серверная архитектура

Чаще всего вместе с сервером создается и клиентский R-класс для доступа к нему. Он инкапсулирует класс для работы с сессией и содержит методы для отправки запросов серверу. Таким образом, разработчику не требуется помнить коды команд сервера и их параметры. Кроме того, данный R-класс принято размещать в DLL, считающейся частью сервера. Это довольно удобно. Например, для доступа к файловому серверу системы (`efile.exe`) используется класс `RFs`, объявленный в заголовочном файле `f32file.h`. Этот класс реализован в `efile.dll`, поэтому к MMP-файлу программы необходимо подключить библиотеку импорта `efile.lib`. Сама же библиотека `efile.dll`, как и сервер, являются частью системы, они предустановлены на устройстве. Таким образом, можно не включать клиентскую библиотеку в дистрибутив программы. Разработчику также не нужно беспокоиться о соответствии версии библиотеки DLL и сервера.

Клиент может установить несколько сессий с сервером, хотя это и не рекомендуется в целях экономии ресурсов. Обычно в этом случае используются **субсессии** (`subsession`) — объекты на стороне ядра, использующие сессию и занимающие несколько меньше ресурсов. Например, установив одну сессию с файловым сервером, вы можете, создав на ее основе субсессии в классах `RFile`, работать с несколькими файлами одновременно. При закрытии сессии все субсессии закрываются автоматически.

Итак, разобравшись с общими принципами работы клиент-серверной архитектуры, мы можем перейти к детальному рассмотрению ее компонентов.

Сервер

Для того чтобы создать новый сервер, необходимо реализовать C-класс, порожденный от класса активного объекта `CServer2`. Класс `CServer2` содержит чистый виртуальный метод `NewSessionL()`, поэтому не может быть использован напрямую.

В конструкторе класса `CServer2` указываются приоритет нового активного объекта и требуемое значение из перечисления типа `TServerType` — оно позволяет задать тип сессий, которые будут создаваться данным сервером. Различаются они лишь по области доступа:

- `EUnsharableSessions` — (принимается по умолчанию) доступ возможен только из текущего потока;
- `ESharableSessions` — сессия может быть доступна только из потоков текущего процесса;
- `EGlobalSharableSessions` — сессия может быть доступна из любого потока.

Передачей и клонированием сессий занимаются клиентские приложения. Для этого в классе, представляющем сессию на стороне клиента, имеются соответствующие методы. В случае если сервер не разрешил использовать затребованную область доступа, работа этих методов завершается с ошибкой.



Названия большинства используемых при реализации клиент-серверной архитектуры классов имеют цифру 2 на конце. Это означает: “версия 2”. Классы первой версии применялись до введения платформы безопасности в Symbian 9.x. В настоящий момент они не используются.

После создания порожденного от `CServer2` активного объекта, разработчику доступны следующие методы.

- `Start()` и `StartL()` — запуск сервера с заданным именем. Сервер периодически вызывает асинхронные запросы и ожидает установления новой сессии с ним. Остановить работу сервера можно, воспользовавшись методом `Cancel()` или `CActiveScheduler::Stop()`.
- `ReStart()` — перезапуск сервера. Эквивалентен последовательному вызову методов `Cancel()` и `Start()`. Используется редко.
- `Server()` — возвращает объект класса `RServer2`, позволяющий получить доступ к серверному представлению на стороне ядра. Используется довольно редко, в основном для получения хендла этого объекта. Такой хендл может указываться клиентом вместо имени сервера для установления сессии.
- `NewSessionL()` — виртуальный метод, который сервер должен реализовать. Этот метод вызывается системой в тот момент, когда клиент устанавливает с сервером новую сессию. Сервер должен создать и вернуть в качестве результата экземпляр порожденного от `CSession2` класса. Этот объект будет являться представлением сессии на стороне сервера. Он уничтожается в тот момент, когда сессия будет разорвана. Именно ему будут поступать передаваемые серверу сообщения. В качестве аргументов метода `NewSessionL()` на сервер поступает информация о версии клиента и дополнительная информация об устанавливаемой сессии.



Так как передавать хендл запущенного сервера клиентам для установления сессии затруднительно, то в подавляющем большинстве случаев серверы ищутся по имени с помощью класса `TFindServer`. Это создает опасность подмены имени вашего сервера вредоносным приложением. Существует возможность в некоторой степени снизить подобный риск. Для этого необходимо назвать сервер именем, начинающимся с “!”. Запускать такие серверы могут только процессы, имеющие доступ к защищенной возможности `ProtServ`.

Приведем пример простейшего сервера.

```
class CSimpleServer: public CServer2
{
public:
    CSimpleServer();
private:
    virtual CSession2* NewSessionL(const TVersion& aVersion,
    const RMessage2& aMessage) const;
};
```

```
// Простейший конструктор
CSimpleServer::CSimpleServer() : CServer2(EPriorityStandard)
{
}

// Метод для создания серверного представления новой сессии
CSession2* CSimpleServer::NewSessionL(const TVersion&
/*aVersion*/,const RMessage2& /*aMessage*/) const
{
// Мы не создаем сессии,
// поскольку еще не объявили нужный класс
return NULL;
}
```

Создать и запустить такой сервер можно следующим образом (предполагается, что инфраструктурные объекты в программе уже созданы).

```
// Создание сервера
CSimpleServer* server = new (ELeave) CSimpleServer();
CleanupStack::PushL(server);
// Имя сервера
_LIT(KServerName, "MySimpleServer");
// Запуск сервера с заданным именем
server->StartL(KServerName);
// Запуск цикла ожидания планировщика
CActiveScheduler::Start();
// Удаление сервера
CleanupStack::PopAndDestroy(server);
```

Вышеприведенный код можно поместить в метод `MainL()` приложения `HelloWorld`.

Представление сессии на стороне сервера

Сессия на стороне сервера должна быть представлена классом, порожденным от класса `CSession2`. Класс `CSession2` имеет чистый виртуальный метод `ServiceL()` и не может быть использован напрямую.

Объекты порожденного от `CSession2` класса должны создаваться только в методе `NewSessionL()` сервера и возвращаться в качестве его результата. Только так они будут получены системой и использованы для передачи посланного из клиента сообщения.

Класс `CSession2` предоставляет следующие методы.

- `Server()` — возвращает указатель на активный объект сервера, с которым установлена сессия.
- `ServiceL()` — метод-обработчик посланных клиентом сообщений. Должен быть реализован в порожденном от `CSession2` классе. Выполняется как часть метода `RunL()` активного объекта сервера.

- `ServiceError()` — вызывается в случае, если в методе `ServiceL()` произошел не пойманный в ловушку сброс. Вызывается из метода `RunError()` активного объекта сервера. В качестве аргументов получает сообщение, обработка которого привела к сбросу, и код ошибки. Если разработчик не переопределяет этот метод в порожденном от `CSession2` классе, то сообщение завершится с кодом ошибки.
- `CreateL()` — метод, вызываемой системой после создания сессии. Разработчик может переопределить его, чтобы выполнить инициализацию необходимых объектов. По умолчанию этот метод ничего не делает.

Простейшее серверное представление сессии выглядит следующим образом.

```
class CSimpleSession : public CSession2
{
public:
    CSimpleSession();
    virtual void ServiceL(const RMessage2& aMessage);
};

CSimpleSession::CSimpleSession() : CSession2()
{
}

void CSimpleSession::ServiceL(const RMessage2& aMessage)
{
    aMessage.Complete(KErrNone);
}
```

Данная сессия завершает любой посланный клиентом запрос кодом `KErrNone`. Работу с сообщениями с помощью класса `RMessage2` мы обсудим несколько позднее. Для создания объектов серверного представления сессии необходимо внести изменения в метод `NewSessionL()` вышеприведенного сервера.

```
CSession2* CSimpleServer::NewSessionL(const TVersion&
    /*aVersion*/, const RMessage2& /*aMessage*/) const
{
    return new (ELeave) CSimpleSession;
}
```

Представление сессии на стороне клиента

Как уже отмечалось ранее, именно клиент инициирует установление сессии с сервером. Для этого необходимо объявить R-класс, порожденный от класса `RSessionBase`. Все необходимые для работы методы в `RSessionBase` помещены в секцию `protected`, поэтому напрямую воспользоваться ими нельзя. Такая политика вынуждает разработчика создавать полноценные клиентские R-классы, скрывающие использование кодов команд и детали передачи данных.

Порожденный от `RSessionBase` класс получает доступ к следующим методам.

- Множество перегруженных вариантов (в том числе асинхронных) метода `CreateSession()` для создания новой сессии.
- Множество перегруженных вариантов метода `Open()` для открытия существующей сессии.
- `Send()` — отправка сообщения без ожидания результата его обработки.
- `SendReceive()` — отправка сообщения и получение результата его обработки. Имеются синхронный и асинхронный варианты этого метода.
- `ShareAuto()` — должен быть вызван для подготовки хендла сессии, который может быть передан другим потокам клиентского процесса. Если при создании сервера в конструктор класса `CServer2` было передано значение, запрещающее клиентам передавать сессию между потоками одного процесса, то метод вернет код ошибки.
- `ShareProtected()` — должен быть вызван для подготовки хендла сессии, который может быть передан другим потокам (возможно, принадлежащих разным процессам). Если при создании сервера в конструктор класса `CServer2` было передано значение, запрещающее клиентам передавать сессию между потоками, то метод вернет код ошибки.

При создании сессии клиент должен указать имя или хендл сервера, с которым он хочет установить соединение, а также объект типа `TVersion`, содержащий минимальную версию сервера, с которым совместим клиент. Пример.

```
_LIT(KServerName, "MySimpleServer");
TInt res = session.CreateSession(KServerName,
                                TVersion(1, 0, 0));
```

Перегруженные варианты позволяют также зарезервировать слоты сообщений для новой сессии (слоты сообщений будут обсуждаться позднее), указать необходимую область доступа к сессии (снимает необходимость использования методов `ShareAuto()` и `ShareProtected()`), а также передать объект `TSecurityPolicy` с правилами политики безопасности, которым должен удовлетворять сервер. Класс `TSecurityPolicy` уже обсуждался выше, в разделе “Механизм уведомлений Publish & Subscribe”.

При вызове синхронного метода `CreateSession()` выполняется поиск зарегистрированного в системе сервера. Запущенный сервер создает в ядре именованный объект, подключиться к которому можно с помощью класса `RServer2`, поэтому его легко найти. Если сервер удовлетворяет предъявляемым клиентом требованиям политики безопасности, начинается установка сессии. При этом на стороне сервера вызывается метод `NewSessionL()`, в который передается информация о совместимости клиента (`TVersion`). Если метод `NewSessionL()` возвращает указатель на порожденный от `CSession2` объект, то система, чтобы завершить его создание, вызывает у него метод `CreateL()`. Затем связанный с представлением сессии на сервере объект создается в ядре, а экземпляр порожденного от `RSessionBase` класса на стороне клиента открывает хендл на него.

При отправке команд и данных с помощью открытой сессии на стороне сервера будет вызываться метод `ServiceL()` представления сессии. Простейший класс для установления сессии и отправки команд выглядит следующим образом.

```
class RSimpleSession: RSessionBase
{
public: TInt Connect();
};

TInt RSimpleSession::Connect()
{
//Класс скрывает имя сервера и информацию о совместимости
_LIT(KServerName, "MySimpleServer");
return CreateSession(KServerName, TVersion(1, 0, 0));
}
```

Использоваться он может следующим образом.

```
RSimpleSession session;
User::LeaveIfError(session.Connect());
CleanupClosePushL(session);
<...> // Работа с сервером
CleanupStack::PopAndDestroy(&session);
```

На данный момент класс `RSimpleSession` не позволяет отправить никаких команд или данных на сервер.

Запуск сервера при установке сессии

В случае если сервер относится к категории временных, то в момент установления первой сессии он еще не будет запущен. Поэтому класс `RSimpleSession` должен уметь запускать его при необходимости. Здесь нам пригодятся навыки работы с процессами и потоками. Следующий пример демонстрирует улучшенный вариант класса `RSimpleSession`.

```
class RSimpleSession: RSessionBase
{
public:
    TInt Connect();
private:
    TInt StartServer();
};

// Запуск сервера
TInt RSimpleSession::StartServer()
{
    RProcess server;
    // Исполняемый файл, реализующий сервер
```

```

_LIT(KServerExecutable, "simpleserver0xE0102040.exe");
// Создаем процесс сервера
TInt res = server.Create(KServerBinaryName, KNullDesC);

if (KErrNone == res)
{
    // Готовимся к ожиданию randevу с сервером
    TRequestStatus req;
    server.Rendezvous(req);
    // Запускаем процесс на выполнение
    server.Resume();
    // Синхронно ожидаем randevу
    User::WaitForRequest(req);
    // Сервер либо погиб, либо randevу наступило

    // Проверяем, выполняется ли сервер
    if (server.ExitType() == EExitPanic)
        res = KErrGeneral;
    else
        res = req.Int();

    server.Close(); // Закрываем хендл на процесс
}

return res;
}

TInt RSimpleSession::Connect()
{
    // Установление сессии с сервером
    _LIT(KServerName, "MySimpleServer");
    TVersion ver(1, 0, 0);
    TInt res = CreateSession(KServerName, ver);
    if (res == KErrNotFound || res == KErrServerTerminated)
    {
        // Если сервер не найден, пробуем запустить
        res = StartServer();
        // Если запустился, еще раз пытаемся установить сессию
        if (res == KErrNone || res == KErrAlreadyExists)
            res = CreateSession(KServerName, ver);
    }

    return res;
}

```

Вышеприведенный код содержит одну маленькую хитрость — использование метода `Rendezvous()`. Клиент не пытается подключиться к серверу сразу после запуска процесса, а ожидает сигнала-randevу. Это связано с тем, что между стартом приложения и стартом активного объекта, реализующего сервер,

проходит некоторое время. Естественно, серверный процесс должен вызывать метод `Rendezvous()` сразу после запуска активного объекта сервера, чтобы уведомить о своей готовности установить сессию.

```
// Создание сервера
CSimpleServer* server = new (ELeave) CSimpleServer();
CleanupStack::PushL(server);

// Запуск сервера с заданным именем
server->StartL(KServerName);

// Рассылка уведомления о готовности сервера
RProcess::Rendezvous(KErrNone);

// Запуск цикла ожидания планировщика
CActiveScheduler::Start();
```

Я советую использовать этот прием в реализации всех клиентских сессий, даже если сервер постоянный и запускается самостоятельно при старте системы. Это поможет восстановить работу сервера в том случае, если он завершил свое выполнение аварийно или был отключен пользователем с помощью сторонних утилит.

Остановка сервера

При реализации временного сервера возникает необходимость программно завершить его работу, после того как от него отключится последняя сессия. К сожалению, класс `CServer2` не предоставляет возможности получить количество установленных с ним сессий, поэтому в порожденный им класс необходимо добавить новый член для его хранения, а также методы для изменения этой величины.

```
class CSimpleServer : public CServer2
{
public:
    inline void AddSession();
    inline void RemoveSession();
    <...>
private:
    TInt iSessionCount;
};

inline void CSimpleServer::AddSession()
{
    ++iSessionCount;
}

inline void CSimpleServer::RemoveSession()
```

```

{
if (--iSessionCount==0)
    CActiveScheduler::Stop(); // Остановка сервера
}

```

Вызывать метод `AddSession()` следует при создании новой сессии — в методе `NewSessionL()` или в методе `CreateL()` класса `CSimpleSession`, а `RemoveSession()` при ее отключении — в деструкторе класса `CSimpleSession`. Получить указатель на сервер из его сессии можно следующим образом.

```

CSimpleServer* server = static_cast<CSimpleServer*>
(const_cast<CServer2*>(Server()));

```

Сложнее обеспечить задержку перед отключением сервера. Такая задержка позволит случайно прерванной сессии восстановиться раньше, чем это приведет к выгрузке сервера. Для ее реализации необходимо воспользоваться одним из асинхронных таймеров, рассматриваемых ниже в этой главе, в разделе “Таймеры”. Хороший пример такого таймера описан в книге Джо Стичбури (Jo Stichbury) “Symbian OS Explained”. С ее разрешения я привожу здесь этот класс.

```

const TInt KShutdownDelay=2000000; // 2 секунды
class CShutdown : public CTimer
{
public:
    inline CShutdown();
    inline void ConstructL();
    inline void Start();
private:
    void RunL();
};

inline CShutdown::CShutdown()
    :CTimer(-1)
    {CActiveScheduler::Add(this);}
inline void CShutdown::ConstructL()
    {CTimer::ConstructL();}
inline void CShutdown::Start()
    {After(KShutdownDelay);}

void CShutdown::RunL()
{
    CActiveScheduler::Stop();
}

```

Класс `CSimpleServer` в этом случае должен иметь член `CShutdown`, а методы `AddSession()` и `RemoveSession()` выглядеть следующим образом.

```

inline void CSimpleServer::AddSession()
{
    ++iSessionCount;
}

```

```

        iShutdown.Cancel();
    }

inline void CSimpleServer::RemoveSession()
{
    if (--iSessionCount==0)
        iShutdown.Start();
}

```



Описанное в книге Джо демонстрационное клиент-серверное приложение Hercules можно скачать с ее интернет-сайта. Ссылка приводится в разделе “Документация” в конце книги. Приложение Hercules интересно еще и тем, что имеется его версия для Symbian 7.x–8.x с использованием классов CServer, CSession и RMessage.

Команды, сообщения и передача данных

Для передачи данных между клиентом и сервером используются сообщения — специальные объекты, создаваемые на стороне ядра. В сообщении содержится код команды и от 0 до 4 аргументов. Аргументами могут быть целые числа и указатели. После завершения обработки сообщения, оно удаляется.

Для хранения передаваемых серверу сообщений в системе создается **пул (pool) слотов сообщений** (message slots). В пуле выделяются 255 слотов, поэтому сервер может обрабатывать до 255 сообщений одновременно. В случае если клиент отправляет только синхронные запросы, то в рамках сессии в каждый момент времени может передаваться только одно сообщение. Если же несколько команд клиент отправляет асинхронно, то в рамках сессии могут одновременно существовать несколько сообщений.

Некоторые перегруженные методы CreateSession() и Open() класса RSubSessionBase позволяют зарезервировать определенное количество слотов для использования сессией. Разработчик сам должен решить, сколько слотов ему необходимо. Если свободных слотов сервера окажется недостаточно, то сессия не будет установлена, а методы завершатся с ошибкой KErrServerBusy. Если при создании сессии слоты не резервируются, то такая проверка не проводится, и ошибку KErrServerBusy можно получить уже во время отправки сообщения.

Для отправки сообщения на сервер используется ряд перегруженных методов Send() и SendReceive() класса RSubSessionBase. Их обязательным аргументом является код команды сервера (целое число), необязательным — аргументы команды. Метод SendReceive() также может выполняться асинхронно, для чего ему передается ссылка на объект TRequestStatus.

Для указания параметров команды используется объект типа TIpArgs, способный хранить от 0 до 4 аргументов. Эти аргументы передаются через конструктор в момент создания объекта, либо позднее, с помощью метода Set().

Аргументами могут быть целые числа, указатели, хендлы и дескрипторы. На стороне сервера доступ к ним осуществляется по порядковому номеру.

В серверном потоке доступ к сообщению осуществляется с помощью класса `RMessage2` (рис. 6.6). После отправки команды клиентом, в объекте серверного представления сессии вызывается метод `ServiceL()`, в качестве аргумента которого по ссылке передается объект `RMessage2`, уже подключенный к находящемуся на стороне ядра сообщению.

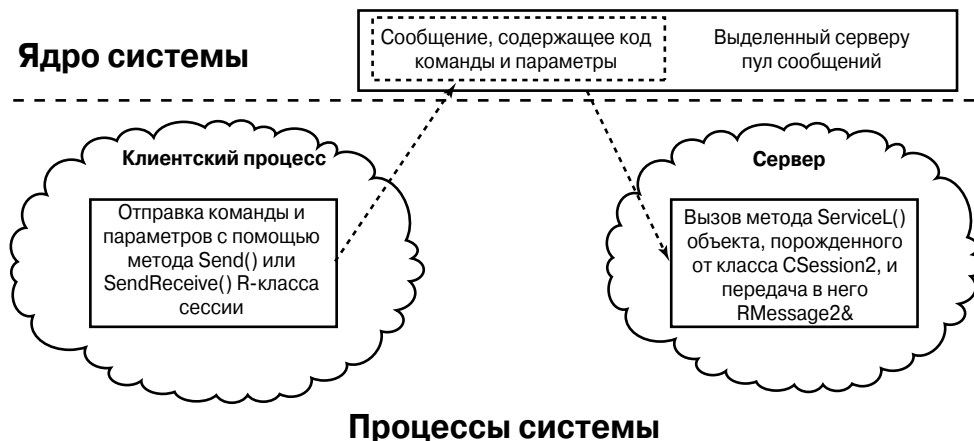


Рис. 6.6. Передача сообщения серверу

Для класса `RMessage2` доступны следующие методы.

- `Function()` — возвращает код команды, запрашиваемой клиентом.
- `Int0()`, `Int1()`, `Int2()`, `Int3()` — возвращают значение переданных серверу параметров в виде `TInt`.
- `Ptr0()`, `Ptr1()`, `Ptr2()`, `Ptr3()` — возвращают значение переданных серверу параметров в виде обобщенных указателей `TAny`.
- `GetDesLength()` и `GetDesLengthL()` — позволяют получить длину строки, хранящейся в дескрипторе, адрес которого находится в одном из переданных серверу параметров. Сам дескриптор находится в клиентском потоке.
- `GetDesMaxLength()` и `GetDesMaxLengthL()` — позволяют получить размер дескриптора, адрес которого находится в одном из переданных серверу параметров. Сам дескриптор находится в клиентском потоке.
- `Read()` и `ReadL()` — чтение данных из указанного параметра в заданный 8- или 16-битовый дескриптор.
- `Write()` и `WriteL()` — запись данных из 8- или 16-битового дескриптора в указанный параметр сообщения. Предполагается, что в параметре содержится адрес дескриптора на стороне клиента, куда будут помещены данные.
- `Complete()` — сигнализирует о завершении выполнения запрошенной клиентом функции. В качестве аргумента методу `Complete()` передается результат выполнения функции (`KErrNone` — в случае, если завершен).

успешно). Если клиент посылал синхронный запрос, то методы `Send()` и `SendReceive()` вернут управление, и его работа продолжится. Если асинхронный, то будет вызван обработчик `RunL()` того активного объекта, чей `iStatus` был передан в сообщении. После вызова метода `Complete()` сообщение удаляется, а занимаемый им слот в пуле сообщений освобождается. Продолжать работу с объектом `RMessage2` нельзя (метод `IsNull()` вернет значение `ETrue`).

- `IsNull()` — проверяет, является ли хендл объекта `RMessage2` нулевым. Если это так, то он уже не подключен к сообщению в пуле сообщений сервера.

В большинстве случаев вышеприведенных методов вполне достаточно для получения всех данных сообщения, их обработки и завершения клиентского запроса. Следующие доступные объекту `RMessage2` методы используются гораздо реже.

- `Session()` — возвращает указатель на сессию (на объект класса `CSession2`), которой принадлежит сообщение.
- `ClientStatus()` — получение указателя на объект `TRequestStatus` клиента. Если запрос был послан синхронно, возвращает `NULL`.
- `Client()` и `ClientL()` — позволяют подключить объект `RThread` к клиентскому потоку, отправившему сообщение.
- `SecureId()` и `VendorId()` — возвращают идентификаторы `SID` и `VID` процесса, которому принадлежит клиентский поток.
- Несколько перегруженных методов `HasCapability()` позволяют проверить наличие у содержащего клиентский поток процесса доступа к заданной защищенной возможности.
- `Panic()`, `Kill()` и `Terminate()` — завершение работы клиентского потока, отправившего сообщение (см. раздел “Работа с процессами и потоками”, выше в этой главе).

Следующие примеры демонстрируют передачу и получение различных типов данных на стороне сервера и клиента.

```
/*
 * Коды команд.
 * Должны совпадать на стороне клиента и сервера.
 */

enum TSimpleServerFunctions
{
    ERunFunction,    // Простой вызов функции сервера
    ESendInt,        // Передача целого числа
    ESendDes,        // Передача дескриптора
    ESendStruct,     // Передача данных простого типа
    ERequestInt,     // Получение числа
    ERequestDes,     // Получение дескриптора
    ERequestStruct   // Получение данных простого типа
}
```

```

};

/*
 * Класс сессии на стороне клиента
 * содержащий методы для вызова функций сервера
 */

_LIT(KServerName, "MySimpleServer");

class RSimpleSession : RSessionBase
{
public:

    // Простой T-класс
    struct TSimpleData
    {
        TInt iDataInt;
        TBuf<100> iDataBuf;
    };

    // Подключение к серверу
    TInt Connect();
    // Простой вызов функции сервера
    TInt NotifyServer();
    // Передача целого числа
    TInt SendInt(const TInt aInt);
    // Передача дескриптора
    TInt SendDes(const TDesC& aDes);
    // Передача данных простого типа
    TInt SendStruct(const TSimpleData aData);
    // Получение числа
    TInt RequestInt(TInt& aInt);
    // Получение дескриптора
    TInt RequestDes(TDes& aDes);
    // Получение данных простого типа
    TInt RequestStruct(TSimpleData& aData);
};

// Подключение к серверу
TInt RSimpleSession::Connect()
{
    return CreateSession(KServerName, TVersion(1, 0, 0));
}

// Простой вызов функции сервера
TInt RSimpleSession::NotifyServer()
{

```

```
        return SendReceive(ENotify);
    }

    // Передача целого числа
    TInt RSimpleSession::SendInt(const TInt aInt)
    {
        TIPCArgs args(aInt);
        return SendReceive(ESendInt, args);
    }

    // Передача дескриптора
    // Обратите внимание, что при передаче аргумента дескриптора
    // с помощью базового типа он всегда должен
    // передаваться по ссылке

    TInt RSimpleSession::SendDes(const TDesC& aDes)
    {
        TIPCArgs args(&aDes);
        return SendReceive(ESendDes, args);
    }

    // Передача данных простого типа
    TInt RSimpleSession::SendStruct(const TSimpleData aData)
    {
        TPckg<TSimpleData> data(aData);
        TIPCArgs args(&data);
        return SendReceive(ESendStruct, args);
    }

    // Получение числа
    TInt RSimpleSession::RequestInt(TInt& aInt)
    {
        TPckg<TInt> data(aInt);
        TIPCArgs args(&data);
        return SendReceive(ERequestInt, args);
    }

    // Получение дескриптора
    TInt RSimpleSession::RequestDes(TDes& aDes)
    {
        TIPCArgs args(&aDes);
        return SendReceive(ERequestDes, args);
    }

    // Получение данных простого типа
    TInt RSimpleSession::RequestStruct(TSimpleData& aData)
    {

```

```
TPckg<TSimpleData> data(aData);
TIpArgs args(&data);
return SendReceive(ERequestStruct, args);
}
```



Если вы решите вместо метода `SendReceive()` использовать синхронный, но не дожидаящийся завершения обработки сообщения сервером метод `Send()`, учтите, что чтение передаваемых как дескриптор данных осуществляется из памяти клиентского потока, а значит, они должны в этот момент существовать. Например, нельзя просто заменить `SendReceive()` на `Send()` в методе `RequestStruct()`, так как дескриптор-указатель будет удален из памяти раньше, чем к нему обратится сервер.

```
/*
 * Класс сессии на стороне сервера
 */

class CSimpleSession : public CSession2
{
public:
    CSimpleSession();
    ~CSimpleSession();

    virtual void ServiceL(const RMessage2& aMessage);
    virtual void ServiceError(const RMessage2& aMessage,
                              TInt aError);

private:
    // Простой вызов функции сервера
    void NotifyServer();
    // Получение целого числа
    void GetInt(const RMessage2& aMessage);
    // Получение дескриптора
    void GetDesL(const RMessage2& aMessage);
    // Получение данных простого типа
    void GetStructL(const RMessage2& aMessage);
    // Отправка числа
    void AnswerIntL(const RMessage2& aMessage);
    // Отправка дескриптора
    void AnswerDesL(const RMessage2& aMessage);
    // Отправка данных простого типа
    void AnswerStructL(const RMessage2& aMessage);

private:
    TInt iInt;
    HBufC* iDes;
    RSimpleSession::TSimpleData iData;
};
```

```
// Обработчик сообщений клиента
void CSimpleSession::ServiceL(const RMessage2& aMessage)
{
    switch (aMessage.Function())
    {

        // Простой вызов функции сервера
        case ENotify:
        {
            NotifyServer();
            aMessage.Complete(KErrNone);
        }
        break;

        // Получение целого числа
        case ESendInt:
        {
            GetInt(aMessage);
        }
        break;

        // Получение дескриптора
        case ESendDes:
        {
            GetDesL(aMessage);
        }
        break;

        // Получение данных простого типа
        case ESendStruct:
        {
            GetStructL(aMessage);
        }
        break;

        // Отправка числа
        case ERequestInt:
        {
            AnswerIntL(aMessage);
        }
        break;

        // Отправка дескриптора
        case ERequestDes:
        {
            AnswerDesL(aMessage);
        }
    }
}
```

```

        break;

// Отправка данных простого типа
case ERequestStruct:
{
    AnswerStructL(aMessage);
}
break;

default:
    // Клиент попытался вызвать несуществующую команду
    aMessage.Panic(_L("Wrong command"), KErrGeneral);
}
}

// Простой вызов функции сервера
void CSimpleSession::NotifyServer()
{
}

// Получение целого числа
void CSimpleSession::GetInt(const RMessage2& aMessage)
{
    TInt = aMessage.Int0();
}

// Получение дескриптора
void CSimpleSession::GetDesL(const RMessage2& aMessage)
{
    TInt len = aMessage.GetDesLengthL(0);
    // Проверим, можем ли мы прочитать все данные
    if (!iDes || iDes->Length() < len)
    {
        delete iDes;
        iDes = HBufC::NewL(len);
    }

    // Чтение
    TPtr des(iDes->Des());
    aMessage.ReadL(0, des);
    aMessage.Complete(KErrNone);
}

// Получение данных простого типа
void CSimpleSession::GetStructL(const RMessage2& aMessage)
{
    TPckg<RSimpleSession::TSimpleData> data(iData);
    aMessage.ReadL(0, data);
}

```

```

    aMessage.Complete(KErrNone);
}

// Отправка числа
void CSimpleSession::AnswerIntL(const RMessage2& aMessage)
{
    TPckg<TInt> pkgint(iInt);
    aMessage.WriteL(0, pkgint);
    aMessage.Complete(KErrNone);
}

// Отправка дескриптора
void CSimpleSession::AnswerDesL(const RMessage2& aMessage)
{
    if (iDes)
    {
        aMessage.WriteL(0, *iDes);
        aMessage.Complete(KErrNone);
    }
    else
        aMessage.Complete(KErrNotReady);
}

// Отправка данных простого типа
void CSimpleSession::AnswerStructL(const RMessage2& aMessage)
{
    TPckg<RSimpleSession::TSimpleData> data(iData);
    aMessage.WriteL(0, data);
    aMessage.Complete(KErrNone);
}



// Обработка возникших в ServiceL() сбросов
void CSimpleSession::ServiceError(const RMessage2& aMessage,
                                  TInt aError)
{
    aMessage.Complete(aError);
}

// Деструктор
CSimpleSession::~CSimpleSession()
{
    delete iDes;
}


```



Обычно сервер предоставляет довольно большое число различных функций. Поэтому в методе `ServiceL()` находится оператор `switch`, проверяющий с помощью метода `Function()` значение переданного в него сообщения и вызывающий необходимые клиенту функции.

-  Вызов паники в потоке, попытавшемся обратиться к несуществующей команде сервера, — стандартная практика. Разработчик клиента не должен понапрасну тратить системное время на отправку неверных сообщений и будет вынужден исправить свои ошибки.
-  Не забывайте вызывать метод `Complete()` у сообщения — это распространенная ошибка.

Как вы помните, метод `SendReceive()` имеет асинхронный вариант, но на стороне сервера запрос выполняется синхронно. Дело в том, что сервер — активный объект и вызывает метод `ServiceL()` сессии из своего, скрытого от глаз разработчика, метода `RunL()`. Поэтому вы можете отправить несколько различных сообщений серверу асинхронно (с разными объектами `TRequestStatus`, разумеется), и он будет обрабатывать их по очереди. Для того чтобы действительно обеспечить параллельное выполнение запросов клиентов, их обработку на стороне сервера нужно реализовать с помощью дополнительных активных объектов (сообщения `RMessage2` можно передавать им по ссылке). При этом желательно реализовать в сервере функциональность, позволяющую клиенту отменить ранее посланный асинхронный запрос.

-  При проектировании класса активного объекта сервера разработчик может наследовать его не от `CServer2`, а от класса `CPolicyServer` (`CPolicyServer` в свою очередь порожден от `CServer2`). Класс `CPolicyServer` позволяет удобно реализовать дополнительные проверки прав доступа клиента к тем или иным защищенным возможностям системы в зависимости от вызванной им команды сервера. В данной книге класс `CPolicyServer` не рассматривается.

Подготовка к сертификации ASD

- Знание структуры и преимуществ клиент-серверной архитектуры:
 - разделение ресурсов системы и использующего их кода;
 - предоставление доступа к ресурсу множеству клиентов;
 - управление конкурентным доступом к ресурсам и их защита.
- Понимание различных ролей системных и временных серверов. Умение определить тип сервера по предложенному примеру серверного приложения.
- Знание основ реализации клиент-серверной архитектуры в Symbian OS:
 - клиент и сервер всегда выполняются в различных потоках и чаще всего в различных процессах;
 - клиент-серверная архитектура используется для реализации асинхронных служб;
 - канал клиент-серверной коммуникации называется сессией;
 - клиент отправляет один или несколько запросов серверу;
 - клиент и сервер не могут напрямую обращаться к адресному пространству друг друга;

- запросы клиента могут быть как синхронными, так и асинхронными;
 - только один синхронный запрос клиента может выполняться в каждый момент времени, но асинхронных запросов от одного клиента может быть отправлено множество.
 - Знание роли следующих классов в клиент-серверной архитектуре: `RSessionBase`, `TIpArgs`, `TSecurityPolicy`, `RMessage2`, `CSession2`, `CServer2`, `CPolicyServer`.
 - Знание объектов, которые должны создаваться при запуске сервера.
 - Понимание механизмов защиты от подмены серверов в Symbian OS.
 - Знание основ передачи данных между клиентом и сервером при использовании синхронных и асинхронных сообщений.
 - Умение отличить правильный код для передачи данных у наследованного от класса `RSessionBase` клиента серверу.
 - Знание того, как отправляются синхронные и асинхронные клиент-серверные сообщения.
 - Знание того, как конвертируются базовые и пользовательские типы данных в объекты, пригодные для передачи на сервер в качестве изменяемых и неизменяемых аргументов запроса.
 - Понимание влияния клиент-серверных коммуникаций на производительность и условий, при которых они эффективны.
 - Умение определять сценарии, при которых лучше использовать субсессии.
 - Понимание влияния необходимости переключения контекста при клиент-серверном взаимодействии на производительность. Знание того, как организовать коммуникацию с максимальной производительностью.
-

Механизм ECom

Как уже отмечалось в разделе “Создание библиотек”, выше в этой главе, динамически загружаемые полиморфные библиотеки — простейший способ реализации системы подключаемых модулей, расширяющих или совершенствующих функциональность вашего приложения. Но при его использовании разработчик на этапе проектирования должен решить, каким образом будет выполняться поиск реализующих нужный интерфейс библиотек и выбор подходящей версии реализации.

Подключаемые модули DLL очень часто используются в системных службах. Поэтому в Symbian OS был создан единый механизм, обеспечивающий регистрацию реализующих различные интерфейсы библиотек, их поиск, контроль версий и загрузку. Это повысило надежность системных приложений и уменьшило размер их исполняемых файлов. Данный механизм называется **ECom** (EPOC Component Object Model).

На практике в пользовательских приложениях необходимость в реализации возможности расширения функциональности с помощью плагинов возникает довольно редко. Тем не менее мы не можем обойти ECom стороной, так как пе-

риодически разработчику приходится писать подобные плагины для системных служб. В виде ECom DLL создаются скринсейверы, FEP-модули (например, T9), добавляется поддержка новых форматов видео- и аудиоданных, изображений, различных коммуникационных протоколов, регистрируются новые типы файлов, реализуется конвертирование данных в экзотических кодировках и многое другое. Поэтому вы можете пропустить этот раздел и вернуться к нему тогда, когда вы действительно столкнетесь с ECom.

Общие сведения

Итак, ECom представляет собой системный сервер, осуществляющий регистрацию, выбор и загрузку плагинов для клиентских приложений. Сами плагины должны быть полиморфными DLL, содержащими одну или несколько реализаций определенного интерфейса. И интерфейс, и его реализации имеют уникальные идентификаторы, с помощью которых ECom может выполнять их поиск. Регистрация плагинов в службе ECom происходит с помощью копирования специального файла ресурсов в каталог `\Resource\Plugins\`. Клиентские приложения могут подключаться к службе ECom и запрашивать реализацию определенного интерфейса. Для этого ECom передается его идентификатор. Сервер ищет необходимую DLL, загружает ее и возвращает клиентскому приложению указатель на объект, реализующий требуемый интерфейс. Процесс выбора подходящей реализации называется **resolution** (разрешение, принятие решения), а объект, его осуществляющий — **Resolver**. Сервер ECom содержит Resolver, используемый по умолчанию, но разработчик может также создать собственный и изменить логику принятия решения.

Работа с сервером ECom ведется с помощью класса `REComSession`, реализованного в клиентской библиотеке `ecom.dll`. При его использовании к MMR-файлу проекта необходимо подключить библиотеку импорта `ecom.lib`.

Большинство методов класса `REComSession` статические. Создавать объект класса и устанавливать постоянную сессию с сервером ECom имеет смысл лишь в том случае, если вы хотите подписаться на уведомление об изменении списка зарегистрированных в нем реализаций. Для этого необходимо открыть сессию с сервером с помощью метода `OpenL()` и воспользоваться асинхронным запросом `NotifyOnChange()`. Отменить его выполнение можно с помощью метода `CancelNotifyOnChange()`, а метод `Close()` закрывает сессию с Ecom-сервером.

Класс `REComSession` предоставляет следующие статические методы.

- `ListImplementationsL()` — позволяет получить информацию о доступных реализациях указанного интерфейса. В качестве аргументов передаются идентификатор интерфейса и ссылка на объект класса `RImplInfoPtrArray`, в который будут помещаться данные о найденных реализациях. Класс `RImplInfoPtrArray` является псевдонимом для массива `RPointerArray<CImplementationInformation>`. В элементах

CImplementationInformation хранится информация о плагинах, передаваемая серверу ECom при их регистрации (идентификатор, отображаемое имя, версия и т.д.). Данный метод широко используется для построения списка доступных программе реализаций интерфейса, который впоследствии демонстрируется пользователю для выбора одной из реализаций или в информационных целях.

- Множество перегруженных методов CreateImplementationL() позволяет создать реализацию указанного интерфейса и получить обобщенный указатель на полученный объект. В качестве аргументов в метод передаются идентификатор интерфейса, а также ссылка на объект TUid либо смещение члена класса, предназначенные для хранения идентификатора реализации. Идентификатор реализации возвращается сервером ECom и впоследствии должен использоваться для уведомления сервера о его уничтожении. Метод CreateImplementationL() также позволяет указать параметры, передаваемые при создании объекта реализации.
- DestroyedImplementation() — в качестве аргумента принимает идентификатор реализации, возвращаемый методом CreateImplementationL(). Сигнализирует серверу ECom о том, что данная реализация клиентом больше не используется.
- FinalClose() — сигнализирует серверу ECom о необходимости проведения ревизии и освобождении неиспользуемых ресурсов, например, закрытия хендлов сессий и выгрузки библиотек. Этот метод обычно вызывается при завершении работы клиентского приложения. Его нельзя использовать в деструкторе класса-интерфейса.

Методы ListImplementationsL() и CreateImplementationL() имеют перегруженные варианты, позволяющие передать данные в Resolver по умолчанию или использовать другой Resolver.

Интерфейс

Мы будем рассматривать работу механизма ECom на примере гипотетического приложения-архиватора, который с помощью плагинов способен расширять количество поддерживаемых форматов файлов и алгоритмов сжатия. Предположим, что у вас имеется проект приложения, в который необходимо добавить поддержку ECom DLL. Для начала необходимо определить интерфейс, который будет реализовываться в ECom-плагилах. Допустим, что в процессе проектирования мы выяснили, что для полноценной работы нам достаточно двух методов:

- TInt Compress(TFileName& aFileName) — сжатие файла;
- TInt Decompress(TFileName& aFileName, TDes8& aResult) — распаковка файла и запись полученных данных в 8-битовый дескриптор.

Теперь мы должны объявить класс-интерфейс, содержащий эти виртуальные (обычно чисто виртуальные) методы. Для этого необходимо ответить на вопрос: “Какого типа класс необходимо использовать?” Работать с объектом через М-класс, от которого он унаследован, очень неудобно, поскольку тяжело обеспечить его правильное уничтожение в случае сброса. Лучшим решением для интерфейса ECom-плагинов является базовый С-класс. Стандартной практикой является размещение в его методах NewL () кода, создающего объект с помощью сервера ECom. Таким образом, работа с ECom и используемые идентификаторы скрываются от глаз разработчика, использующего этот интерфейс. Более того, создавать подобный класс можно точно так же, как и обычный С-класс.

Необходимый нам интерфейс может быть реализован таким образом, как показано в листинге 6.13.

Листинг 6.13. Файл ArcInterface.h.

```
/**
 * Класс-интерфейс для ECom-плагинов
 */

// Идентификатор нашего интерфейса
const TUid KArcInterfaceUid = {0xEC2213E2};

class CArcInterface : public CBase
{
public:
    // Виртуальный деструктор
    virtual ~CArcInterface();

    // Часть двухфазного конструктора
    static CArcInterface* NewL();

    // Методы интерфейса для работы с данными
    virtual TInt Compress(const TFileName& aFileName) = 0;
    virtual TInt Decompress(const TFileName& aFileName,
        TDes8& aResult) = 0;
private:
    // Уникальный идентификатор реализации интерфейса
    TUid iIDKey;
};

inline CArcInterface* CArcInterface::NewL()
{
    // Поиск и создание реализации
    TAny* obj = REComSession::CreateImplementationL(
        KArcInterfaceUid, _FOFF (CArcInterface, iIDKey));
```

```

    return reinterpret_cast<CArcInterface*> (obj);
}

// Деструктор
inline CArcInterface::~CArcInterface()
{
    // Сигнализируем серверу ECOM о том,
    // что данная реализация интерфейса уничтожается
    REComSession::DestroyedImplementation(iIDKey);
}

```

Значение идентификатора интерфейса должно быть уникальным в системе. Проще всего добиться этого, используя значение UID3 исполняемого файла.

Методы `NewLC()` и `ConstructL()` отсутствуют, так как никакого двухфазного конструирования в интерфейсе не требуется.

Обратите внимание, что деструктор класса объявлен виртуальным. Это позволяет унаследованным классам использовать собственный деструктор.

Для хранения идентификатора реализации используется член класса интерфейса `iIDKey`. Он передается в метод `CreateImplementationL()` по смещению, получаемому с помощью макроса `_FOFF`.

Приведенный в листинге 6.13 код следует сохранить именно в заголовочном файле (например, `ArcInterface.h`) папки `\inc\` проекта. Для работы с классом `REComSession` в ММР-файле необходимо подключить библиотеку импорта `ecom.lib`. Заголовочный файл `ecom.h`, содержащий объявление класса `REComSession`, находится не в папке `\include\SDK` а в подпапке `\include\ecom\`. Поэтому в ММР-файле следует добавить значение `\include\ecom` в список **System Includes** на вкладке **Options**. В качестве альтернативного способа можно заменить директиву `#include <ecom.h>` на `#include <ecom\ecom.h>`.

Наконец, в приложении необходимо добавить вызов статического метода `REComSession::FinalClose()` для освобождения неиспользуемых сервером ECom ресурсов. Сделать это лучше всего непосредственно при завершении программы. Для GUI-приложения это может быть деструктор класса `AppUi`. Для простых программ — точка входа `E32Main()` (до макроса `__UNHEAP_MARKEND`).

Подключив заголовочный файл `ArcInterface.h` к исполняемому файлу приложения, вы сможете загружать и использовать реализации интерфейса `CArcInterface` следующим образом.

```

// Обращение к ECom
CArcInterface* arc = CArcInterface::NewL();
CleanupStack::PushL(arc);
// Вызов метода реализации
User::LeaveIfError(arc->Compress(aFileName));
// Удаление объекта, уведомление ECom
CleanupStack::PopAndDestroy(arc);

```

На данный момент сервер ECom загружает первую найденную реализацию интерфейса. Несколько позднее мы усовершенствуем этот механизм.

Как видите, создавать интерфейсы для механизма ECom и использовать их реализации в приложении достаточно просто.

Реализация интерфейса, ECom DLL

Конечно, наш гипотетический архиватор пока не может функционировать. Ведь ни одной реализации интерфейса CArcInterface еще не создано. Поэтому мы создадим простейшую DLL, содержащую один класс, реализующий необходимый интерфейс. Для этого следует создать проект динамической библиотеки (тип **Basic dynamic linked library** в мастере New Symbian OS C++ Project) и удалить из нее весь автоматически генерируемый исходный код. Затем установите в поле Target type MMP-файла значение Plugin, а UID2 равным 0x10009D8D. Эти идентификаторы позволяют системе отличать ECom-плагины от прочих библиотек.

Теперь вы можете скопировать в проект файл ArcInterface.h и создать C-класс, унаследованный от класса CArcInterface. Предположим, что новая реализация интерфейса позволяет работать с архивами формата ZIP. Тогда класс может выглядеть так, как показано в листингах 6.14. и 6.15.

Листинг 6.14. Файл ArcZip.h

```
#include <e32std.h>
#include <e32base.h>
#include "ArcInterface.h"

/*
 * CArcZip - реализация работы с ZIP-архивами
 */

class CArcZip : public CArcInterface
{
public:
    ~CArcZip();
    static CArcZip* NewL();

    // Методы интерфейса CArcInterface
    TInt Compress(const TFileName& aFileName);
    TInt Decompress(const TFileName& aFileName,
                    TDes8& aResult);

private:
    CArcZip();
    void ConstructL();
};
```

Листинг 6.15. Файл ArcZip.cpp

```
#include "ArcZip.h"

CArcZip::CArcZip()
{
}

CArcZip::~CArcZip()
{
}

CArcZip* CArcZip::NewL()
{
    CArcZip* self = new (ELeave) CArcZip();
    CleanupStack::PushL(self);
    self->ConstructL();
    CleanupStack::Pop(self);
    return self;
}

void CArcZip::ConstructL()
{
}

TInt CArcZip::Compress(const TFileName& aFileName)
{
    // Реализация сжатия файла
}

TInt CArcZip::Decompress(const TFileName& aFileName,
    TDes8& aResult)
{
    // Реализация распаковки файла
}
```

Обратите внимание на приведенный в примере класс `CArcZip`. Он несколько отличается от шаблона, генерируемого для C-класса средой разработки Carbide. с++ (помимо наследования). В нем отсутствует метод `NewLC()`, и вся логика двухфазного конструктора сосредоточена в `NewL()`. Причина в том, что созданием объектов этого класса занимается сервер `ECom` и ему не требуется метод, оставляющий объект в стеке очистки.



Так как плагин нами создается в демонстрационных целях, реализация методов для работы с ZIP-файлами не приводится. Но я хочу отметить, что в состав Symbian OS входит порт библиотеки ZLIB и ряд классов, обеспечивающих работу с ней. Например, `CZipFile` из `zipfile.h`.

Имея в DLL реализацию интерфейса `CArcInterface`, необходимо ответить на вопрос, каким образом сервер `ECom` сможет получить к ней доступ. Очень просто — он ожидает, что первая (и единственная), экспортируемая динамической библиотекой функция возвращает указатель на массив, элементами которого являются пара значений *<идентификатор_реализации, указатель_на_метод_для_ее_создания>*. В качестве элементов массива используются структуры типа `TImplementationProxy`, объявленного в файле `ImplementationProxy.h`. Полями `TImplementationProxy` являются `iImplementationUid` типа `TUid` и `iNewLFuncPtr` типа `TProxyNewLPtr`. В свою очередь `TProxyNewLPtr` является псевдонимом для обобщенного указателя `TAny*`. Для удобства в том же заголовочном файле объявлен макрос `IMPLEMENTATION_PROXY_ENTRY`, позволяющий создать объект `TImplementationProxy` на основании значения идентификатора и указателя на функцию. Учитывая все вышеназванное, записать необходимый нам массив можно следующим образом.

```
#include <ImplementationProxy.h>

const TImplementationProxy ImplementationTable[] =
{
    IMPLEMENTATION_PROXY_ENTRY(0xE0003333, CArcZip::NewL)
};
```

В данном случае значение идентификатора реализации выбрано случайным образом и должно быть уникальным. Если в библиотеке находятся несколько классов-реализаций, все они должны быть перечислены в этом массиве. Предположим, что мы создали класс `CArcRar` для работы с архивами формата RAR. В этом случае таблица реализаций примет вид.

```
const TImplementationProxy ImplementationTable[] =
{
    IMPLEMENTATION_PROXY_ENTRY(0xE0003333, CArcZip::NewL),
    IMPLEMENTATION_PROXY_ENTRY(0xE0003334, CArcRar::NewL)
};
```

Идентификатор реализации `CArcRar` выбран случайным образом и должен быть уникален.

Экспортируемая библиотекой функция должна иметь следующий вид.

```
const TImplementationProxy* foo(TInt& aTableCount);
```

В аргумент `aTableCount` следует поместить количество записей в массиве. Общепринятый вариант реализации этой функции приведен ниже.

```
// Экспортируем таблицу разрешения методов реализаций
EXPORT_C const TImplementationProxy*
    ImplementationGroupProxy( TInt& aTableCount)
{
    aTableCount = sizeof(ImplementationTable) /
```

```

        sizeof(TImplementationProxy);
    return ImplementationTable;
}

```

Не забывайте о сложностях, связанных с заморозкой проекта при добавлении новых экспортируемых функций. Созданную библиотеку, как и исполняемый файл приложения, необходимо размещать в каталоге `\sys\bin\`.

Регистрация ECom DLL

Итак, мы получили библиотеку, которая может быть загружена сервером ECom. Но несмотря на то, что система может отличить ECom-плагин от прочих библиотек по паре параметров `<UID1, UID2>`, загружать их по очереди в процессе поиска реализаций интерфейсов нерационально. Поэтому в ECom используется механизм регистрации, позволяющий предоставить данные о содержащихся в библиотеке реализациях.

Для регистрации нового плагина используется файл ресурсов специального формата, размещаемый в каталоге `\resource\plugins\` и имеющий то же имя, что и библиотека. Мы уже несколько раз сталкивались с подобным способом регистрации в предыдущих разделах, поэтому подробно останавливаться на процессе создания и установки файла ресурсов не будем.

Регистрационный ресурс ECom DLL должен содержать запись структуры `REGISTRY_INFO`, объявленную в заголовочном файле `RegistryInfo.rh`. В ней имеется всего два поля: `dll_uid` и `interfaces`. Поле `dll_uid` является числовым и должно содержать `UID3` библиотеки. Это дополнительный контрольный механизм (помимо имени файла) для установления связи между библиотекой и ее регистрационным ресурсом. Второе поле `interfaces` содержит массив записей структуры `INTERFACE_INFO`, объявленной в том же заголовочном файле, что и `REGISTRY_INFO`.

Запись `INTERFACE_INFO` предназначена для хранения информации об интерфейсе, реализации которого содержатся в библиотеке. В структуре `INTERFACE_INFO` также определены лишь два поля: `interface_uid` и `implementations`. Как несложно догадаться, поле `interface_uid` предназначено для хранения идентификатора интерфейса, а поле `implementations` представляет собой массив записей структуры `IMPLEMENTATION_INFO` либо `BINARY_IMPLEMENTATION_INFO`.

Запись структуры `IMPLEMENTATION_INFO` предназначена для описания конкретной реализации интерфейса. Именно эти данные может получить клиент из классов `CImplementationInformation` с помощью статического метода `REComSession::ListImplementationsL()`. Структура `IMPLEMENTATION_INFO` имеет следующий формат:

- `implementation_uid` — значение идентификатора реализации интерфейса;
- `version_no` — номер версии реализации;

- `display_name` — отображаемое имя реализации (например, "Jabber protocol", или "Zip archive");
- `default_data` — строка; данные, используемые механизмом Resolver при выборе подходящей реализации;
- `opaque_data` — строка; дополнительные данные для Resolver.

Структура `BINARY_IMPLEMENTATION_INFO` отличается от `IMPLEMENTATION_INFO` лишь тем, что поля `default_data` и `opaque_data` в ней являются не строковыми, а массивами байт.

Учитывая все вышесказанное, регистрационный файл для создаваемой нами ECom DLL должен иметь следующий вид.

```
#include "RegistryInfo.rh"

RESOURCE REGISTRY_INFO
{

    // UID3 ECom плагина, см. mmp файл
    dll_uid = 0xE18F0E7A;

    // Список реализованных интерфейсов
    interfaces =
    {
        INTERFACE_INFO
        {

            // UID интерфейса. Должен быть равен
            // KArcInterfaceUid из ArcInterface.h
            interface_uid = 0xEC2213E2;

            implementations =
            {
                IMPLEMENTATION_INFO
                {
                    // UID реализации
                    implementation_uid = 0xE0003333;
                    // Версия реализации
                    version_no = 1;
                    // Отображаемое имя
                    display_name = "ZIP archive support";
                    // Данные для выбора
                    default_data = "ZIP";
                    opaque_data = "";
                }
            };
        }
    };
}
```

Теперь наш плагин при установке на устройство будет регистрироваться в службе ECom и может быть загружен сервером.

Выбор реализаций

На данный момент наш гипотетический архиватор загружает первую попавшуюся реализацию интерфейса CArcInterface. Естественно, это необходимо исправить. Как вы знаете, служба ECom имеет собственный механизм Resolver, предназначенный для выбора подходящей реализации. Этот выбор осуществляется на основании сопоставления данных, переданных в методе REComSession:: CreateImplementationL(), и данных, указанных в поле default_data записи регистрационного ресурса интерфейса.

Рассмотрим случай, при котором у нас имеются несколько реализаций в одной ECom DLL, выбор которых мы и хотим осуществлять. Предположим, что в нашей библиотеке реализованы два порожденных от CArcInterface класса: CArcZip и CArcRar. Экспортируемая из DLL измененная таблица реализаций для этого случая уже приводилась ранее. Запись INTERFACE_INFO в регистрационном ресурсе должна выглядеть следующим образом.

```
INTERFACE_INFO
{
    // UID интерфейса. Должен быть равен
    // KArcInterfaceUid из ArcInterface.h
    interface_uid = 0xEC2213E2;

    implementations =
    {
        IMPLEMENTATION_INFO
        {
            implementation_uid = 0xE0003333;
            version_no = 1;
            display_name = "ZIP archive support";
            default_data = "ZIP";
            opaque_data = "";
        },

        IMPLEMENTATION_INFO
        {
            implementation_uid = 0xE0003334;
            version_no = 1;
            display_name = "RAR archive support";
            default_data = "RAR";
            opaque_data = "";
        }
    };
}
```

Теперь добавим в интерфейс перегруженный метод `NewL()`, с помощью которого мы могли бы осуществлять выбор одной из этих реализаций.

```
class CArcInterface : public CBase
{
...
    static CArcInterface* NewL(const TDesC8& aParam);
...
}

inline CArcInterface* CArcInterface::NewL(
                                const TDesC8& aParam)
{
    // Поиск и создание реализации
    TComResolverParams params;
    params.SetDataType(aParam);

    TAny* obj =
        REComSession::CreateImplementationL(KArcInterfaceUid,
        _FOFF(CArcInterface, iDtor_ID_Key), params);
    return reinterpret_cast<CArcInterface*>(obj);
}
```

Данный метод демонстрирует использование перегруженного метода `REComSession::CreateImplementationL()`, позволяющего передать системному механизму Resolver информацию о необходимой реализации. Параметры передаются с помощью типа `TComResolverParams`, который имеет всего несколько методов.

- `SetDataType()` — задает параметр. Параметром является 8-битовый дескриптор.
- `SetGenericMatch()` или `SetWildcardMatch()` — установка флага (по умолчанию сброшен), сигнализирующего о том, что клиент согласен на получение “общей реализации”.
- `IsGenericMatch()` или `IsWildcardMatch()` — проверка значения флага “общей реализации”.

Передаваемые в Resolver сервера ECom данные будут сверяться со значением поля `default_data` записи регистрационного ресурса интерфейса. В результате клиентское приложение может загружать различные реализации следующим образом.

```
CArcInterface* zip = CArcInterface::NewL(_L8("ZIP"));
CArcInterface* rar = CArcInterface::NewL(_L8("RAR"));
```

Значения, необходимые для обращения к доступным реализациям интерфейса, можно получить программно с помощью метода `REComSession::ListImplementationsL()`. Рассмотренный нами пример подходит также и для случая, когда реализации находятся в разных динамических библиотеках: сервер ECom полностью скрывает от клиента истинное местоположение реализации.

По умолчанию Resolver службы ECom ищет реализацию, регистрационные данные поля `default_data` которой в точности соответствуют переданному параметру. Флаг, задаваемый с помощью метода `SetGenericMatch()`, позволяет искать этот параметр как подстроку поля `default_data`. Таким образом, подходящей может оказаться более “общая” реализация, обеспечивающая работу сразу с несколькими форматами архивов. Например: реализация со значением `default_data`, равным "ZIP, RAR, ARJ, Z, CAB, LZH".

Resolver

В случае если логика выбора реализаций стандартного механизма Resolver сервера ECom вас не устраивает, вы можете создать собственный. Для этого необходимо разработать ECom DLL, содержащую реализацию системного интерфейса `CResolver`, объявленного в `resolver.h` и имеющего идентификатор `KCustomResolverInterfaceUid(0x10009D90)`. Интерфейс `CResolver` содержит чистые виртуальные методы для выбора и получения списка реализаций на основании передаваемых в них параметров `TEComResolverParams`. В унаследованном от `CResolver` классе вы можете реализовать в них собственную логику.

После того как содержащая реализацию механизма Resolver библиотека будет установлена на устройство и зарегистрирована в службе ECom, клиентское приложение сможет указывать идентификатор этой реализации в качестве аргументов перегруженных методов `REComSession::ListImplementationsL()` и `REComSession::CreateImplementationL()` для использования нестандартной логики выбора плагинов.

Хороший пример реализации интерфейса `CResolver` можно найти в подготовленном Forum Nokia проекте “S60 Platform: ECom Plug-In Examples”, ссылка на который приводится в разделе “Документация” в конце книги.

Распознаватели

Для определения и сопоставления типов файлов с использующими их приложениями в Symbian OS применяются специальные библиотеки — **распознаватели** (recognizers). В Symbian 6.x–8.x они были полиморфными библиотеками с расширением “.mdl”, а в Symbian 9.x распознаватели стали настоящими ECom-плагинами.

При запуске системы сервером архитектуры приложений AppArcServer производится поиск всех установленных реализаций распознавателей. Составляется список доступных реализаций, упорядоченный по их приоритету, и список поддерживаемых ими MIME-типов файлов. Способность распознавателей загружаться при старте системы использовалась в старых версиях Symbian для запуска приложений. В Symbian 9.x это запрещено — для автозапуска должно использоваться специальное Startup List Management API, иначе приложение не пройдет сертификацию Symbian Signed.

В тот момент, когда системе необходимо определить тип файла, она начинает опрашивать все реализации распознавателей в порядке приоритета, передавая им имя файла и небольшой блок прочитанных из его начала данных. Размер буфера по умолчанию равен 256 байт, но может меняться распознавателем. На основании переданных ему данных распознаватель должен определить MIME-тип файла. Если ему это удалось, то MIME-тип передается системе, а распознаватель устанавливает уровень “уверенности” в своем решении. Система может продолжать опрашивать распознаватели до тех пор, пока необходимый уровень “уверенности” не будет достигнут.

Если MIME-тип файла определен, а необходимый уровень уверенности достигнут, то система вызывает поддерживающее данный тип файлов приложение, передавая в него имя файла. Если таких приложений несколько, то выбирается программа, заявившая о наибольшем приоритете на обработку данного типа файлов. Информация о поддерживаемых приложением MIME-типах и их приоритетах указывается в виде массива записей DATATYPE в регистрационном файле ресурса. Пример.

```
RESOURCE APP_REGISTRATION_INFO
{
    app_file="HelloWorld";
    localisable_resource_file = "\\resource\\apps\\HelloWorld";
    localisable_resource_id = R_LOCALISABLE_APP_INFO;
    datatype_list =
    {
        DATATYPE
        {
            priority = EDataTypePriorityHigh;
            type = "image/tiff";
        }
    };
}
```

Распознаватель должен быть ECom DLL, содержащей одну или более реализаций базового класса CApaDataRecognizerType. Уникальный идентификатор реализуемого интерфейса равен 0x101F7D87. Библиотека должна декларировать доступ к защищенной возможности ProtServ.

Абстрактный базовый класс CApaDataRecognizerType содержит следующие методы, которые необходимо реализовать.

- Конструктор, в который необходимо передать UID реализации и ее приоритет (значение CApaDataRecognizerType::TRecognizerPriority).
- Статический метод CreateRecognizer(), создающий объект реализации и возвращающий указатель на него.
- Метод PreferredBufSize(), возвращающий количество байт, необходимых распознавателю для определения типа файла по его содержанию. Реализация по умолчанию возвращает 0.

- `MimeTypesCount()` — возвращает количество MIME-типов, определение которых поддерживает распознаватель. Реализация по умолчанию возвращает 1.
- `SupportedDataTypeL()` — метод, получающий от системы индекс MIME-типа (от 0 до `MimeTypesCount() - 1`) и возвращающий его описание. Описание MIME-типа содержится в объекте `TDataType`, принимающим в конструкторе 8-битовый дескриптор с его именем.
- `DoRecognizeL()` — в качестве аргументов принимает имя распознаваемого файла и буфер с частью его данных. Метод не возвращает значения, вы должны поместить определенный MIME-тип в член класса `iDataType`, а степень уверенности в `iConfidence`. Они будут проверены системой по завершении работы метода. Если определить тип не удалось, в `iConfidence` помещается значение `ENotRecognized`.

В качестве MIME-типа могут использоваться не только те, которые зарегистрированы IANA⁶, но и собственные, а также произвольные строковые идентификаторы.

Хороший пример реализации распознавателя и приложения, обрабатывающего определенный им тип файла, можно найти в проекте “S60 Platform: Document Handler Example”, подготовленном Forum Nokia. Ссылка на него приводится в конце книги.

Подготовка к сертификации ASD

- Умение определять правильность утверждений о роли распознавателей в Symbian OS.
-

Работа со временем

Для работы со временем и датой в Symbian C++ используется класс `TTime`. Он инкапсулирует член типа `TInt64`, хранящий количество микросекунд, прошедших с 00:00 1-го января 0 г. н.э. Отрицательные значения соответствуют времени до н.э. Создавать переменную класса `TTime` можно в стеке, как и любой T-класс. При этом для инициализации в конструктор класса можно передать аргумент типа `TInt64`, содержащий время. Пример.

```
TInt64 t(36000000); // 01:00 1-го янв. 0-го года
TTime time(t);
```

Чаще всего возникает необходимость инициализировать объект `TTime` текущим временем и датой. Для этого класс `TTime` предоставляет следующие методы.

⁶ <http://www.iana.org/assignments/media-types/>

- `HomeTime()` — инициализирует объект текущим временем, в соответствии с выбранными в настройках системы часовым поясом и режимом летнего/зимнего времени. Это именно то время, которое отображается на системных часах.
- `UniversalTime()` — инициализирует объект текущим универсальным временем (UTC, Coordinated Universal Time), совпадающим со временем по Гринвичу. Значение `HomeTime()` в восточных часовых поясах будет больше, а в западных — меньше UTC.

Пример.

```
TTime time;
time.HomeTime(); // Текущее время
TTime time2;
time2.UniversalTime(); // Время UTC
```

Смещение между значениями, предоставляемыми методами `HomeTime()` и `UniversalTime()`, нельзя получить арифметическим вычитанием, так как между их вызовами часы уже ушли вперед. Для определения смещения необходимо воспользоваться методом `UTCOffset()` статического класса `User`.

Следующие служебные методы класса `TTime` позволяют получить информацию о хранящемся в нем значении.

- `Int64()` — возвращает время в виде числа, содержащего количество микросекунд.
- `DayNoInWeek()` — номер текущего дня в неделе. Понедельник считается нулевым днем.
- `DayNoInMonth()` — день месяца.
- `DaysInMonth()` — число дней в месяце.
- `DayNoInYear()` — номер дня в году. По умолчанию 1-е января считается первым днем. Существует вариант метода, позволяющий задать произвольное время начала года.
- `WeekNoInYear()` — номер недели в году. Существует вариант метода, позволяющий задать произвольное время начала года. Помимо этого, можно указать правило, в соответствии с которым определяется первая неделя года: первой считается либо неделя, содержащая первый день года, либо первая полная (семидневная) неделя, либо содержащая не менее четырех дней нового года (по умолчанию).

Для более удобной работы со временем существует класс `TDateTime`, представляющий дату и время с помощью семи компонентов: год, месяц, день, час, минута, секунда и микросекунда. Создать объект можно в стеке. Для инициализации в конструктор могут быть переданы все семь параметров. Но чаще всего экземпляры `TDateTime` получают из объекта `TTime` с помощью метода `DateTime()`. В классе `TTime` также определен оператор присваивания для инициализации из `TDateTime`. Пример.

```

TTime time;
time.HomeTime(); //Текущая дата и время (dd.mm.yyyy aa:bb:cccc)
TDateTime dt = time.DateTime();
dt.SetYear(1984);
time = dt; // time = (dd.mm.1984 aa:bb:cccc)

```

Класс TDateTime предоставляет следующие методы.

- Set() — позволяет задать значения всех семи компонентов.
- SetMicroSecond() и MicroSecond() — позволяют установить и получить количество микросекунд во времени.
- SetSecond() и Second() — позволяют установить и получить количество секунд во времени.
- SetMinute() и Minute() — позволяют установить и получить количество минут во времени.
- SetHour() и Hour() — позволяют установить и получить количество часов во времени.
- SetDay() и Day() — позволяют установить и получить день даты.
- SetMonth() и Month() — позволяют установить и получить месяц даты.
- SetYear() — изменение года, Year() — возвращает год.
- SetYearLeapCheck() — позволяет установить год с проверкой, является ли он високосным. Если день и месяц даты представляют собой 29-е февраля, то при попытке изменить год на не високосный метод вернет код ошибки KErrNone.

В классе TTime определены операторы сравнения и удобный метод, округляющий время до минут: RoundUpToNextMinute(). Действительно, секунды и микросекунды часто не нужны и мешают при грубом сравнении двух объектов TTime.

Помимо операторов сравнения, в классе TTime определены операторы вычитания и сложения, а также их аналоги с присваиванием. С их помощью время можно увеличить на некоторый интервал. Для задания интервалов в Symbian C++ определен ряд простых классов, перечисленных ниже и содержащих поле целочисленного типа для хранения величины интервала и определяющих ряд операторов сравнения и присваивания: TTimeIntervalYears, TTimeIntervalMonths, TTimeIntervalDays, TTimeIntervalHours, TTimeIntervalMinutes, TTimeIntervalSeconds, TTimeIntervalMicroSeconds32, TTimeIntervalMicroSeconds. Все они, кроме класса TTimeIntervalMicroSeconds, используют объект TInt32 для хранения интервала. В классе TTimeIntervalMicroSeconds для этой цели применяется объект TInt64. Пример.

```

TTime time;
time.HomeTime(); // Текущая дата и время
TTimeIntervalYears yrs(3);
TTimeIntervalHours hrs(3);

```

```
time += yrs; // Время увеличено на 3 года
time -= hrs; // а затем, уменьшено на 3 часа
```

Следует заметить, что используемые в классах-интервалах операторы сравнения не учитывают тип интервала, а сравнивают лишь его величину, поэтому `yrs` будет равен `hrs`.

Для инициализации объекта `TTime` из дескриптора используется метод `Parse()`. Преобразование времени в строковое представление осуществляется с помощью метода `FormatL()`. Оба метода довольно сложны в использовании: существует огромное количество нюансов того, как должны быть записаны компоненты даты и времени в строке, и еще больше команд форматирования для ее получения. Мы не будем останавливаться на них подробно — эту информацию можно почерпнуть в справочнике SDK. Ограничимся лишь небольшим примером использования этих методов.

```
TTime time;
TInt res = time.Parse(_L("26/11/1984 20:00"));
// time = 20:00 26.11.1984
if (res & EParseDatePresent && res & EParseTimePresent)
{
    TBuf<20> buf;
    time.FormatL(buf, _L("%F%D/%M/%Y %H:%S"));
    // buf = "26/11/1984 20:00"
}
```

Таймеры

Системный таймер находится в микроядре (точнее, в наноядре (`nanokernel`)) и основан на подсчете тактов. С его помощью реализуется множество фундаментальных служб Symbian OS, например, механизм вытеснения равноприоритетных потоков. Устройство системного таймера чрезвычайно сложно и рассматривать его мы не будем, тем более, что разработчик никогда не использует таймер микроядра напрямую. Если же вам все же посчастливится программировать службы уровня ядра Symbian OS, то все необходимое о таймере наноядра вы можете почерпнуть из книги Джейн Сейлес (Jane Sales) “Symbian OS Internals”.

В приложении доступ к службам системного таймера осуществляется с помощью класса `RTimer`. Он является поставщиком асинхронных сервисов и предназначен для использования в активных объектах. С помощью класса `RTimer` можно отправлять следующие асинхронные запросы:

- запрос об уведомлении через некоторый интервал времени;
- запрос об уведомлении по наступлению определенного системного времени;
- запрос об уведомлении в определенную долю секунды;
- запрос об уведомлении о неактивности пользователя на протяжении некоторого интервала времени;
- запрос об уведомлении через некоторое количество тактов.

Как и любой R-класс, экземпляр объекта класса `RTimer` можно создавать как в стеке, так и в куче. Но, так как таймер обычно используется как поставщик асинхронного сервиса в активном объекте, его обычно объявляют членом класса этого объекта. Таким образом, объект `RTimer` создается в куче вместе с экземпляром C-класса.

Перед использованием разработчик должен проинициализировать ресурс, к которому собирается обращаться через R-класс. В `RTimer` для этого используется метод `CreateLocal()`. По окончании работы с объектом `RTimer` вы должны вызвать его метод `Close()`, чтобы освободить занимаемый им ресурс.

Точность системного таймера зависит от аппаратного обеспечения и обычно равна одной миллисекунде. Но в большинстве предоставляемых классом `RTimer` методов точность не столь высока и составляет 1/64 секунды.

Итак, класс `RTimer` предоставляет следующие методы.

- `CreateLocal()` — инициализация таймера.
- `At()` — запрос об уведомлении по наступлению определенного системного времени. Время указывается в виде аргумента типа `TTime`. Следует отметить, что если во время исполнения запроса системное время будет изменено пользователем или сторонней программой, то он завершится с ошибкой `KErrAbort`. Если в качестве аргумента будет передан уже прошедший момент времени, то запрос будет тут же завершен с кодом `KErrUnderflow`.
- `AtUTC()` — аналогичен `At()` с тем исключением, что указывается универсальное, а не местное время.
- `After()` — запрос об уведомлении через некоторое количество микросекунд.
- `HighRes()` — запрос об уведомлении через некоторое количество микросекунд с повышенной точностью (до одной микросекунды).
- `AfterTicks()` — запрос об уведомлении через некоторое количество тактов.
- `Lock()` — запрос об уведомлении в определенную долю секунды. Время не указывается — асинхронный запрос завершается, как только таймер окажется в нужной доле любой секунды. Точность составляет 1/12 секунды.
- `Inactivity()` — запрос об уведомлении о неактивности пользователя на протяжении некоторого интервала времени (задается в секундах). В системе есть специальный таймер, отслеживающий периоды активности пользователя. Его значение можно получить с помощью метода `User::InactivityTime()`. Этот таймер используется, например, для выключения подсветки экрана. Если во время отправки асинхронного запроса пользователь уже неактивен более длительный промежуток времени, чем указанный разработчиком, то запрос завершится только в следующем периоде неактивности.
- `Cancel()` — отмена текущего асинхронного запроса.
- `Close()` — отключение таймера, освобождение системных ресурсов.

Пользоваться классом `RTimer` довольно просто. Более того, мы уже встречались с примером его использования при изучении активных объектов. Если вы создадите в проекте активный объект с помощью шаблона, то в нем уже будет использоваться объект `RTimer` и его асинхронная функция `After()` (см. главу 5, листинги 5.2 и 5.3).

Интересно, что свойство методов `After()` и `AfterUTC()` завершаться с ошибкой при изменении системного времени иногда используют для отслеживания этого события. Вот небольшой пример⁷ активного объекта, уведомляющего слушателя `iObserver` об изменении системного времени (листинги 6.16 и 6.17). Обратите внимание, что подписчик реализован через `M`-класс, обеспечивающий интерфейс функций обратного вызова. Ссылка на объект, наследованный от класса `MObserver`, должна передаваться в двухфазный конструктор класса `CMyTimer`.

Листинг 6.16. Файл `MyTimer.h`

```
#include <e32base.h>
#include <e32std.h>

class MObserver // Интерфейс подписчика на событие
{
public:
    // Системное время изменилось
    virtual void SystemTimeChanged() = 0;
    // Таймер остановился в результате ошибки
    virtual void TimerStopped(TInt) = 0;
};

class CMyTimer : public CActive
{
public:
    ~CMyTimer();
    static CMyTimer* NewL(MObserver& aObserver);
    static CMyTimer* NewLC(MObserver& aObserver);

public:
    void Start(); // Не StartL(), т.к. не вызывает сброс

private:
    CMyTimer(MObserver& aObserver);
    void ConstructL();

private:
    void RunL();
    void DoCancel();
};
```

⁷ TSS000261 — Timer issues and tips

```
private:
    RTimer iTimer; // Таймер
    MObserver& iObserver; // Подписчик
};
```

Листинг 6.17. Файл MyTimer.cpp

```
#include "MyTimer.h"

CMyTimer::CMyTimer(MObserver& aObserver) :
    CActive(EPriorityStandard), iObserver(aObserver)
{
}

CMyTimer* CMyTimer::NewLC(MObserver& aObserver)
{
    CMyTimer* self = new (ELeave) CMyTimer(aObserver);
    CleanupStack::PushL(self);
    self->ConstructL();
    return self;
}

CMyTimer* CMyTimer::NewL(MObserver& aObserver)
{
    CMyTimer* self = CMyTimer::NewLC(aObserver);
    CleanupStack::Pop(); // self;
    return self;
}

void CMyTimer::ConstructL()
{
    // Инициализация таймера
    User::LeaveIfError(iTimer.CreateLocal());
    CActiveScheduler::Add(this);
}

void CMyTimer::Start()
{
    Cancel();
    // Получаем текущее время
    TTime now; now.HomeTime();
    // Запускаем таймер на 1 час (можно и больше)
    iTimer.At(iStatus, now + TTimeIntervalHours(1));
    SetActive();
}

void CMyTimer::RunL()
```

```

{
switch (iStatus.Int())
{
case KErrAbort:
    // Системное время изменилось
    iObserver.SystemTimeChanged();
    break;

case KErrNone:
    // Час прошел, повторяем запрос
    Start();
    break;

default:
    // Уведомление о прочих ошибках
    iObserver.TimerStopped(iStatus.Int());
    break;
}
}

CMyTimer::~CMyTimer()
{
    Cancel();
    iTimer.Close(); // Освобождаем ресурсы таймера
}

void CMyTimer::DoCancel()
{
    iTimer.Cancel();
}

```

Следует помнить, что обработчик `RunL()` активного объекта может быть вызван позже, чем произойдет завершение асинхронного запроса, если его вызов отложен планировщиком.



Не следует превращать асинхронные методы `RTimer::After()` и `RTimer::At()` в синхронные с помощью метода `User::WaitRequest()`. Для этого существуют отдельные методы `User::After()` и `User::At()`.

Очень часто при проектировании приложения возникает потребность в периодическом вызове какой-либо функции. Для этого также может использоваться активный объект со встроенным объектом `RTimer`. Но объявление нового класса для такой простой задачи выглядит излишне сложным решением. Это увеличивает число файлов в проекте и делает код более запутанным. Поэтому в Symbian C++ есть два уже готовых простых C-класса для реализации периодичности: `CPeriodic` и `CHeartbeat`. Базовым для них является класс `CTimer` — частично реализованный активный объект.

Класс `CTimer` содержит объект класса `RTimer` в качестве своего члена, реализует его инициализацию (в конструкторе `ConstructL()`), освобождение ресурсов (в деструкторе) и отмену асинхронных запросов (в методе `DoCancel()`). Класс `CTimer` также предоставляет все методы класса `RTimer`, подставляя в них свой `iStatus` для получения извещения о завершении запроса. Частично реализованным он является потому, что в нем отсутствует метод `RunL()`. По этой причине нельзя создать экземпляр объекта `CTimer` — этот класс используется только при наследовании. Класс `CTimer` позволяет вызвать асинхронную функцию содержащегося в нем объекта `RTimer` и получить результат в свой обработчик `RunL()`. Например, мы можем переделать созданный ранее класс `CTimer`, взяв за основу класс `CTimer`. Для этого выполните следующее.

- Замените все упоминания о классе `CActive` на `CTimer` (класс `CTimer` будет наследоваться от него).
- Удалите методы `ConstructL()`, `DoCancel()` и деструктор класса.
- Уберите объявление `iTimer` из заголовочного файла (такой член есть в классе `CTimer`).
- В методе `Start()` вместо метода `iTimer.At(...)` введите следующее.

```
At(now + TTimeIntervalHours(1));
```

Как видите, кода в нашем классе стало меньше, но это не сделало разработку проще, — наоборот, мы существенно ограничили свою свободу действий. По этой причине порожденные от класса `CTimer` пользовательские активные объекты используются редко. Основное предназначение класса `CTimer` — быть базовым для двух других системных классов. Они в свою очередь являются полностью реализованными активными объектами и используются совсем по-другому.

Первым таким классом является `CPeriodic`. С его помощью можно организовать периодический вызов заданной функции через определенные промежутки времени. Для создания экземпляра класса `CPeriodic` используются методы двухфазного конструктора. При этом в качестве аргумента передается приоритет активного объекта.

```
CPeriodic* p = CPeriodic::NewL(CActive::EPriorityStandard);
```

Таймер класса `CPeriodic` запускается с помощью метода `Start()`, в который передаются задержка перед началом отсчета периодов, интервал периода и функция обратного вызова. Функция обратного вызова должна иметь вид:

```
TInt Func(TAny*);
```

и быть статическим методом класса либо не принадлежать классу. В качестве результата выполнения функции должно возвращаться число. В случае, если оно отлично от нуля (`ETrue`), работа объекта `CPeriodic` будет продолжена, и функция снова будет вызвана через отведенный промежуток времени. Если результат нулевой (`EFalse`), работа объекта `CPeriodic` прекращается. Для

передачи функции обратного вызова в метод `Start()` класса `CPeriodic` используется объект типа `TCallBack`. Он создается следующим образом.

```
TInt EventProcessor(TAny*)
{ // Метод обратного вызова, не принадлежащий классу
  return ETrue; // Продолжаем работу CPeriodic
}

void CSomeObserver::StartTimer()
{
    // Помещаем наш метод обратного вызова в TCallBack
    TCallBack callback(EventProcessor);
    <...> // Запуск CPeriodic
}
```

В качестве второго аргумента конструктора класса `TCallBack` может быть использован произвольный указатель. Он будет передаваться в функцию обратного вызова в качестве аргумента. Обычно в качестве указателя используется `this` для связи статического метода с экземпляром класса. Соответствующий пример приведен в листинге 6.18.

Листинг 6.18. Пример использования класса `CPeriodic`

```
/*
 * Объявление класса
 */

class CSomeObserver : public CBase
{
public:
    <...> // Опущены конструкторы и пр.
    void ConstructL();
    // Запуск таймера CPeriodic
    void StartTimer();
private:
    ~CSomeObserver();
    // Статический метод обратного вызова
    static TInt EventProcessor(TAny*);
private:
    CPeriodic* iPeriodic;
};

/*
 * Реализация методов
 */

void CSomeObserver::ConstructL()
{
```

```

        // Создание таймера
        iPeriodic = CPeriodic::NewL(CActive::EPriorityStandard);
    }

TInt CSomeObserver::EventProcessor(TAny *aArg)
{
    // Доступ к классу можно получить через
    // CSomeObserver* obj = static_cast<CSomeObserver>(aArg);
    <...> // Делаем что-нибудь
    return ETrue; // Для остановки CPeriodic верните EFalse
}

void CSomeObserver::StartTimer()
{
    // Запуск периода немедленно
    const TTimeIntervalMicroSeconds32 KDelay = 0;
    // Период в 1 минуту
    const TTimeIntervalMicroSeconds32 KPeriod = 60000000;

    // Помещаем наш метод обратного вызова в TCallback
    TCallback callback(EventProcessor, this);

    // Запуск периодического вызова EventProcessor
    iPeriodic->Start(KDelay, KPeriod, callback);
}

CSomeObserver::~CSomeObserver()
{
    if (iPeriodic)
        iPeriodic->Cancel();
    delete iPeriodic;
}

```

Вторым порожденным от CTimer классом является CHeartbeat. Он так же, как и класс CPeriodic, является готовым к использованию активным объектом и предназначен для периодического уведомления подписчика о наступлении определенной доли секунды. Таким образом, он основан на использовании асинхронной функции RTimer::Lock(). Объект класса CHeartbeat вызывает метод обратной связи каждую секунду, но, так как он является высокоточным инструментом, то способен также уведомлять о произошедшей рассинхронизации (например, вследствие отложенного планировщиком вызова RunL()). Поэтому для связи с подписчиком вместо объекта TCallback используется интерфейс MBeating, содержащий два виртуальных метода. Остановить запущенный объект класса CHeartbeat можно только с помощью метода Cancel(). Соответствующий пример приведен в листинге 6.19.

Листинг 6.19. Пример использования класса CHeartbeat

```

/*
 * Объявление класса
 */

class CSomeObserver : public CBase, MBeating
{
public:
    <...> // Опущены конструкторы и пр.
    void ConstructL();
    // Запуск таймера CHeartbeat
    void StartTimer();
    // Остановка таймера CHeartbeat
    void CancelTimer();
private:
    // Методы интерфейса MBeating
    void Beat();
    void Synchronize();
private:
    ~CSomeObserver();
private:
    CHeartbeat* iHeartbeat;
};

/*
 * Реализация методов
 */

void CSomeObserver::ConstructL()
{
    // Создание таймера
    iHeartbeat = CHeartbeat::NewL(CActive::EPriorityStandard);
}

void CSomeObserver::Beat()
{
    <...> // Выполнение каких-либо операций
}

void CSomeObserver::Synchronize()
{
    // Произошла рассинхронизация, несколько тактов пропущены
}

void CSomeObserver::StartTimer()
{
    //Запуск таймера,
    //Срабатывание в 3/12 доли каждой секунды
}

```

```

//this-указатель на класс, реализовавший интерфейс MBeating
iHeartbeat->Start(ETThreeOClock, this);
}

void CSomeObserver::CancelTimer()
{
    iHeartbeat->Cancel();
}

CSomeObserver::~~CSomeObserver()
{
    if (iHeartbeat)
        iHeartbeat->Cancel();
    delete iHeartbeat;
}

```

Еще два класса для работы с таймером я оставляю на ваше самостоятельное изучение. Один из них — класс `CIdle`, позволяющий вызывать функцию обратной связи в момент перехода системы в неактивное состояние. Он удобен для реализации задач, которые должны выполняться в фоновом режиме. Используется класс `CIdle` точно так же, как `CPeriodic`.

Вторым классом, не освещаемым в данной книге, является `CDeltaTimer`. Этот довольно редко используемый на практике класс, как и `CPeriodic`, он позволяет подписаться на уведомление с помощью функции обратной связи через определенный промежуток времени. Разница в том, что класс `CDeltaTimer` реализует очередь таких подписчиков с различными функциями обратной связи. После уведомления подписчик из очереди удаляется.

Работа с файловой системой

В Symbian OS реализована поддержка нескольких типов файловых систем. Все операции с ними ведутся через системный файловый сервер (file server), функциональность которого может расширяться за счет подключаемых модулей с расширением “.fsy”. Файловый сервер скрывает особенности работы с различными файловыми системами, а также реализует механизмы оптимизации операций чтения и записи. Таким образом, разработчик получает возможность осуществлять единообразный доступ к данным, хранящимся как в ROM, так и, например, на картах памяти, отформатированных под файловую систему FAT. Еще одно назначение файлового сервера — обеспечение экранирования данных (см. главу 1).

Файловый сервер довольно часто используется системными и пользовательскими приложениями. Он спроектирован как очень надежный и высокопроизводительный сервис. Взаимодействие с ним осуществляется по правилам клиент-серверной архитектуры, что подразумевает передачу сообщений и данных между процессами. Поэтому, несмотря на то, что выполняемые файловым

сервером операции оптимизированы, неграмотно реализованное межпроцессное взаимодействие может снизить их производительность до неприемлемого уровня. Существует ряд рекомендаций, позволяющих работать с файловой системой и файлами наиболее эффективным способом. Они будут приводиться в рамках описания классов и методов. Я призываю отнестись к этим рекомендациям максимально серьезно и всегда их придерживаться.

Именованние файлов и папок

Полное имя файла в Symbian OS имеет формат: *диск:\путь\имя.расширение*.

Оно не должно превышать 256 символов и не зависит от регистра букв. Для хранения полного имени файла часто используется дескриптор типа `TFileName`, являющийся псевдонимом для `TBuf<KMaxFileName>`. Константа `KMaxFileName` во всех современных версиях ОС равна `0x100`. Не следует забывать, что объект `TFileName` довольно большой для стека и использовать его нужно с осторожностью (см. главу 5, раздел “Дескрипторы-буферы `TBuf` и `TBufC`”). Имена каталогов в пути разделяются обратным слешем, как в Windows-системах (поскольку в макросах `_L()` и `_LIT()` обратный слеш является символом для задания управляющих последовательностей, он должен удваиваться).

```
_LIT(KFile, "c:\\path1\\path2\\filename.ext");
TFileName fn(KFile);
```

Разделы файловых систем монтируются в каталоги, имена которых соответствуют буквам латинского алфавита, и называются памятью (*memory*). Мы же воспользуемся более традиционным термином “диски”. Не следует путать память устройства, под которой понимается встроенный накопитель, и оперативную RAM-память. Обычно в системе присутствуют следующие диски.

- `C:\` — память устройства. Раздел на встроенном накопителе, доступный для чтения и записи.
- `D:\` — виртуальный диск RAM. Энергозависим, используется для хранения временных данных. Присутствует только в устройствах на платформе S60.
- `E:\` — внешняя память, чаще всего находящаяся на карте памяти. (В устройствах платформы UIQ буквой этого диска является “D”, а не “E”.)
- `Z:\` — раздел, находящейся в RAM-памяти устройства. Недоступен для записи и содержит данные системы.

Не следует безоглядно полагаться на вышеприведенные соответствия. Вполне возможно, что в ближайшем будущем появятся устройства с поддержкой нескольких внешних накопителей. В этом случае определять, какой из дисков является встроенной памятью, а какой — внешней, а также их количество, придется с помощью специальных API.

В отношении именования файлов и каталогов, а также записи полного пути к ним существуют следующие ограничения.

- В именах файлов и каталогов запрещены символы “<”, “>”, “|”, “/”, “\”, а также двойной обратный слеш “\\”.
- Двоеточие может встречаться только после буквы диска.
- В именах каталогов не должно быть символов “*” и “?”. Они могут использоваться в качестве имени файла или расширения, но лишь для получения полного имени с помощью класса TParse.
- Использование пробелов допустимо, но имена не должны состоять из них целиком.
- Пробел не должен встречаться и между буквой диска и путем.
- Никакой файл или каталог не может быть назван “.” или “..”.

Для разбора имен и путей файловой системы используется класс TParse, определенный в заголовочном файле `f32file.h`. Для работы с ним к проекту необходимо подключить библиотеку импорта `efsrv.lib`. Класс TParse предоставляет следующие методы.

- `Set()` — позволяет указать дескриптор, содержимое которого будет разбираться по правилам именования файлов и каталогов. В дальнейшем мы будем называть его исходным именем. Исходный путь копируется во внутренний буфер (типа `TFileName`) объекта TParse. Вместе с ним должны передаваться указатели на дескрипторы, содержащие относительное имя и имя по умолчанию. Они используются в случае, если в переданном исходном имени отсутствуют некоторые составляющие части (например, диск или расширение файла). Если разработчик не желает использовать уточняющие имена, он должен передать `NULL`.
- `SetNoWild()` — аналогичен методу `Set()`, с той лишь разницей, что запрещает использование символов маски “*” и “?” в названии файла или его расширении.
- `Drive()` — возвращает дескриптор-указатель `TPtrC` на подстроку, содержащую букву диска и двоеточие.
- `IsRoot()` — позволяет узнать, является ли каталог корневым.
- `Path()` — возвращает дескриптор-указатель на подстроку, содержащую путь. Например, из строки `c:\f1\f2\file.ext` будет выделена подстрока `\f1\f2\`. Результат (если путь не пустой) всегда начинается и заканчивается обратным слешем.
- `DriveAndPath()` — возвращает дескриптор-указатель на подстроку, содержащую и диск, и путь. Эквивалентен объединению результатов выполнения методов `Drive()` и `Path()`.
- `AddDir()` — добавление каталога (единственного) к исходному имени. Название не должно содержать обратный слеш. Если в исходном имени присутствуют имя файла или расширение, каталог будет вставлен перед ними.
- `PopDir()` — удаление последнего каталога из исходного имени. Не может применяться, если в нем остался только корневой каталог.

- `Name()` — возвращает дескриптор-указатель на подстроку, содержащую имя файла без расширения.
- `Ext()` — возвращает дескриптор-указатель на подстроку, содержащую расширение имени файла с предшествующей ему точкой (например, “.exe”).
- Методы `IsKMatchAny()` и `IsKMatchOne()` — позволяют узнать, содержат ли имя файла или расширение символы “*” и “?” соответственно.
- `NameAndExt()` — возвращает дескриптор-указатель на подстроку, содержащую имя файла с расширением. Эквивалентен объединению результатов выполнения методов `Name()` и `Ext()`.
- `FullName()` — возвращает дескриптор, содержащий полное имя в формате: “диск:\путь\имя.расширение”. В случае если в исходном имени какие-либо из компонентов отсутствовали, они берутся из относительного имени или имени по умолчанию.
- Методы `DrivePresent()`, `PathPresent()`, `NamePresent()`, `ExtPresent()`, `NameOrExtPresent()` — позволяют удостовериться в существовании соответствующих компонентов исходного имени.
- `IsNameWild()` — позволяет проверить, содержатся ли в имени файла из полного имени символы маски. `IsExtWild()` — аналог метода `IsNameWild()` для расширения. Метод `IsWild()` — эквивалентен конкатенации аргументов методов `IsNameWild()` || `IsExtWild()`.

Как видите, методов в классе `TParse` довольно много, но все они очень простые. Приведем небольшой пример.

```
_LIT(KFile, "c:\\Path1\\Path2\\FileName.Ext");
TParse p;
TFileName res;
p.Set(KFile, NULL, NULL);
res = p.Drive(); // res == "c:"
res = p.Path(); // res == "\\Path1\\Path2\\"
res = p.Ext(); // res == ".Ext"
```

Единственная сложность заключается в использовании символов маски, а также относительного имени и имени по умолчанию для получения полного имени. Для того чтобы прояснить этот вопрос, я приведу несколько примеров из SDK.

```
_LIT(KFile, "a:file1");
_LIT(KRelated, "c:\\path1\\related.xxx");
TParse p;
p.Set(KFile, &KRelated, NULL);
```

	Диск	Путь	Имя файла	Расширение
Исходный путь	A:		file1	
Относительный путь	C:	\\path1\\	related	.xxx
Путь по умолчанию				
Полный путь	A:	\\path1\\	file1	.xxx

```
TParse p;
_LIT(KFile, "a:file1");
_LIT(KDefault, "c:\\path1\\*.");
p.Set(KFile, NULL, &KDefault);
```

	Диск	Путь	Имя файла	Расширение
Исходный путь	A:		file1	
Относительный путь			*	.*
Путь по умолчанию	C:	\\path1\\		
Полный путь	A:	\\path1\\	file1	.*

Помимо класса `TParse`, существуют аналогичные классы `TParsePtr` и `TParsePtrC`. Разница между ними лишь в том, что `TParsePtr` и `TParsePtrC` не копируют передаваемый в методе `Set()` исходный путь во внутренний буфер, а используют дескриптор-указатель (в случае `TParsePtrC` — неизменяемый) на него.

Сессия файлового сервера

Доступ к файловому серверу осуществляется через устанавливаемую с ним с помощью класса `RFs` сессию. Класс `RFs` объявлен в заголовочном файле `f32file.h`, и работа с ним требует подключения библиотеки импорта `efsrv.lib`. После создания экземпляра класса `RFs`, для установления сессии с сервером используется метод `Connect()`. Сессия завершается с помощью метода `Close()`.

```
RFs fs;
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);
<...> // Работа с файловой системой
CleanupStack::PopAndDestroy(&fs);
```

Рекомендуется минимизировать количество открытых сессий `RFs` (экономия ресурсов) и не устанавливать их слишком часто (экономия процессорного времени). Обычно достаточно во всей программе пользоваться одной сессией, устанавливаемой при ее запуске и разрываемой при завершении работы. Более того, в GUI-приложениях сессия с файловым сервером устанавливается и разрывается автоматически. Получить доступ к ней можно с помощью класса `CCoeEnv`, например, следующим образом.

```
RFs fs;
// Создаем новый объект RFs,
// использующий уже установленную системой сессию
fs.SetHandle(CCoeEnv::Static()->FsSession().Handle());
```

В классах, порожденных от `CCoeAppUi`, указатель на объект `CCoeEnv` содержится в `iCoeEnv`.

```

class CFSAppUi : public CAknViewAppUi
{
    <...>
public:

    RFs& iFs;
    <...>
}

CFSAppUi::CFSAppUi() : iFs(iCoeEnv->FsSession())
{
}

```

Установив сессию с файловым сервером, разработчик может выполнять ряд операций, наиболее важные из которых мы рассмотрим подробно.

Текущий каталог сессии, работа с именами файлов и каталогов

Каждая сессия файлового сервера содержит дескриптор, в котором хранится ее текущий каталог. В случае если при обращении к файлу или каталогу их полное имя не указано, сервер будет искать эти объекты в текущем каталоге. Обратите внимание, что, в отличие от MS DOS, где текущие каталоги есть у каждого диска, сессия файлового сервера имеет лишь один текущий каталог для всех дисков.

Получить и изменить текущий каталог можно с помощью методов `SessionPath()` и `SetSessionPath()`. По умолчанию сразу после установки сессии текущим каталогом является приватный каталог процесса.

```

// Установка новой сессии с файловым сервером
RFs fs;
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);

// Получение текущего каталога
TFileName path;
User::LeaveIfError(fs.SessionPath(path));
// path == "c:\Private\e03e2e69\"

// Изменение текущего каталога
_LIT(KNewSessionPath, "c:\\");
User::LeaveIfError(fs.SetSessionPath(KNewSessionPath));
<...> // Продолжаем работу с сессией
CleanupStack::PopAndDestroy(&fs); // Закрываем сессию

```

Существует ряд ограничений, накладываемых на новое значение текущего каталога сессии при использовании метода `SetSessionPath()`.

- Оно должно оканчиваться обратным слешем, чтобы гарантировать, что это не имя файла.

- Если в нем не указан диск, то будет выполнен поиск этого каталога на всех дисках в стандартном порядке. Первый диск, на котором такая папка будет найдена, и станет диском текущего каталога.
- Компоненты пути должны быть синтаксически верны (не нарушать правил именования файлов и каталогов).
- Если новым значением является папка `\sys\` или приватная папка другого процесса, то от программы потребуются наличие доступа к защищенной возможности AllFiles.

Стандартным порядком обхода дисков при поиске чего-либо до Symbian 9.x являлся алфавитный. Это представляло опасность для системы. На диске C: могли быть размещены исполняемые файлы с теми же именами, что и системные приложения, расположенные на диске Z:, и таким образом осуществлялась подмена компонентов Symbian OS. Начиная с Symbian 9.x поиск сначала проводится на диске Z:, а затем на всех остальных в алфавитном порядке.

Для того чтобы проверить синтаксическую корректность имени файла или папки, можно воспользоваться методом `IsValidName()`. Он возвращает значение `EFalse`, если нарушено одно из правил именования. Его перегруженный вариант также позволяет получить символ, присутствие которого приводит к ошибке. Пример.

```
_LIT(KPath, "c:\\Path1\\filename.*");
if (!fs.IsValidName(KPath))
{
    // Имя синтаксически некорrekтно
}

// или

TText bad_char;
if (!fs.IsValidName(KPath, bad_char))
{
    // bad_char = 0x002A ("*")
}
```

Текущий каталог сессии файлового сервера можно использовать для получения полного имени файла или каталога с помощью класса `TParse`. Для этого класс `RFs` предоставляет методы `Parse()`, позволяющие передавать это значение в объект `TParse` в качестве имени по умолчанию. Пример.

```
_LIT(KFileName, "file.ext");
TParse p;

// Эквивалентно
// p.Set(KFileName, NULL, &fs.SessionPath());
fs.Parse(KFileName, p);

_LIT(KRelatedPath, "c:\\dir\\");
```

```
// Эквивалентно
// p.Set(KFileName, &KRelatedPath, &fs.SessionPath());
fs.Parse(KFileName, KRelatedPath, p);
```

Приватный каталог процесса

Так как текущий каталог можно изменить, не стоит использовать метод `SessionPath()` для доступа к приватному каталогу. Для этого класс `RFs` предоставляет отдельный метод — `PrivatePath()`. Он возвращает дескриптор, содержащий путь к приватному каталогу без буквы диска. Следует помнить, что в эмуляторе приватный каталог процесса автоматически не создается, так как приложения попадают в него, минуя этап инсталляции. Поэтому разработчик должен сам позаботиться о его создании, прежде чем размещать в нем файлы. Для этого можно воспользоваться следующим кодом.

```
// Следующий код выполняется только в сборках WINS и WINSCW
#ifdef __WINS__
    // Получение буквы диска
    TBuf<2> dir;
    RProcess p;
    dir = p.FileName().Left(2);

    // Получение полного пути к приватному каталогу
    TFileName pr_path;
    fs.PrivatePath(pr_path);
    pr_path.Insert(0, dir);

    // Убедимся, что он существует
    fs.MkDirAll(pr_path);
#endif
```

Получение информации о доступных дисках и разделах

Сессия файлового сервера позволяет получить список всех подключенных дисков с помощью метода `DriveList()`. Его единственный аргумент типа `TDriveList` представляет собой дескриптор `TBuf8` длиной `KMaxDrives` (26) символов. В этот массив сервер помещает информацию обо всех смонтированных дисках. Если элемент массива в *i*-й позиции отличен от нуля, то в системе существует диск с таким номером.

Получение буквы диска по его порядковому номеру и обратная операция осуществляются методами `CharToDrive()` и `DriveToChar()`. Сами порядковые номера для ясности принято записывать с помощью значений перечисления `TDriveNumber`, объявленного в том же файле `e32file.h`. Эти значения имеют вид `EDriveA` (равен 0), `EDriveB`, `EDriveC` и т.д. до `EDriveZ` (равен 25).

Следующий пример демонстрирует вывод всех букв обнаруженных дисков в консоль.

```
// Получение данных о подключенных дисках
TDriveList drv_list;
User::LeaveIfError(fs.DriveList(drv_list));

for (TInt drv_num = EDriveA; drv_num <= EDriveZ; drv_num++)
    if (drv_list[drv_num] != 0) // Поиск подключенных дисков
    {
        // Получение буквы диска по номеру
        TChar drv_letter;
        User::LeaveIfError(fs.DriveToChar(drv_num, drv_letter));

        // Вывод в консоль в формате "диск:\"
        _LIT(KDrive, "%c:\\");
        console->Printf(KDrive, TUint(drv_letter));
    }
```

Информацию о диске можно получить с помощью метода `Drive()`. В него передается объект класса `TDriveInfo`, в который помещаются результаты. Вторым аргументом метода является порядковый номер диска (значение `TDriveNumber`). Если он не указан, используется диск текущего каталога сессии. Пример.

```
TDriveInfo drv_inf;
// Получение информации о диске для текущего каталога
fs.Drive(drv_inf);
// Получение информации о диске c:\
fs.Drive(drv_inf, EDriveC);
```

Класс `TDriveInfo` позволяет получить следующие сведения:

- `iBattery` — имеет ли диск собственный аккумулятор и его состояние;
- `iType` — тип устройства (дисковод, жесткий диск, CD-ROM, RAM, Flash, ROM и т.д.);
- `iDriveAtt` — атрибуты диска (`Local`, `Rom`, `Redirected`, `Substed`, `Internal`, `Removable`);
- `iMediaAtt` — атрибуты медиа (`VariableSize`, `DualDensity`, `Formattable`, `WriteProtected`, `Lockable`, `Locked`).

Более подробную информацию предоставляют сведения о разделе, получаемые с помощью метода `Volume()`. Ему передаются объект типа `TVolumeInfo` (для размещения в нем результата) и порядковый номер диска (если не указан, используется диск текущего каталога сессии). Класс `TVolumeInfo` предоставляет следующие данные:

- `iDrive` — объект класса `TDriveInfo` с информацией о диске, поэтому информация, получаемая с помощью метода `Volume()`, включает в себя результаты метода `Drive()`;
- `iUniqueID` — уникальный идентификатор раздела;

- `iSize` — размер раздела в байтах (текущий размер плюс свободное пространство);
- `iFree` — размер свободного места в байтах;
- `iName` — метка (название) раздела, может содержать до 256 символов Unicode.

Метку раздела можно также получить с помощью метода `GetDriveName()`. Метки являются необязательными и чаще всего отсутствуют.

```
TFileName fn;
fs.GetDriveName(EDriveE, fn);
// fn == "MultiMediaCard0"
```

Создание каталогов, переименование и удаление файлов и каталогов

Класс `RFs` предоставляет два метода для создания каталогов: `MkDir()` и `MkDirAll()`. Метод `MkDir()` позволяет создать каталог с заданным именем в уже существующем каталоге. Если такой каталог уже имеется, будет возвращена ошибка `KErrAlreadyExists`.

В отличие от `MkDir()`, метод `MkDirAll()` способен создать не только сам каталог, но и путь к нему. Если часть каталогов в пути уже существует, они будут проигнорированы, но если уже существует конечный каталог заданного пути, метод вернет ошибку `KErrAlreadyExists`.

В обоих методах аргумент должен содержать путь, оканчивающийся обратным слешем (иначе последний элемент будет воспринят как имя файла и проигнорирован). Корневой каталог диска создаваться этими методами не может. Небольшой пример.

```
// Создание каталога dir2 в c:\dir\
_LIT(KPath, "c:\\dir\\dir2\\");
fs.MkDir(KPath); // Если c:\dir\ не существует —
                // произойдет ошибка
fs.MkDirAll(KPath); // Если c:\dir\ не существует,
                  // то будет создан
```

Если диск в пути не указан, он будет получен из текущего каталога сессии.

```
_LIT(KPath2, "c:\\test\\");
// Создаем каталог test на диске c:
fs.MkDir(KPath2);
// Меняем текущий каталог сессии
fs.SetSessionPath(KPath1);
// Создаем каталог test2 на диске c:
fs.MkDir(_L("\\test2\\"));
// Создаем каталог test4 в c:\test\test3\
fs.MkDirAll(_L("\\test\\test3\\test4\\"));
```

Для удаления каталога необходимо воспользоваться методом `Rmdir()`. При этом каталог должен быть пустым, иначе будет возвращена ошибка `KErrInUse`. Аргумент должен содержать путь, оканчивающийся обратным слешем (иначе последний элемент будет воспринят как имя файла и проигнорирован). Символы маски в методе `Rmdir()` не поддерживаются, поэтому удалять можно только один каталог за раз. Если диск в пути не указан, он будет получен из текущего каталога сессии. Корневой каталог диска удаляться не может.

```
_LIT(KPath3, "c:\\test\\test5\\test6\\");
TInt err = fs.Rmdir(KPath3);
```

Файлы удаляются с помощью метода `Delete()`. Этот метод также не поддерживает символы маски, поэтому файлы можно удалять только по одному. Удалять можно только файлы, не имеющие атрибутов “системный” (`system`) и “только для чтения” (`read-only`). Кроме того, файл не должен быть в этот момент открыт для чтения или записи. При использовании неполного пути недостающие компоненты берутся из текущего каталога сессии.

```
TInt err = fs.Delete(_L("c:\\file.ext"));
```

Переименование файлов и каталогов выполняется одним методом — `Rename()`. Он имеет два аргумента: старое и новое имя объекта. Новое имя может находиться в другом родительском каталоге — в этом случае объект будет перенесен в него. При переименовании файлы и каталоги могут менять свое местоположение только в рамках того же диска.

Метод `Rename()` не копирует или перезаписывает данные, а лишь изменяет записи в таблице размещения файловой системы диска. При переносе каталога в другой родительский каталог, все его содержимое переносится вместе с ним. Метод `Rename()` не может создавать каталоги как метод `MkdirAll()`, поэтому родительский каталог должен существовать. В отличие от описанных ранее методов, при переименовании каталога не нужно указывать обратный слеш в ее старом или новом имени.

Метод `Rename()` не может перезаписывать файлы, поэтому при их переименовании файла с новым именем существовать не должно. Кроме того, в данный момент файл не должен быть открыт для чтения или записи. В отличие от метода `Delete()`, ограничений на атрибуты переименовываемого объекта нет.

При использовании неполного пути недостающие компоненты берутся из текущего каталога сессии.

```
fs.Rename(_L("c:\\test"), _L("c:\\my_folder"));
fs.Rename(_L("c:\\file.ext"),
          _L("c:\\my_folder\\fname.dat"));
```

Для перезаписи файлов существует метод `Replace()`. Он не может применяться к каталогам, не поддерживает символы маски и имеет два аргумента — имена перемещаемого и перезаписываемого файлов. Оба файла должны быть на одном диске и в данный момент закрыты. При перезаписи данные о времени

создания, последнего изменения и атрибуты копируются из заменяемого файла в перемещаемый. Поэтому перемещаемый файл не должен иметь атрибута “только для чтения”. Перезаписываемый файл может не существовать, тогда произойдет простое переименование, но вышеназванного требования к атрибутам это не отменяет. При попытке заменить каталог файлом метод вернет ошибку `KErrAccessDenied`. Пример.

```
fs.Replace(_L("c:\\file2.ext"),
          _L("c:\\myfolder\\fname.dat"));
```

Все вышеприведенные методы не способны вызвать сброс, но возвращают числовой код ошибки. Обязательно проверяйте его. Если операция завершилась успешно, метод вернет значение `KErrNone`.

Платформа безопасности накладывает ограничения на доступ к определенным каталогам. Если вы выполняете операции в системных каталогах `\sys\` или `\resource\`, то вашему приложению потребуется доступ к защищенной возможности TCB. А при обращении к приватным каталогам, не принадлежащим процессу, — к `AllFiles`.

Операции с атрибутами каталогов и файлов

Все атрибуты хранятся в одном 32-битовом беззнаковом числе. Для присвоения одного из атрибутов необходимо установить соответствующий ему бит в этом числе равным 1 (или равным 0, для того, чтобы снять атрибут). Если ни один атрибут не установлен, то число соответствует константе `KEntryAttNormal (0x00)`. В Symbian OS используются следующие атрибуты:

- `KEntryAttReadOnly` — только для чтения;
- `KEntryAttHidden` — скрытый;
- `KEntryAttSystem` — системный;
- `KEntryAttVolume` — раздел;
- `KEntryAttDir` — каталог без атрибутов “скрытый” и “системный”;
- `KEntryAttArchive` — архивный файл.

Получение и изменение атрибутов объектов файловой системы производится с помощью методов `Att()` и `SetAtt()` класса `RFs`.

Метод `Att()` имеет два аргумента: имя объекта и ссылка на переменную типа `TUint` для хранения атрибутов. Для того чтобы проверить состояние некоторого атрибута, необходимо получить результат логической операции “И” между числом, в котором он хранится, и соответствующей ему константой. Если этот результат не равен нулю, то атрибут присутствует. Пример.

```
// Проверяем, установлен ли атрибут "архивный"
TUint atts;
User::LeaveIfError(fs.Att(_L("c:\\file.ext"), atts));
TBool arc = atts & KEntryAttArchive; // Логическое "И"
```

Метод `SetAtt()` имеет три аргумента: имя объекта и два числа. Первое должно содержать битовую маску атрибутов, которые необходимо установить. Второе — битовую маску атрибутов, которые необходимо убрать. Пример.

```
// Установка атрибутов "только для чтения" и "системный"
// Снятие атрибута "архивный"
fs.SetAtt(_L("c:\\file.ext"), KEntryAttReadOnly |
          KEntryAttSystem, KEntryAttArchive);
```

При использовании методов `Att()` и `SetAtt()` для файлов они должны быть предварительно закрыты. Если имя объекта не является полным, недостающие компоненты берутся из текущего каталога сессии.

Помимо атрибутов, объекты файловой системы хранят данные о времени их последнего изменения. Для работы с ним используются методы `Modified()` и `SetModified()`. Время последнего изменения представлено объектом `TTime` в формате UTC.

```
_LIT(KFile, "c:\\file.ext");
TTime time;
if (fs.Modified(KFile, time) == KErrNone)
{
    // Увеличим время изменения на 3 дня
    time += TTimeIntervalDays(3);
    fs.SetModified(KFile, time);
}
```

Для более удобной работы с атрибутами в классе `RFs` определены методы `Entry()` и `SetEntry()`. Метод `SetEntry()` похож на `SetAtt()` с добавлением функциональности `SetModified()`: в нем предусмотрен аргумент для передачи нового времени последнего изменения.

```
TTime time;
time.UniversalTime(); // Текущее время UTC
fs.SetEntry(_L("c:\\file.ext"), time,
            KEntryAttReadOnly | KEntryAttSystem, KEntryAttArchive);
```

Метод `Entry()` несколько более полезен, так как возвращает информацию об объекте в виде объекта типа `TEntry`. Класс `TEntry` предоставляет следующие методы и данные:

- `IsDir()` — проверяет, является ли объект каталогом;
- `IsArchive()` — проверяет, содержит ли объект атрибут “архивный”;
- `IsHidden()` — проверяет, содержит ли объект атрибут “скрытый”;
- `IsReadOnly()` — проверяет, содержит ли объект атрибут “только для чтения”;
- `IsSystem()` — проверяет, содержит ли объект атрибут “системный”;
- `IsValidType()` — проверяет, содержит ли объект UID-идентификаторы;
- `IsUidPresent()` — проверяет, содержит ли объект указанный UID-идентификатор;

- `MostDerivedUid()` — возвращает наиболее специфичный UID объекта (если он содержит три идентификатора, то UID3; если два, то UID2; и т.д.);
- `iAtt` — поле, в котором хранится содержащее атрибуты число;
- `iModified` — время последнего изменения в формате UTC;
- `iType` — объект типа `TUidType`, содержащий от 0 до 3-х UID объекта, обращаться к ним можно также с помощью оператора `[]`;
- `iName` — дескриптор с именем объекта (относительно содержащего его каталога);
- `iSize` — размер файла в байтах.

Пример.

```
_LIT(KFile, "c:\\file.ext");
TEntry ent;
fs.Entry(KFile, ent);
TBool arc = ent.IsArchive();
TTime time = ent.iModified;
TUid uid3 = ent[2];
```

При выполнении всех вышеприведенных операций действуют те же условия платформы безопасности, что и при переименовании объектов файловой системы.

Получение списка подкаталогов и файлов в каталоге

Файловая сессия сервера позволяет получить список объектов в указанном каталоге, отсортированный и отфильтрованный в соответствии с их атрибутами. Результат возвращается в виде указателя на класс `CDir`, инкапсулирующий массив `CArrayPakFlat<TEntry>`. Класс `CDir` предоставляет следующие открытые методы и операторы:

- `Count()` — возвращает количество элементов в массиве;
- `Sort()` — сортировка по указанному ключу;
- оператор `[]` — доступ к элементу массива по индексу.

Как видите, объекты класса `CDir` позволяют только считывать данные. Они никогда не создаются в пользовательском приложении.

Для получения списка объектов в каталоге файловой системы класс `RFs` предлагает три перегруженных метода `GetDir()`.

```
// Получение списка всех объектов
// с фильтрацией по атрибутам и сортировкой
TInt GetDir(const TDesC& aName, TUint anEntryAttMask,
            TUint anEntrySortKey, CDir*& anEntryList);

// Получение списка всех объектов и (отдельно) списка
// каталогов с фильтрацией по атрибутам и сортировкой
TInt GetDir(const TDesC& aName, TUint anEntryAttMask,
```

```

TUint anEntrySortKey, CDir*& anEntryList, CDir*& aDirList);

// Получение списка файлов
// с фильтрацией по идентификаторам и сортировкой
TInt GetDir(const TDesC& aName, const TUidType& anEntryUid,
    TUint anEntrySortKey, CDir*& aFileList);

```

Во всех трех случаях аргумент `aName` должен содержать путь к каталогу и (опционально) маску для поиска файлов, например: `"c:\dir\"` или `"c:\dir*.jpg"`.

Аргумент `anEntryAttMask` позволяет задать фильтрацию по атрибутам объекта. Помимо самих атрибутов могут использоваться следующие битовые маски константы:

- `KEntryAttNormal` — все, кроме каталогов и объектов с атрибутами “скрытый” или “системный”;
- `KEntryAttMatchMask` — все объекты;
- `KEntryAttMaskSupported` — все объекты, кроме разделов;
- `KEntryAttMatchExclude` — исключить все объекты, подходящие под смешанные с этой маской атрибуты;
- `KEntryAttMatchExclusive` — включить только те объекты, которые подходят под смешанные с этой маской атрибуты;
- `KEntryAttAllowUid` — все объекты, имеющие идентификаторы UID.

Например:

```

anEntryAttMask = KEntryAttMatchExclude | KEntryAttReadOnly —
все, кроме объектов “только для чтения”;
anEntryAttMask = KEntryAttMatchExclusive | KEntryAttReadOnly —
только объекты, имеющие атрибут “только для чтения”.

```

Аргумент `anEntrySortKey` является ключом сортировки и может принимать комбинацию следующих значений.

- `ESortNone` — сортировка не производится.
- Только одно из следующих значений для задания порядка отображения:
 - `EAscending` — в алфавитном порядке по возрастанию (это порядок по умолчанию);
 - `EDescending` — в алфавитном порядке по убыванию;
 - `EDirDescending` — в алфавитном порядке по убыванию (только для каталогов).
- Только одно из следующих значений для определения положения списка каталогов:
 - `EDirsAnyOrder` — каталоги и файлы смешаны (по умолчанию);
 - `EDirsFirst` — каталоги помещаются в начало списка результата;
 - `EDirsLast` — каталоги помещаются в конец списка результата.
- Только одно из следующих значений для задания сортировки файлов:

- `ESortByName` — по имени (эта сортировка применяется по умолчанию);
- `ESortByExt` — по расширению (файлы без расширения будут первыми);
- `ESortBySize` — по размеру файла;
- `ESortByDate` — по дате последнего изменения (по умолчанию более новые файлы располагаются в конце списка результата);
- `ESortByUid` — по значению идентификаторов UID.

Примеры возможных значений `anEntrySortKey`:

`anEntrySortKey = Edescending | EDirsFirst | ESortByDate` — каталоги помещаются в начало списка, файлы и каталоги сортируются по убыванию и по дате изменения;

`anEntrySortKey = EDirDescending | EDirsLast` — все сортируется по имени, файлы помещаются в начало списка и сортируются по возрастанию, а каталоги в конец списка и сортируются по убыванию;

`anEntrySortKey = EDescending` — сортировка производится по имени, файлы и каталоги смешаны.

Третий перегруженный метод `GetDir()` позволяет фильтровать объекты по набору UID с помощью аргумента `anEntryUid` типа `TUidType`. Конструктор `TUidType` позволяет задать `UID1`, `UID2` и `UID3`, но последние два идентификатора не являются обязательными. Если идентификатор не указан, вместо него используется значение `TUid::Null()` и фильтрация по нему не проводится.

Следующий пример демонстрирует создание объекта `TUidType`, позволяющего отфильтровать все, кроме исполняемых файлов Symbian 9.x.

```
const TUid KExecutableImageUid = {0x1000007A};
const TUid KUidApp = {0x100039CE};
TUidType uids(KExecutableImageUid, KUidApp);
```

Метод `GetDir()` возвращает результаты в виде указателя на объект `CDir` в аргументе `aFileList`. Метод `GetDir()` сначала присваивает `aFileList` значение `NULL` и лишь затем создает новый объект. Поэтому не передавайте в `GetDir()` указатель на уже созданный объект `CDir`, однако удаление объекта `CDir` должно выполняться разработчиком.

Один из перегруженных методов также принимает дополнительный параметр `aDirList` — через него возвращается список каталогов, таким образом создаются два списка результатов: каталоги с файлами, и каталоги отдельно. Это может быть полезно для реализации рекурсивного обхода каталогов.

Ниже приведен пример одного из вариантов обхода всех файлов и каталогов на одном из дисков.

```
void ListDirL(RFs& aFs, const TDesC& aDir,
const TUint aAttr, const TUint aSort)
{
    // Простейший обход всех объектов файловой системы
    CDir* result(NULL);
    User::LeaveIfError(aFs.GetDir(aDir, aAttr,
```

```

        aSort, result));

    if (result)
    {
        CleanupStack::PushL(result);
        for (int i = 0; i < result->Count(); i++)
        {
            if ((*result)[i].IsDir())
                ListDirL(aFs, (*result)[i].iName, aAttr, aSort);
        }
        CleanupStack::PopAndDestroy(result);
    }

    // Использование функции
    ListDirL(fs, _L("c:\\"), KEntryAttMatchMask, EDirsLast);

```

Прочие полезные функции файловой сессии

Мы рассмотрели только основные функции файловой сессии, но класс RFs предоставляет еще довольно много методов. Они используются не так часто, и рассматривать их подробно не имеет смысла. Ниже приводится краткое описание наиболее полезных методов класса RFs.

- `IsFileInRom()` — проверка, находится ли файл в ROM-памяти.
- `IsFileOpen()` — проверяет, открыт ли файл для чтения или записи.
- `ReadFileSection()` — считывает содержимое файла в 8-битовый дескриптор. Метод примечателен тем, что позволяет прочесть файл, не открывая его, и может применяться даже к уже открытым файлам.
- `IsValidDrive()` — попадает ли индекс диска в допустимый диапазон (от 0 до `KMaxDrives-1`).
- `CreatePrivatePath()` — создает приватный каталог процесса на указанном диске; обычно, на устройстве этот каталог создается автоматически системным установщиком SIS-файлов, но в эмуляторе это не так.
- `SetSessionToPrivate()` — устанавливает текущий каталог сессии равным приватному каталогу процесса на заданном диске.
- `NotifyChange()` — асинхронный запрос, позволяющий получить уведомление о проводящихся в файловой системе или каком-то определенном ее каталоге операциях (например, создание или изменение файлов и каталогов).
- `NotifyDiskSpace()` — асинхронный запрос, позволяющий получить уведомление в случае, если свободного места на диске останется меньше заданного значения.
- `NotifyDismount()` — асинхронный запрос, позволяющий получить уведомление об отключении смонтированной файловой системы.

- `Subst()` — аналог команды `subst` в Windows. Позволяет симитировать собственный диск для каталога.
- `ScanDrive()` — запуск сканера ошибок файловой системы FAT внешнего устройства.

Для работы со многими из этих методов требуется доступ к наиболее конфиденциальным защищенным возможностям системы.

Файловый менеджер CFileMan

Как вы уже заметили, все операции класса `RFs` не поддерживают символы маски и выполняются только для одного объекта. Отсутствует возможность перезаписи, рекурсии, удаления непустых каталогов, создания копий объектов. Наконец, нет асинхронных вариантов функций. Таким образом, класс `RFs` реализует лишь базовые операции, необходимые, но не достаточные для удобной работы. Конечно, разработчик может самостоятельно написать функции, обеспечивающие все вышеописанное на основе методов класса `RFs`, но это привело бы к дублированию кода в исполняемых файлах и снижению его надежности, поэтому все недостающие операции были реализованы на основе методов класса `RFs` в системном классе `CFileMan`, предоставляемом Symbian API.

Класс файлового менеджера `CFileMan` объявлен в том же заголовочном файле и требует подключения той же библиотеки импорта, что и класс `RFs`. Объект `CFileMan` создается с помощью метода `NewL()`, в который по ссылке передается установленная файловая сессия `RFs`.

```
RFs fs;
User::LeaveIfError(fs.Connect());
CFileMan* fm = CFileMan::NewL(fs);
```

После создания с помощью файлового менеджера можно менять атрибуты, копировать, перемещать, переименовывать и удалять объекты.

Операции с атрибутами папок и файлов

Для смены атрибутов используется такой метод.

```
Attribs(const TDesC &aName, TInt aSetMask, TInt aClearMask,
        const TTime &aTime, TInt aSwitch=0)
```

- `aName` — дескриптор, содержащий путь и маску файлов, атрибуты которых необходимо изменить, например: `"c:*.txt"`. Если некоторые компоненты пути не указаны, они будут получены из текущей сессии каталога.
- `aSetMask` и `aClearMask` — битовые маски атрибутов, которые нужно установить и снять.
- `aTime` — время последнего изменения файла. Если разработчик не желает менять этот параметр у всех файлов, то необходимо передать нулевое время.

- `aSwitch` — необязательный аргумент, сигнализирующий о необходимости (если равен 1) выполнения операции рекурсивно во всех подкаталогах. По умолчанию операция не рекурсивна.

Небольшой пример,

```
// Снимаем атрибут "скрытый" у всех ".txt"-файлов диска c:
fm->Attribs(_L("c:\\*.txt"), 0, KEntryAttHidden, TTime(0), 1);
```

Копирование папок и файлов

Копирование файлов выполняется с помощью следующего метода.

```
Copy(const TDesC &anOld, const TDesC &aNew,
      TInt aSwitch=EOverWrite)
```

Операция позволяет копировать файлы между дисками, а также копировать файлы с атрибутами “скрытый”, “системный” и “только для чтения” (атрибуты также копируются). Допускается копирование файлов, открытых в режиме `EFileShareReadersOnly`.

- `anOld` — дескриптор, содержащий путь к каталогу, файлы которого должны быть скопированы. Обратный слеш на конце пути значения не имеет.
- `aNew` — дескриптор, содержащий путь к каталогу, в который должны быть скопированы файлы.
- `aSwitch` — флаг, позволяющий указать режим перезаписи и рекурсии. По умолчанию файлы с совпадающими именами в новом каталоге перезаписываются, а операция выполняется не рекурсивно. Для рекурсивного выполнения флаг должен включать в себя значение `CFileMan::ERecurse`. В этом случае файловый менеджер сам создаст все необходимые вложенные каталоги.

Если некоторые компоненты пути аргументов `anOld` и `aNew` не указаны, они будут получены из текущей сессии каталога. Метод `Copy()` имеет перегруженный вариант, позволяющий скопировать один файл, передав открытый для него объект класса `RFile`.

```
// Копирование всех файлов с диска c:
// в соответствующие папки диска e:
// При совпадении имен старые файлы будут перезаписаны
fm->Copy(_L("c:\\"), _L("e:\\"),
        CFileMan::EOverWrite | CFileMan::ERecurse);
```

Переименование каталогов и файлов

Переименование файлов и каталогов выполняется с помощью следующего метода.

```
Rename(const TDesC &anOld, const TDesC &aNew,
        TInt aSwitch=EOverWrite)
```

Операция не может выполняться рекурсивно и позволяет переносить файлы лишь в пределах диска.

- `anOld` — дескриптор, содержащий путь к переименовываемому объекту или объектам. Если он оканчивается обратным слешем, то слеш трактуется как `"*.*"`. Переименовываемые файлы должны быть закрыты. Символы маски не могут быть использованы для переименования каталогов. Если некоторые компоненты пути не указаны, они будут получены из текущей сессии каталога.
- `aNew` — дескриптор, содержащий новое имя объекта или объектов.
- `aSwitch` — флаг перезаписи. Он равен `CFileMan::EOverWrite`, если перезапись файлов разрешена (по умолчанию), либо нулю, если недопустима. Каталоги перезаписываться не могут.

Примеры.

```
// Следующие четыре операции эквивалентны
// И сводятся к переносу всех файлов
// из папки c:\src\ в c:\trg\
fm->Rename(_L("c:\\src\\"), _L("c:\\trg\\"));
fm->Rename(_L("c:\\src\\*.*"), _L("c:\\trg\\"));
fm->Rename(_L("c:\\src\\"), _L("c:\\trg\\*.*"));
fm->Rename(_L("c:\\src\\*.*"), _L("c:\\trg\\*.*"));

// Переименование c:\src\ в c:\trg\src\
fm->Rename(_L("c:\\src"), _L("c:\\trg\\"));

// Переименование c:\src\ в c:\trg\
fm->Rename(_L("c:\\src"), _L("c:\\trg"));

// Изменение расширения txt-файлов в корневой папке диска c:
fm->Rename(_L("c:\\*.txt"), _L("c:\\*.jpg"));
```

Перенос каталогов и файлов

Для переноса каталогов и файлов предназначен такой метод.

```
Move(const TDesC &anOld, const TDesC &aNew,
      TInt aSwitch=EOverWrite)
```

Он обеспечивает перенос файлов или каталога, заданных аргументом `anOld`, в каталог, заданный аргументом `aNew`. Если некоторые компоненты пути аргументов `anOld` и `aNew` не указаны, они будут получены из текущей сессии каталога. Операция может быть рекурсивной. Открытые файлы не могут быть перенесены. Пример.

```
// Перенос всех файлов из c:\dir\ в c:\test\
// При совпадении имен происходит перезапись
fm->Move(_L("c:\\dir\\"), _L("c:\\test\\"));
// Перенос папки c:\dir\ в c:\test\
```

```
// При совпадении имен возникает ошибка
fm->Move(_L("c:\\dir"), _L("c:\\test\\"), 0);
// Перенос всех файлов с расширением txt на диск e:
fm->Move(_L("c:\\*.txt"), _L("e:\\"),
    CFileMan::EOverWrite | CFileMan::ERecurse);
```

Удаление объектов

Файлы удаляются с помощью следующего метода.

```
Delete(const TDesC &aName, TUint aSwitch=0)
```

Он принимает в качестве аргументов путь к файлу (или маску файлов) и переменную, определяющую наличие рекурсивности. Не могут быть удалены открытые файлы и файлы, имеющие атрибут “только для чтения”. Если дескриптор aName содержит пустую строку, то вместо нее подразумевается “*.*”. Если некоторые компоненты пути не указаны, они будут получены из текущей сессии каталога. Пример.

```
// Удаление всех файлов с расширением txt в корневой папке
fm->Delete(_L("c:\\*.txt"));
// Удаление всех файлов с расширением txt на диске c:
fm->Delete(_L("c:\\*.txt"), CFileMan::ERecurse);
// Удаление одного файла
fm->Delete(_L("c:\\file.ext"));
```

Удаление каталогов выполняется с помощью метода RmDir(), имеющего один аргумент: путь к удаляемому каталогу. Эта операция всегда выполняется рекурсивно. Все файлы в удаляемом каталоге и вложенных подкаталогах должны быть закрыты и не иметь атрибута “только для чтения”, иначе метод вернет ошибку с кодом KErrInUse. Все содержимое удаляемого каталога будет стерто вместе с ним. Если некоторые компоненты пути не указаны, они будут получены из текущей сессии каталога. Метод имеет асинхронный вариант. Пример.

```
//Удаление каталога c:\\dir\
fm->RmDir(_L("c:\\dir"));
```

Вышеприведенные методы Attribs(), Copy(), Rename(), Move(), Delete() и RmDir() имеют перегруженные асинхронные аналоги. Для выполнения асинхронных операций объект CFileMan автоматически создает новый поток в процессе.

Слушатель файлового менеджера

При создании либо позднее с помощью метода SetObserver() в объект CFileMan может быть передан указатель на класс, реализующий интерфейс слушателя. Данный интерфейс объявлен в виде класса примеси MFileManObserver, содержащего следующие чистые виртуальные методы:

- NotifyFileManStarted() — вызывается, когда файловый менеджер начинает выполнение операции над объектом;

- `NotifyFileManOperation()` — вызывается в процессе выполнения операции, сигнализируя о завершении ее этапа;
- `NotifyFileManEnded()` — вызывается, когда файловый менеджер завершает выполнение операции над объектом.

Все эти методы должны возвращать одно из значений перечисления `MFileManObserver::TControl`. С их помощью слушатель управляет ходом выполнения операции. Результатом выполнения методов интерфейса могут быть:

- `EContinue` — продолжение операции, переход к следующему объекту;
- `ERetry` — повтор операции для текущего объекта;
- `ECancel` — отмена операции для текущего объекта, переход к следующему объекту;
- `EAbort` — отмена операции для всех объектов, метод `CFileMan`, с помощью которого была вызвана операция, вернет код ошибки `KErrCanceled`.

С помощью слушателя `MFileManObserver` можно обрабатывать ошибки, возникшие для отдельных объектов, а также отображать информацию о ходе выполнения операции. Для получения этих данных класс `CFileMan` предоставляет следующие методы.

- `CurrentAction()` — позволяет получить информацию о типе операции, выполняемой в данный момент файловым менеджером. Результат возвращается в виде значения перечисления `CFileMan::TOperation`:
 - `ENone` — никаких операций не выполняется;
 - `EAttribs` — изменение атрибутов объектов;
 - `ECopy` — копирование объектов;
 - `EDelete` — удаление файлов;
 - `EMove` — перемещение объектов;
 - `ERename` — переименование объектов;
 - `ERmdir` — удаление папки со всем содержимым;
 - `ERenameInvalidEntry` — изменение названия объекта на короткое имя VFAT (гарантировано уникальное);
 - `ECopyFromHandle` — копирование одного файла по открытому хендлу (использование перегруженного метода `Copy()`, принимающего ссылку на `RFile`).
- `GetCurrentSource()` — помещает в переданный дескриптор исходное имя объекта, над которым проводится операция (копирование, перемещение или удаление).
- `GetCurrentTarget()` — помещает в переданный дескриптор целевое имя объекта, над которым проводится операция (копирование, перемещение или переименование).
- Методы `FullPath()` и `AbbreviatedPath()` позволяют получить дескриптор-указатель на полное и относительное имя текущего обрабатываемого объекта соответственно.

- `BytesTransferredByCopyStep()` — позволяет получить количество переданных байтов объекта при копировании или перемещении. Большие файлы копируются или переносятся по частям. После обработки каждой из частей вызывается метод `NotifyFileManOperation()` слушателя. В нем можно воспользоваться методом `BytesTransferredByCopyStep()` и узнать, сколько байт файла уже обработано. Данный метод очень удобен при использовании шкалы прогресса (progress bar), визуализирующие ход выполнения операции.
- `CurrentEntry()` — возвращает объект класса `TEntry`, содержащий информацию о текущем обрабатываемом объекте. Класс `TEntry` уже рассматривался нами ранее.
- `GetLastError()` — возвращает код ошибки (или `KErrNone`, если ошибки не произошло) при выполнении операции для текущего объекта. Метод предназначен для использования в методе `NotifyFileManEnded()` слушателя.
- `GetMoreInfoAboutError()` — позволяет получить расширенную информацию о произошедшей ошибке. Возвращает одно из значений перечисления `TFileManError`:
 - `ENoExtraInformation` — дополнительная информация об ошибке отсутствует;
 - `EInitializationFailed` — на этапе подготовки к выполнению операции (начальное сканирование каталога) произошел сброс;
 - `EScanNextDirectoryFailed` — во время сканирования следующего по порядку каталога произошел сброс, выполнение операции так и не началось;
 - `ESrcOpenFailed` — ошибка при попытке открыть исходный файл для копирования или перемещения;
 - `ETrgOpenFailed` — ошибка при попытке создания или перезаписи целевого файла во время копирования или перемещения;
 - `ENoFilesProcessed` — операция завершилась, но ни один файл не был обработан, так как не нашлось подпадающих под критерии.

Следующий пример (листинги 6.20 и 6.21) демонстрирует простейший класс-слушатель, инкапсулирующий файловый менеджер. Он поддерживает только операцию копирования, но его легко усовершенствовать.

Листинг 6.20. Файл `FileManObserver.h`

```
/*
// Объявление класса слушателя,
// унаследованного от примеси MFileManObserver
*/
#include <f32file.h>

class CFileManObserver : public CBase, MFileManObserver
{

```

```

private:
    // Деструктор
    ~CFileManObserver();
    // Методы для создания экземпляра класса
    static CFileManObserver* NewL();
    static CFileManObserver* NewLC();

public:
    // Запуск операции копирования
    TInt Copy(const TDesC& anOld, const TDesC& aNew,
    TInt aSwitch = CFileMan::EOverWrite);
public:
    // Конструкторы
    CFileManObserver();
    void ConstructL();

private:
    // Реализация чистых виртуальных методов из
    // примеси MFileManObserver
    MFileManObserver::TControl NotifyFileManStarted();
    MFileManObserver::TControl NotifyFileManOperation();
    MFileManObserver::TControl NotifyFileManEnded();

private:
    // Сессия файлового сервера
    RFs iFs;
    // Файловый менеджер
    CFileMan* iFileMan;
};

```

Листинг 6.21. Файл FileManObserver.cpp

```

// Реализация методов класса CFileManObserver

// Стандартные методы NewX()
CFileManObserver* CFileManObserver::NewLC()
{
    CFileManObserver* self = new (ELeave) CFileManObserver ();
    CleanupStack::PushL(self);
    self->ConstructL();
    return self;
}

CFileManObserver* CFileManObserver::NewL()
{
    CFileManObserver* self = CFileManObserver::NewLC();
    CleanupStack::Pop(); // self;
    return self;
}

```

```

// Конструкторы

CFileManObserver::CFileManObserver()
{
}

void CFileManObserver::ConstructL()
{
    // Установление сессии с файловым сервером
    User::LeaveIfError(iFs.Connect());
    // Создание файлового менеджера. Передача указателя
    // на объект, реализующий интерфейс слушателя
    iFileMan = CFileMan::NewL(iFs, this);
}

// Деструктор

CFileManObserver::~CFileManObserver()
{
    // Удаление файлового менеджера
    delete iFileMan;
    // Закрытие сессии с файловым сервером
    iFs.Close();
}

// Метод, инкапсулирующий операцию копирования CFileMan

TInt CFileManObserver::Copy(const TDesC& anOld,
                           const TDesC& aNew, TUint aSwitch)
{
    return iFileMan->Copy(anOld, aNew, aSwitch);
}

// Методы интерфейса слушателя

MFileManObserver::TControl
    CFileManObserver::NotifyFileManStarted()
{
    // Начало операции
    return MFileManObserver::EContinue;
}

MFileManObserver::TControl
    CFileManObserver::NotifyFileManOperation()
{
    // Очередной этап операции
    switch (iFileMan->CurrentAction())
    {

```

```

        case CFileMan::ECopy:
        {
            if (iFileMan->GetLastError() != KErrNone)
            {
                // Обработка ошибок,
                // информирование пользователя,
                // попытка исправить ситуацию
                // (закрыть файл, поменять их атрибуты)
                return MFileManObserver::ERetry;
            }

            return MFileManObserver::EContinue;
        }
        break;

    default:
        return MFileManObserver::EContinue;
    }

}

MFileManObserver::TControl
    CFileManObserver::NotifyFileManEnded()
{ // Завершение операции
    return MFileManObserver::EContinue;
}

```

Пример использования класса CFileManObserver.

```

CFileManObserver* obs = CFileManObserver::NewLC();
_LIT(KSrc, "c:\\test\\*.");
_LIT(KDest, "e:\\test\\*.");
TInt res = obs->Copy(KSrc, KDest,
    CFileMan::EOverWrite | CFileMan::ERecurse);
CleanupStack::PopAndDestroy(obs);

```

Платформа безопасности накладывает ограничения на доступ к определенным каталогам. Если вы выполняете операции в системных каталогах `\sys\` или `\resource\`, то вашему приложению потребуется доступ к защищенной возможности TCB. А при обращении к приватным каталогам, не принадлежащим процессу, — к AllFiles.

Подготовка к сертификации ASD

- Понимание роли файлового сервера в системе.
- Знание базовой функциональности, предоставляемой классом RFs.
- Умение отличить код, корректно устанавливающий сессию с файловым сервером (RFs).

- Понимание того, как класс `TParse` может быть использован для работы с именами файлов.
 - Знание того, что приватный каталог процесса может быть получен с помощью метода `RFs::PrivatePath()`.
-

Файлы, чтение и запись данных

Мы уже упоминали метод `RFs::ReadFileSection()`, используемый для чтения данных из файлов без их открытия. Иногда он может быть полезен, но файловый сервер не дает гарантий, что полученные данные актуальны, так как в тот же самый момент, когда выполняется чтение, в файл может идти запись.

Основным классом для доступа к содержимому файлов является `RFile`, объявленный в `f32file.h` и реализованный в `efsrv.lib`. Он является субсессией сессии `RFs`. Все операции с хранящимися в файле данными выполняет файловый сервер. Сервер также обеспечивает разделяемый доступ к файлу (если он возможен). Для работы `RFile` необходим подключенный к серверу объект `RFs`, который он использует для отправки команд и обмена данными с файловым сервером. После открытия или создания файла, объект `RFile` получает и хранит его хендл. В случае если сессия файлового сервера будет разорвана раньше, чем `RFile` закроет файл, все субсессии будут также разорваны, а файлы закрыты. Тогда при попытке считать или записать данные с помощью `RFile` произойдет ошибка.

Открытие файла

Для того чтобы открыть файл и начать работу с ним, необходимо воспользоваться одним из трех предоставляемых классом `RFile` методов:

- `Open()` — открытие существующего файла;
- `Create()` — создание и открытие нового файла и установка атрибута “архивный”;
- `Replace()` — создание и открытие файла; если файл не существовал, то он будет предварительно создан; если же файл существовал, то после открытия его размер изменяется на нулевой (все содержимое удаляется), в любом случае файлу устанавливается атрибут “архивный”.

В качестве аргументов в данные методы передаются установленная сессия файлового сервера, имя файла и режим доступа к нему. В случае если в имени файла имеются недостающие компоненты, они будут получены из текущего каталога объекта `RFs`. Режим доступа к файлу представляет собой комбинацию (полученную с помощью логического “ИЛИ”) значений перечисления `TFileMode`, объявленного в том же заголовочном файле, что и класс `RFile`. Их мы подробно рассмотрим несколько позднее.

Следующий пример демонстрирует открытие файла для чтения данных. В случае если файл не существовал, он создается. Заметьте, что данный код не эквивалентен методу `Replace()`, так как позволяет сохранить данные в уже существующем файле.

```
RFs fs;
// Подключение к файловому серверу
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);

RFile file;
_LIT(KFileName, "c:\\file.ext");
// Попытаемся создать и открыть для чтения новый файл
TInt res = file.Create(fs, KFileName, EFileRead);

// Если файл уже существует, то будет возвращена
// ошибка KErrAlreadyExists
if (KErrAlreadyExists == res)
{
    // Попытаемся открыть существующий файл
    User::LeaveIfError(file.Open(fs, KFileName, EFileRead));
}
else
    // Иначе проверим, не возникло ли при создании
    // другой ошибки
    User::LeaveIfError(res);

// Файл открыт
CleanupClosePushL(file);
<...> // Работа с файлом
// Закрываем файл и сессию
CleanupStack::PopAndDestroy(2, &fs);
```

Вышеприведенные методы, как следует из их названия, не способны привести к сбросу. Вместо этого возможные ошибки возвращаются в качестве результата и относиться к ним нужно очень внимательно.

Класс `RFile` предоставляет еще один метод, с помощью которого можно начать работу с файлом: `Temp()`. Он похож на метод `Create()`, с тем отличием, что разработчику не нужно указывать имя файла, а лишь папку, в которой он должен быть создан. Выбор имени нового файла осуществляется файловым сервером таким образом, чтобы оно было уникальным в каталоге. Кроме того, в методе `Temp()` передается ссылка на дескриптор `TFileName`, позволяющий получить имя созданного файла. Метод `Temp()` часто применяется для создания временных файлов. Небольшой пример.

```
RFile file;
_LIT(KFilePath, "c:\\");
// Создаем временный файл
```

```

TFileName fname;
// Создание временного файла,
// в fname помещается его имя (например, "c:\TMP04C73.$$$")
User::LeaveIfError(file.Temp(fs, KFilePath, fname,
                             EFileRead));

CleanupClosePushL(file);
<...> // Работа с файлом
CleanupStack::PopAndDestroy(&file);

// Удаление временного файла
User::LeaveIfError(fs.Delete(fname));

```

Режимы доступа к файлу

Последним аргументом в методах `Open()`, `Create()`, `Replace()` и `Temp()` является беззнаковое число, задающее режим доступа к файлу. Этот параметр позволяет определить, могут ли другие объекты `RFile` открывать тот же файл, и на каких условиях. Он является комбинацией значений перечисления `TFileMode`, полученной с помощью логического “ИЛИ”.

Например, если объект `RFile` №1 первым открывает файл и указывает режим совместного доступа для чтения/записи, то после него тот же файл сможет открыть объект `RFile` №2. Таким образом, с файлом будут работать одновременно два экземпляра `RFile`, возможно, принадлежащие разным процессам. Более того, объект `RFile` №2 может дополнительно ограничить режим совместного доступа для прочих клиентов. Как только объект `RFile` №2 закроет свой хендл на файл, режим совместного доступа снова облегчится до условий, выдвинутых объектом `RFile` №1.

Разобраться в хитросплетениях режимов и ограничений, а также порядке их применения довольно сложно. Поэтому мы выделим среди значений `TFileMode` две группы: задающие режим доступа (*access mode*) и задающие режим совместного доступа (*share mode*). Эти группы содержат не все значения перечисления `TFileMode`, но оставшиеся за рамками рассмотрения значения используются крайне редко и служат для тонкой настройки операций ввода-вывода (например, позволяют отключить буферизацию при записи данных в файл). Кроме того, часть из них поддерживается, начиная лишь с определенной версии файлового сервера.

Итак, мы рассматриваем две группы значений `TFileMode`, комбинация которых управляет доступом к файлу. Первая из них задает режима доступа к файлу данного объекта `RFile`. В нее входят всего два значения:

- `EFileRead` — доступ только для чтения (по умолчанию), запись данных в открытый с таким флагом файл невозможна;
- `EFileWrite` — доступ как для чтения, так и для записи данных.

Вторая группа значений задает режим совместного доступа к файлу.

- `EFileShareExclusive` — исключительный доступ (по умолчанию).

Ни один другой клиент не сможет открыть этот файл, пока он не будет

закрыт. Если же файл уже открыт кем-то, то файловый сервер откажет в доступе и метод вернет ошибку.

- `EFileShareReadersOnly` — файл может быть открыт другими объектами `RFile`, но только для чтения. Этот режим совместного доступа не может комбинироваться с `EFileWrite`.
- `EFileShareAny` — файл может быть открыт другими объектами `RFile` как для чтения, так и для записи. При этом при последующих открытиях запрещается ограничивать режим совместного доступа до `EFileShareReadersOnly`.
- `EFileShareReadersOrWriters` — файл может быть открыт другими объектами `RFile` как для чтения, так и для записи. При этом режим совместного доступа может быть ограничен.

В итоговую комбинацию режимов должно входить по одному значению из каждой группы. Если значение какой-либо группы не указано, используется соответствующий режим по умолчанию.

В табл. 6.1 приведены варианты возможных режимов для двух клиентов, по очереди открывающих один файл, и результат их наложения. Для краткости “`EFileShare`” исключено из названий. Оригинал таблицы можно найти в справочнике SDK.

Таблица 6.1. Режимы совместного доступа для двух клиентов

Клиент №1	Клиент №2	Результирующий режим
...ReadersOnly	...ReadersOnly	...ReadersOnly
	...ReadersOrWriters EFileRead	...ReadersOnly
	...ReadersOrWriters EFileWrite	-
	...Any	-
...ReadersOrWriters EFileRead	...ReadersOnly	...ReadersOnly
	...ReadersOrWriters EFileRead	...ReadersOrWriters
	...ReadersOrWriters EFileWrite	...ReadersOrWriters
	...Any	...Any
...ReadersOrWriters EFileWrite	...ReadersOnly	-
	...ReadersOrWriters EFileRead	...ReadersOrWriters
	...ReadersOrWriters EFileWrite	...ReadersOrWriters
	...Any	...Any
...Any	...ReadersOnly	-
	...ReadersOrWriters EFileRead	...Any

Окончание табл. 6.1

Клиент №1	Клиент №2	Результирующий режим
	...ReadersOrWriters EFileWrite	...Any
	...Any	...Any

Общие рекомендации по выбору подходящего режима доступа таковы.

- Для исключительного доступа к файлу необходимо использовать режим EFileRead либо EFileWrite (режим совместного доступа EFileShareExclusive не указывается, так как он используется по умолчанию).
- Для совместного доступа к файлу, в который одновременно будет вестись запись, необходимо использовать EFileShareAny | EFileWrite, если запись осуществляет данный клиент, либо EFileShareReadersOrWriters | EFileRead, если данный клиент лишь считывает то, что записывают другие.
- Для совместного доступа к файлу, не предназначенному для записи, необходимо использовать режим совместного доступа EFileShareReadersOnly (он может использоваться только в комбинации с EFileRead, но так как это значение по умолчанию, то его можно не писать).
- Режимы EFileShareReadersOrWriters | EFileWrite и EFileShareAny | EFileRead использовать не рекомендуется из-за возможных конфликтов с режимом EFileShareReadersOnly.

В уже открытом файле режим доступа EFileShareExclusive может быть изменен на EFileShareReadersOnly (и обратно) с помощью метода ChangeMode (). Другие режимы доступа метод ChangeMode () не поддерживает.

Чтение и запись данных

Класс RFile позволяет считывать данные в 8-битовый дескриптор и записывать их из него. При этом могут указываться позиция, начиная с которой производится чтение или запись, а также размер данных. Естественно, размер считываемых или записываемых данных не должен превышать максимального размера дескриптора.

Если позиция не указана, то используется так называемая “текущая позиция”. Она всегда равна адресу, на котором остановилась операция чтения или записи. В начале работы текущая позиция указывает на начало файла. Таким образом, если после открытия файла дважды выполнить чтение блоков данных по 10 байт, не указывая позицию, с которой производится чтение, то считаны будут первые 20 байт файла.

Если не указан размер данных, которые должны быть считаны из файла или записаны из дескриптора в файл, то за размер принимается максимальная длина дескриптора.

Чтение из файла выполняется с помощью одного из четырех перегруженных вариантов метода `Read()`. Пример.

```
_LIT(KFileName, "c:\\file.ext");
User::LeaveIfError(file.Open(fs, KFileName,
    EFileRead|EFileShareReadersOrWriters));
CleanupClosePushL(file);

TBuf8<15> buf;
// Считываем 15 байт из файла
file.Read(buf);

// Считываем следующие 10 байт
file.Read(buf, 10);

// Считываем 10 байт: с 5-го по 15-й
file.Read(5, buf, 10);

// Считываем 15 байт: с 5-го по 20-й
file.Read(5, buf);

CleanupStack::PopAndDestroy(&file);
```

Перед чтением содержимое дескриптора-буфера очищается автоматически, а его длина устанавливается равной нулю. Результатом метода `Read()` является код ошибки. Например, в случае, если максимальная длина дескриптора меньше чем размер данных, которые в него должны быть помещены, возвращается ошибка `KErrOverflow`. Класс `RFile` также предоставляет четыре асинхронных аналога вышеприведенных методов. В этом случае результат помещается в объект `TRequestStatus`. Отменить асинхронный запрос метода `Read()` можно с помощью метода `ReadCancel()`.

Следует заметить, что при попытке прочитать данные за пределами файла (текущая позиция плюс размер считываемых данных превышает размер файла) ошибки не произойдет.

Класс `RFile` предоставляет метод `Size()` для получения текущего размера файла и метод `SetSize()` для его изменения. Учитывая это и то, что в качестве буфера для хранения данных можно использовать любой тип изменяемого дескриптора, разработчик может считать файл произвольного размера целиком.

```
HBufC8* bufc(NULL);
TInt len(0);
TInt err = file.Size(len); // получение размера файла
if (KErrNone == err && len > 0)
{
    // Создаем дескриптор нужного размера
```

```

    bufc = HBufC8::NewL(len);
    TPtr8 buf(bufc->Des());
    // Чтение всего файла
    file.Read(0, buf);
}
// Доступ к полученным данным осуществляется через *bufc

```

Методы для записи данных в файл называются `Write()` и аналогичны методам чтения. Они также принимают в качестве аргумента 8-битовый дескриптор, содержащий данные для записи. Если указан определенный размер блока записываемых данных, то он не должен превышать длины строки в дескрипторе. Содержимое дескриптора после записи не очищается и может использоваться повторно. Хранящиеся в файле данные перезаписываются, а его размер увеличивается автоматически. Примеры.

```

RFile file;
_LIT(KFileName, "c:\\file.ext");
User::LeaveIfError(file.Open(fs, KFileName,
    EFileWrite|EFileShareAny));
CleanupClosePushL(file);

_LIT8(KWriteData, "12345abcde");
TBuf8<15> buf(KWriteData);

// Очищаем файл
TInt res = file.SetSize(0);

// Записываем 10 байт в файл
res = file.Write(buf);
// В файле "12345abcde"

// Записываем еще 5 байт из buf
file.Write(buf, 5);
// В файле "12345abcde12345"

// Переписываем 3 байта: с 6-го по 8-й
file.Write(5, buf, 3);
// В файле "12345123de12345"

// В качестве источника данных
// можно использовать и константу
file.Write(KWriteData);
// Записано 10 байт после 8-го
// В файле "1234512312345abcde"

// Или подстроку от исходной
file.Write(0, buf.Right(5));
// В файле "abcde12312345abcde"

CleanupStack::PopAndDestroy(&file);

```

Как видно из примера, в методах `Write()` нет возможности указать, с какого символа дескриптора следует вести запись. В этом случае вместо дескриптора можно передать дескриптор-указатель на его подстроку. Для методов `Write()` также существуют асинхронные варианты, отменить асинхронный запрос которых можно с помощью метода `WriteCancel()`.

Если разработчику необходимо поместить в файл или считать числовые данные или структуру, то он может воспользоваться дескрипторами `TPckg` и `TPckgBuf`. Пример.

```
// Запись числа в файл
TInt i = 100;
TPckg<TInt> pkg_int(i);
file.Write(pkg_int);

// Запись структуры в файл
TPoint point(0, 1);
TPckg<TPoint> pkg_point(point);
file.Write(pkg_point);

// Чтение числа из файла
TInt i;
TPckg<TInt> pkg_int(i);
file.Read(pkg_int);

// Чтение структуры из файла
TPoint point;
TPckg<TPoint> pkg_point(point);
file.Read(pkg_point);
```

Класс `RFile` не содержит методов, позволяющих считывать данные до тех пор, пока в файле не встретится символ перевода строки. Поэтому, в случае необходимости хранения в файле нескольких строк произвольной длины, нужно самостоятельно добавлять информацию об их окончании или размере. Это можно сделать множеством способов, например, так.

```
// Массив строк
// (не забывайте, что число 3 задает зерно, а не размер)
CDesC8ArrayFlat* des_array = new (ELeave) CDesC8ArrayFlat(3);
CleanupStack::PushL(des_array);

// Инициализация массива
des_array->AppendL(_L8("Line 1"));
des_array->AppendL(_L8("Test Line 2"));
des_array->AppendL(_L8("Another line"));
const TInt KMaxLineSize = 50;

// Сохраняем кол-во строк в массиве
TInt cnt = des_array->Count();
```

```

TPckg<TInt> pkg_cnt(cnt);
file.Write(0, pkg_cnt);

// Сохраняем строки массива
for (TInt i = 0; i < cnt; i++)
{
    // Сохраняем длину строки
    TInt len = (*des_array)[i].Length();
    TPckg<TInt> pkg_len(len);
    file.Write(pkg_len);

    // Сохраняем строку
    file.Write((*des_array)[i]);
}

// Очищаем массив
des_array->Reset();

// Считываем число элементов в начале файла
file.Read(0, pkg_cnt);

// Считываем строки
for (TInt i = 0; i < cnt; i++)
{
    // Считываем размер строки
    TPckgBuf<TInt> pkg_len;
    file.Read(pkg_len);

    // Считываем строку заданного размера
    TBuf8<KMaxLineSize> buf;
    file.Read(buf, pkg_len());
    des_array->AppendL(buf);
}

CleanupStack::PopAndDestroy(des_array);

```

Класс RFile не позволяет считывать и записывать Unicode-строки, поэтому их необходимо конвертировать для хранения в 8-битовом дескрипторе.

В системе определен класс TFileText, предназначенный для построчного чтения и записи данных из объекта RFile, к которому он подключается с помощью метода Set(). Он работает с 16-битовыми строками, самостоятельно выполняя необходимые преобразования. Строки записываются с помощью метода Write(), принимающего лишь один аргумент — дескриптор с данными. В конце строки в файл автоматически добавляются символы CR-LF (0x0D, 0x0A). Чтение строк осуществляется с помощью метода Read(), также требующего лишь дескриптор для размещения считанных данных. При этом класс TFileText правильно обрабатывает не только символы перевода строки CR-LF, но и CR (возврат каретки), LF (следующая строка), а также LF-CR. В отличие от метода

`Read()` класса `RFile`, метод чтения `TFileText` сигнализирует о достижении конца файла, возвращая значение `KErrEof`.

Операцию чтения класс `TFileText` выполняет следующим образом. Из файла в буфер считывается небольшой блок данных, затем в них ищется символ перевода строки. Если он найден, то ограниченная им подстрока конвертируется в 16-битовый дескриптор. Если символ перевода строки не найден и конец файла не достигнут, то строка длиннее буфера и метод возвращает ошибку `KErrTooBig`. В противном случае весь буфер конвертируется в 16-битовый дескриптор.

Это означает, что длина строк, с которыми может работать объект класса `TFileText`, ограничена размерами буфера. На практике `TFileText` позволяет считывать строки длиной не более 256 символов.

Вот небольшой пример использования объекта `TFileText` для записи и чтения массива Unicode-строк.

```
// Массив Unicode строк
CDesCArrayFlat* des_array = new (ELeave) CDesCArrayFlat(3);
CleanupStack::PushL(des_array);

// Инициализация массива
des_array->AppendL(_L("Line 1"));
des_array->AppendL(_L("Test Line 2"));
des_array->AppendL(_L("Another line"));

// Создание файла и подключение объекта TFileText к нему
TFileText ft;
ft.Set(file);

// Сохраняем строки массива
for (TInt i = 0; i < des_array->Count(); i++)
    ft.Write((*des_array)[i]);

// Очищаем массив
des_array->Reset();
// Переносим текущую позицию чтения в начало файла
ft.Seek(ESeekStart);

// Считываем строки
TBuf<256> buf;
while (ft.Read(buf) != KErrEof)
    des_array->AppendL(buf);

CleanupStack::PopAndDestroy(des_array);
```

К сожалению, аналога класса `TFileText` для 8-битовых дескрипторов не существует.



Класс `RFile` содержит минимальный, но не всегда достаточный набор методов для чтения и записи данных. Такова его роль в архитектуре Symbian C++ API. Если для решения возникшей задачи класс `RFile` недостаточно удобен, то вам следует обратить внимание на более функциональные файловые потоки `RFileReadStream` и `RFileWriteStream`, рассматриваемые нами в следующем разделе.

При совместном использовании файла объект `RFile` может получить эксклюзивный доступ к области хранящихся в нем данных с помощью метода `Lock()`. В это время прочие клиенты объекта `RFile` не смогут производить операции чтения и записи над этим участком данных. Разблокировать его можно с помощью метода `Unlock()`.

Файловый сервер использует буферизацию при записи данных на диск. Объект `RFile` может использовать асинхронный или синхронный варианты метода `Flush()`, чтобы потребовать от сервера немедленно записать все данные на носитель и очистить буфер. Подобная операция вызывается автоматически при закрытии файла с помощью метода `Close()`. Трудно представить ситуацию, при которой разработчик может обоснованно вмешиваться в работу сервера таким образом. Поэтому на практике метод `Flush()` фактически не используется.

Перемещение текущей позиции

В вышеприведенных примерах текущая позиция в файле изменялась с помощью аргумента методов `Read()` или `Write()`. Класс `RFile` также позволяет изменить ее, не выполняя операций чтения или записи. Для этого служит метод `Seek()`. Он принимает два аргумента: способ перемещения и ссылку на число, содержащее величину перемещения. Способ перемещения задается с помощью одного из значений перечисления `Tseek`.

- `ESeekAddress` — используется для файлов, обрабатываемых в режиме XIP (например, файлы, находящиеся в ROM). Величина перемещения должна содержать позицию (offset) данных в файле от его начала. Она не должна быть отрицательной или превышать размер файла. По завершению работы метода в число, задававшее величину перемещения, записывается текущая позиция в файле.
- `ESeekStart` — отсчет от начала файла. Величина перемещения является позицией данных в файле. Она не должна быть отрицательной или превышать размер файла.
- `ESeekCurrent` — отсчет от текущей позиции. Величина перемещения может быть как отрицательной (движение к началу файла), так и положительной (движение к концу файла). По завершению работы метода, в число, задававшее величину перемещения, записывается текущая позиция в файле. Поэтому режим `ESeekCurrent` с перемещением 0 может быть использован для получения текущей позиции без ее изменения. Если в результате перемещения текущая позиция выходит за границы файла, то она приравнивается к его началу или концу.

- `ESeekEnd` — отсчет от конца файла. Величина перемещения должна быть отрицательной. По завершению работы метода в число, задававшее величину перемещения, записывается текущая позиция в файле. Если в результате перемещения текущая позиция выходит за границы файла, то она приравнивается к его началу или концу.

Следующие примеры демонстрируют использование метода `Seek()`.

```
file.SetSize(200);

TInt pos(100);
file.Seek(ESeekStart, pos);
// pos не изменился, текущая позиция — 100

pos = -50
file.Seek(ESeekEnd, pos);
// pos == 150, текущая позиция — 150

pos = -5;
file.Seek(ESeekCurrent, pos);
// pos == 145, текущая позиция — 145

// Получение текущей позиции
pos = 0;
file.Seek(ESeekCurrent, pos);
// pos == 145
```

Класс `TFileText` также предоставляет возможность перемещать текущую позицию чтения/записи с помощью метода `Seek()`. Но позиция может изменяться лишь на начало, либо конец файла. Для этого в качестве единственного аргумента метода `Seek()` передается либо значение `ESeekStart`, либо значение `ESeekEnd`.

Прочие методы класса `RFile`

Класс `RFile` предоставляет еще ряд методов, отчасти дублирующих функциональность класса `RFs` по работе с информацией о файле. Такими методами являются:

- `Att()` — получение атрибутов файла;
- `SetAtt()` — изменение атрибутов файла;
- `Modified()` — получение времени последнего изменения файла;
- `SetModified()` — переопределение времени последнего изменения файла;
- `Set()` — изменение атрибутов и времени создания файла;
- `Rename()` — переименование файла;
- `Drive()` — получение информации (объект `TDriveInfo`) о диске, на котором расположен файл.

Помимо этого, существует ряд методов для передачи и получения хендла на открытый файл между процессами и элементами клиент-серверной архитектуры. Дополнительную информацию по их использованию вы можете найти в справочнике SDK:

- `Adopt()` — подключение к уже открытому файлу по хендлу;
- `TransferToProcess()` — передача открытого файла другому процессу;
- `AdoptFromCreator()` — получение хендла на открытый файл, переданного с помощью метода `TransferToProcess()`;
- `TransferToClient()` — передача хендла на файл клиенту сервером в клиент-серверной архитектуре;
- `AdoptFromServer()` — получение хендла на открытый файл, переданного сервером клиенту в клиент-серверной архитектуре;
- `TransferToServer()` — передача хендла на файл серверу клиентом в клиент-серверной архитектуре;
- `AdoptFromClient()` — получение хендла на открытый файл, переданного клиентом серверу в клиент-серверной архитектуре.

В том случае, если процесс получил файл по хендлу и не знает его имени, могут пригодиться следующие методы:

- `Name()` — получение имени файла;
- `FullName()` — получение полного имени файла;
- `ChangeMode()` — позволяет изменить режим доступа `EFileShareExclusive` на `EFileShareReadersOnly` (и обратно), другие режимы доступа метод `ChangeMode()` не поддерживает.

Подготовка к сертификации ASD

- Умение отличить код, корректно устанавливающий сессию с файловым сервером (RFs), субсессию с файлом (RFile) и выполняющим чтение/запись данных в него.
 - Знание характеристик четырех методов RFile для открытия файла.
 - Умение выполнять чтение и запись данных в файл.
-

Потоки данных

Под **поток**ом **данных** (stream) в Symbian C++ понимается абстрактное внешнее представление объекта или нескольких объектов, а также методы для **сохранения** (externalization) и **восстановления** (internalization) объекта из него.

Представление должно быть достаточно полным. Это означает, что в нем должны содержаться все необходимые данные, для того чтобы восстановить состояние объекта на момент сохранения. Нет смысла сохранять в поток хендлы на открытые объекты (например, файлы), так как во время восстановления

они уже могут быть закрыты. Вместо этого следует поместить в него информацию, достаточную, чтобы снова найти и открыть тот же объект (например, полное имя файла). Также бессмысленно хранить указатели на принадлежащие объекту данные — следует помещать в поток сами данные, восстанавливать их вместе с объектом и снова передавать ему указатель на них.

Под термином “внешнее представление” подразумевается то, что оно может храниться на различных носителях. Чаще всего потоки подключаются к файлам, сокетам или областям памяти. Представление должно быть однозначным и не зависеть от хранилища. Поэтому размер сохраняемого типа данных и порядок байт в нем должны быть неизменны. Особенно внимательно следует относиться к внешнему представлению `TInt`, так как на его размер могут повлиять особенности процессора устройства или использованный компилятор языка C++.

Также важно соблюдать порядок при сохранении в поток объектов и их элементов, а также обратный порядок при их восстановлении.

Работа с потоком данных ведется с помощью **потока чтения** (read stream) или **потока записи** (write stream), инкапсулирующих **буфер потока** (stream buffer)

Буфером потока является объект, реализующий базовый интерфейс `MStreamBuf`. В классе `MStreamBuf` содержатся основные методы для ввода-вывода данных (бинарных строк), перемещения текущих позиций и т.д. В отличие от класса `RFile`, буфер потока имеет две текущих позиции (для чтения и для записи), независимых друг от друга.

Еще один базовый класс-интерфейс `TStreamBuf` наследован от `MStreamBuf` и добавляет к нему поддержку встроенного буфера для оптимизации производительности операций ввода-вывода. Не все буферы потока поддерживают буферизацию, поэтому часть из них наследуется от класса `MStreamBuf` напрямую, а часть — от класса `TStreamBuf`. Класс `TStreamBuf` не единственный унаследованный от `MStreamBuf` интерфейс, но более подробно на них мы останавливаться не будем. Экземпляры классов-интерфейсов создаваться не могут, так как их конструктор объявлен приватным.

Наследуемые от базовых интерфейсов классы буферов потока реализуют операции ввода-вывода для определенного хранилища данных. Например, унаследованный от `TStreamBuf` класс `RFileBuf` является буфером файлового потока, а `TMemBuf` — буфером потока памяти. Объекты этих классов можно создавать, но их методы ввода-вывода также неудобны (позволяют работать только с бинарными строками). Поэтому доступ к ним лучше осуществлять с помощью потоков чтения и записи.

Базовые классы потоков чтения и записи

Все классы потоков чтения и потоков записи унаследованы от базовых классов-интерфейсов `RReadStream` и `RWriteStream` соответственно. Рассмотрим предоставляемые ими методы, мы будем знать практически все, что необходимо для работы с унаследованными от них классами.

Класс `RReadStream` содержит следующие методы для чтения данных из буфера потока:

- `ReadInt8L()` — чтение `TInt8`;
- `ReadInt16L()` — чтение `TInt16`;
- `ReadInt32L()` — чтение `TInt`;
- `ReadReal32L()` — чтение `TReal`;
- `ReadReal64L()` — чтение `TReal64`;
- `ReadUInt8L()` — чтение `TUInt8`;
- `ReadUInt16L()` — чтение `TUInt16`;
- `ReadUInt32L()` — чтение `TUInt`;
- большое количество перегруженных методов `ReadL()` для чтения данных в 8- или 16-битовые дескрипторы, может считываться заданное число символов: либо до заполнения переданного дескриптора, либо до нахождения определенного байта в потоке данных;
- перегруженные методы `ReadL()` для чтения данных напрямую в поток записи `RWriteStream`;
- метод `ReadL()` для чтения указанного числа байт и записи их в определенную область памяти;
- метод `ReadL(TInt aLength)`, позволяющий пропустить указанное число байт, не считывая их.

Класс `RWriteStream` содержит следующие методы для записи данных в буфер потока:

- `WriteInt8L()` — запись `TInt8`;
- `WriteInt16L()` — запись `TInt16`;
- `WriteInt32L()` — запись `TInt`;
- `WriteReal32L()` — запись `TReal`;
- `WriteReal64L()` — запись `TReal64`;
- `WriteUInt8L()` — запись `TUInt8`;
- `WriteUInt16L()` — запись `TUInt16`;
- `WriteUInt32L()` — запись `TUInt`;
- большое количество перегруженных методов `WriteL()` для записи данных из 8- или 16-битовых дескрипторов; может выполняться запись заданного числа символов либо всего дескриптора;
- перегруженные методы `WriteL()` для записи данных напрямую в поток чтения `RReadStream`;
- метод `WriteL()` для записи указанного числа байт из определенной области памяти;
- `CommitL()` — запись содержимого буфера на носитель, буфер очищается: обычно эта операция выполняется автоматически по мере заполнения буфера либо при закрытии потока записи.

Оба класса `RReadStream` и `RWriteStream` предоставляют методы `PushL()` и `Pop()` для помещения и выталкивания объекта из стека очистки текущего потока. Кроме того, с помощью метода `Source()` класса `RReadStream` и `Sink()` класса `RWriteStream` можно получить указатель на интерфейс `MStreamBuf`, реализованный буфером потока. Наиболее интересными методами этого интерфейса являются:

- `SeekL()` — перемещение текущих позиций чтения или записи;
- `TellL()` — получение текущих позиций чтения или записи;
- `SizeL()` — размер буфера потока в байтах.

Это далеко не все, но мы перечислили наиболее востребованные методы базовых классов потоков чтения и записи.

Потоки чтения и записи

Мы рассмотрим лишь две пары наиболее часто используемых классов потоков чтения и записи: `RFileReadStream` и `RFileWriteStream`, а также `RMemReadStream` и `RMemWriteStream`. Классы `RFileReadStream` и `RMemReadStream` наследованы от класса `RReadStream`, а `RFileWriteStream` и `RMemWriteStream` — от класса `RWriteStream`.

Класс `RFileReadStream` определен в заголовочном файле `s32file.h` и реализует поток чтения для хранящихся в файле данных. Для работы с ним необходимо подключить библиотеку импорта `estor.lib`. Класс `RFileReadStream` инкапсулирует `RFileBuf` и реализует чтение данных из файлового потока. Он может подключаться к уже открытому с помощью объекта `RFile` файлу. Для этого используется метод `Attach()`, принимающий ссылку на объект `RFile`, и необязательный параметр (по умолчанию равный нулю), задающий новую позицию чтения. Существует также перегруженный конструктор с такими же аргументами. Как только объект `RFileReadStream` подключается к уже открытому файлу, объект `RFile` теряет хендл на него и больше не может использоваться в работе.

```
// RFile file;
RFileReadStream rfs(file);
CleanupClosePushL(rfs);
<...> // Работа с файловым потоком
CleanupStack::PopAndDestroy(&rfs);
```

Файловый поток чтения также может самостоятельно открыть файл с помощью метода `Open()`, аналогичного методу `Open()` класса `RFile`.

```
// RFs fs;
RFileReadStream rfs;
_LIT(KFileName, "c:\\file.txt");
User::LeaveIfError(rfs.Open(fs, KFileName,
    EFileWrite|EFileShareAny));
```

Закрывать поток и используемый им файл можно с помощью метода `Close()`.

Работа с классом `RFileWriteStream`, реализующим поток записи в файл, ведется точно так же, как с `RFileReadStream`. Отличие лишь в том, что наряду с методом `Open()` класс `RFileWriteStream` предоставляет методы `Create()`, `Replace()` и `Temp()`, аналогичные соответствующим функциям класса `RFile`.

Так как классы `RFileReadStream` и `RFileWriteStream` использует для доступа к файлу объект `RFileBuf` с поддержкой буферизации, то при частом выполнении операций чтения его производительность выше, чем у `RFile`. Поэтому в большинстве случаев использование файловых потоков предпочтительнее использования объектов `RFile`. В качестве примера изменим код для сохранения массива бинарных строк так, чтобы он сохранялся и восстанавливался из файлового потока.

```
RFs fs;
// Подключение к файловому серверу
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);

// Создаем массив 8-битовых дескрипторов
CDesC8ArrayFlat* des_array = new (ELeave) CDesC8ArrayFlat(3);
CleanupStack::PushL(des_array);

// Инициализация массива
des_array->AppendL(_L8("Line 1"));
des_array->AppendL(_L8("Test Line 2"));
des_array->AppendL(_L8("Another line"));

// Открываем файловый поток записи
RFileWriteStream fws;
_LIT(KFileName, "c:\\file.txt");
User::LeaveIfError(fws.Replace(fs, KFileName, EFileWrite
| EFileShareAny));
CleanupClosePushL(fws);

// Сохраняем кол-во строк в массиве
TInt cnt = des_array->Count();
fws.WriteInt32L(cnt);

// Сохраняем строки массива
for (TInt i = 0; i < cnt; i++)
{
    // Сохраняем длину строки
    fws.WriteInt32L((*des_array)[i].Length());

    // Сохраняем строку
```

```

        fws.WriteL((*des_array)[i]);
    }

    // Очищаем массив
    des_array->Reset();

    // Закрываем поток записи
    CleanupStack::PopAndDestroy(&fws);

    // Открываем файловый поток чтения
    RFileReadStream rfs;
    User::LeaveIfError(rfs.Open(fs, KFileName, EFileRead
        | EFileShareReadersOrWriters));
    // Поместить поток в стек очистки можно и так:
    rfs.PushL();

    // Считываем число элементов в начале файла
    cnt = rfs.ReadInt32L();

    // Создаем в куче дескриптор небольшого размера
    TInt buf_size(50);
    HBufC8* bufc = HBufC8::NewLC(buf_size);

    // Считываем строки
    for (TInt i = 0; i < cnt; i++)
    {
        // Считываем размер строки
        TInt len = rfs.ReadInt32L();

        // Если буфер слишком мал, увеличим его
        if (len > buf_size)
        {
            buf_size = len;
            bufc = bufc->ReAllocL(buf_size);
        }

        TPtr8 buf(bufc->Des());

        // Считываем строку заданного размера
        rfs.ReadL(buf, len);
        des_array->AppendL(*bufc);
    }

    // Освобождение bufc, rfs, des_array, fs
    CleanupStack::PopAndDestroy(4, &fs);

```

В данном примере чтение строк выполняется в дескриптор типа `HBufC8`. Так как максимальный размер объекта `HBufC8` задается при создании, то с его помощью можно реализовать чтение строк произвольной длины. В качестве

альтернативного способа сохранения строк в файле можно было бы отказаться от хранения длин и добавлять разделитель после каждой из них. Поток чтения предоставляет удобные методы для получения таких данных. Но в этом случае нельзя было бы заранее оценить необходимый размер буфера для размещения считанных данных.

Классы `RMemReadStream` и `RMemWriteStream` реализуют чтение и запись данных в память, инкапсулируя буфер потока `TMemBuf`, наследованный от класса `TStreamBuf`. Они объявлены в заголовочном файле `s32mem.h` и требуют подключения все той же библиотеки импорта `estor.lib`. Помимо стандартных методов базовых классов, классы потоков памяти предоставляют лишь одну функцию `Open()` для инициализации и эквивалентный ей перегруженный конструктор. Метод `Open()` в качестве аргументов принимает указатель на выделенный участок памяти и его размер. Пример.

```
// Выделяем 1 Кбайт памяти
TInt KMemBlock = 0x1000;
TAny* ptr = User::AllocLC(KMemBlock);

// Открытие потока для записи в память
RMemWriteStream mws(ptr, KMemBlock);
CleanupClosePushL(mws);

// Запись данных в память
mws.WriteInt8L(0x10);

// Закрытие потока
CleanupStack::PopAndDestroy(&mws);

// Открытие потока для чтения из памяти
RMemReadStream mrs(ptr, KMemBlock);
CleanupClosePushL(mrs);

// Чтение данных из памяти
TInt cnt = mrs.ReadInt8L();

// Закрытие потока и освобождение памяти
CleanupStack::PopAndDestroy(2, ptr);
```

Обратите внимание, что при закрытии потока область памяти, к которой он был подключен, не освобождается.



Иногда вместо классов `RMemReadStream` и `RMemWriteStream` удобнее использовать классы `RDesReadStream` и `RDesWriteStream`, осуществляющие чтение и запись данных в память 8-битового дескриптора. Данные классы в этой книге не рассматриваются, но работать с ними также просто.

Операторы << и >>

Для базовых классов потоков чтения и записи, а также всех их наследников определены операторы “<<” и “>>”, позволяющие сохранить и восстановить из потока какой-либо объект. Данные операторы заданы для многих стандартных типов данных, а также дескрипторов. Пример.

```
TInt KMemBlock = 0x1000;
TAny* ptr = User::AllocLC(KMemBlock);

RMemWriteStream mws(ptr, KMemBlock);
CleanupClosePushL(mws);

RMemReadStream mrs(ptr, KMemBlock);
CleanupClosePushL(mrs);

// Запись данных в память
mws << TInt32(0x10);
mws << _L("Line");

// Чтение из памяти
TInt32 i;
mrs >> i;
TBuf<10> buf;
mrs>> buf;

// Закрытие потоков и освобождение памяти
CleanupStack::PopAndDestroy(3, ptr);
```

Использование стека очистки в данном примере необходимо, поскольку **операторы << и >> могут вызвать сброс** (например, с кодом KErrEof). Поэтому в названии всех использующих их функций должен присутствовать суффикс “L”.

При сохранении дескриптора в поток с помощью оператора << в него сначала помещается его размер, а затем — содержимое. При этом, если размер строки велик, то при его записи проводится сжатие за счет неиспользуемых разрядов. Восстановить дескриптор из такого представления можно с помощью оператора >>. **В целях экономии ресурсов чтение и запись дескрипторов в поток рекомендуется выполнять с помощью операторов << и >>.**

Поддержку операторов << и >> можно добавить в любой класс, реализовав в нем следующие методы.

```
void ExternalizeL(RWriteStream &aStream) const;
void InternalizeL(RReadStream &aStream);
```

Метод ExternalizeL() для такого класса будет вызываться при использовании оператора <<, а метод InternalizeL() — при использовании оператора >>. Пример.

```

struct TMyStreamData
{
    void ExternalizeL(RWriteStream& aStream) const;
    void InternalizeL(RReadStream& aStream);

    TBuf<0x100> iDes;
    TInt32 iInt32;
    TReal64 iReal64;
};

void TMyStreamData::ExternalizeL(RWriteStream& aStream) const
{
    aStream << iDes;
    aStream << iInt32;
    aStream << iReal64;
}

void TMyStreamData::InternalizeL(RReadStream& aStream)
{
    aStream >> iDes;
    aStream >> iInt32;
    aStream >> iReal64;
}

```

Сохранение в поток и восстановление из потока объектов класса TMyStreamData выполняется следующим образом.

```

TMyStreamData data;

// Инициализация
data.iDes = _L("Line 1");
data.iInt32 = 7;
data.iReal64 = 1.0 / 3.0;

// Сохранение в поток (RMemWriteStream mws)
mws << data;

// Восстановление из потока (RMemReadStream mrs)
TMyStreamData data2;
mrs >> data2;

```

Методы ExternalizeL() и InternalizeL() можно использовать и напрямую:

```

TMyStreamData data2;
data2.InternalizeL(mrs);

```

В заключение следует отметить существование довольно удобного класса TCardinality, позволяющего записывать *положительные* значения TInt в поток в сжатом виде и считывать их. Он объявлен в заголовочном

файле `s32strm.h`. По сути, класс `TCardinality` является простой оболочкой для класса `TInt`, содержащей реализацию методов `ExternalizeL()` и `InternalizeL()`. Числовое значение передается в объект `TCardinality` в конструкторе. Его сжатие при хранении в потоке достигается за счет неиспользуемых разрядов. Если значение попадает в диапазон от 0 до 127, то для его хранения будет использован 1 байт, от 128 до $2^{14}-1$ — 2 байта, для прочих — 4 байта. Пример использования.

```
// Сохранение в поток (RMemWriteStream mws)
mws << TCardinality(100);

// Восстановление из потока (RMemReadStream mrs)
TCardinality c;
mrs >> c;
TInt i = (TInt) c;
```

Подготовка к сертификации ASD

- Знание причин того, почему использование потоков можно предпочесть работе с `RFile`.
 - Понимание того, как используются метод `ExternalizeL()` и оператор `<<` в классе `RWriteStream` для записи объекта в поток, а также метод `InternalizeL()` и оператор `>>` в классе `RReadStream` для его чтения.
 - Знание того, что операторы `>>` и `<<` могут вызвать сброс.
-

Хранилища данных

Хранилище (store) является коллекцией потоков данных и может располагаться в памяти, другом потоке или хранилище, а также в файле (самый распространенный случай). С помощью системы потоков можно представить практически любой объект или набор объектов. Например, в хранилище очень удобно записывать документы, которые могут содержать встроенные изображения, графики и прочие медиаданные. Кроме того, хранилища поддерживают механизм транзакций, что делает их довольно надежными.

Существует несколько классов, реализующих разнообразные хранилища. Их иерархия представлена на рис. 6.7. Курсивом обозначены абстрактные классы.

Все классы хранилищ данных происходят от абстрактного класса `CStreamStore` и делятся на **постоянные** (persistent) и **непостоянные** (non-persistent). Непостоянные хранилища чаще всего создаются в памяти и автоматически удаляются сразу после их закрытия. Конкретным классом, реализующим такой способ хранения потоков, является `CBufStore`.

Постоянные хранилища происходят от абстрактного класса `CPersistentStore` (в нем вводятся корневые потоки). Они продолжают существовать после

завершения работы приложения и могут быть открыты вновь. Для их размещения лучше всего подходят файлы, поддержка работы с которыми добавляется в абстрактном классе `CFileStore`. Наконец, от класса `CFileStore` происходят два конкретных класса `CDirectFileStore` и `CPermanentFileStore`, объявленные в файле `s32file.h`. Именно они чаще всего используются на практике. В дальнейшем мы будем говорить только о файловых хранилищах.

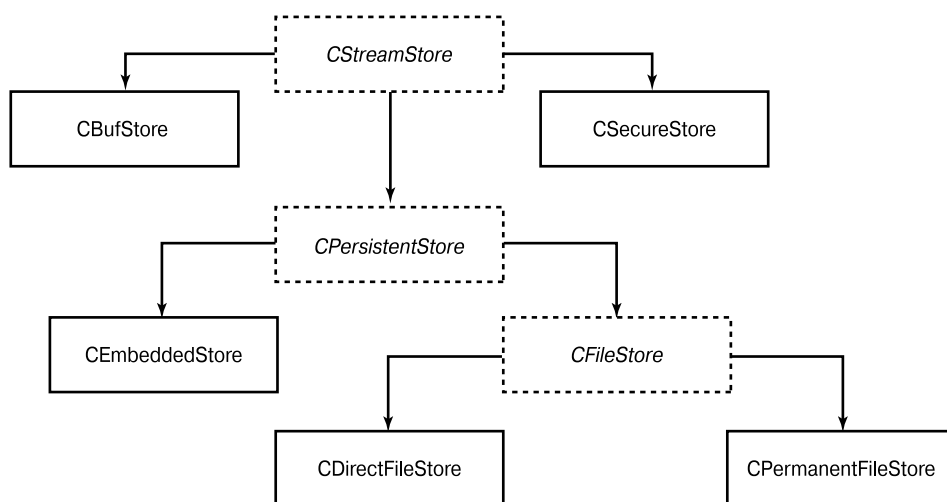


Рис. 6.7. Иерархия классов хранилищ

Классы `CSecureStore` и `CEmbeddedStore` являются вспомогательными. Класс `CSecureStore` обеспечивает шифрование создаваемых в хранилище потоков, а `CEmbeddedStore` позволяет поместить одно хранилище в поток другого.

Организация файлового хранилища

Содержащий хранилище файл начинается с трех идентификаторов `UID`. Значение `UID1` указывает на тип хранилища (создано с помощью класса `CDirectFileStore` или `CPermanentFileStore`). Значения `UID2` и `UID3` не важны, но могут использоваться разработчиком для различных проверок. Например, чтобы быть уверенным в том, что ваша программа не откроет файл, предназначенный для другого приложения, можно использовать ее идентификатор `UID3` в качестве `UID3` хранилища. А в `UID2` можно записать его версию. Идентификаторы файла хранилища устанавливаются с помощью методов класса `CFileStore`.

Файловое хранилище может содержать несколько потоков, работа с которыми ведется с помощью классов `RStoreReadStream` и `RStoreWriteStream`, объявленных в заголовочном файле `s32std.h` (рис. 6.8). Доступ к потокам

осуществляется по уникальным идентификаторам. При создании нового потока в файловом хранилище разработчик получает его идентификатор в объекте `TStreamId`, который инкапсулирует объект `TUint32`, а также предоставляет методы `ExternalizeL()` и `InternalizeL()`. Класс файлового хранилища не позволяет узнать идентификаторы содержащихся в нем потоков, кроме одного — **корневого** (root stream). Обычно при создании нового файлового хранилища и задании его идентификаторов в нем сразу же создается корневой поток. Но в общем случае корневым потоком может быть назначен любой поток хранилища. Основное назначение корневого потока — хранение данных, позволяющих обратиться к остальным потокам хранилища (т.е. объектам класса `TStreamId`). Их можно разместить в нем различными способами, но рекомендуется использовать для этих целей экземпляр класса `CStreamDictionary`.

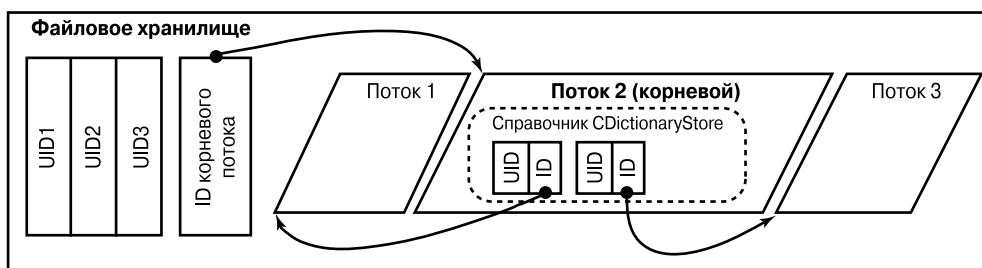


Рис. 6.8. Упрощенное представление файлового хранилища с тремя потоками

Класс `CStreamDictionary` объявлен в заголовочном файле `s32std.h` и инкапсулирует массив пар значений `<TUid, TStreamId>`, где `TStreamId` — идентификатор потока, а `TUid` — идентификатор, ставящийся ему в соответствие разработчиком. Поскольку на этапе разработки невозможно предсказать, какой идентификатор `TStreamId` будет присвоен потоку, то возникает необходимость поставить ему в соответствие некоторое predetermined значение, которое будет однозначно определять, что же в нем хранится. Класс `CStreamDictionary` позволяет получать или удалять элементы массива по `TUid`, а также поддерживает операторы `<<` и `>>`.

Создание хранилища

Существует два типа файловых хранилищ: **долговременные файловые хранилища** (permanent file store), создаваемые с помощью класса `CPermanentFileStore`, и **непрерывные хранилища файлов** (direct file store), создаваемые с помощью класса `CDirectFileStore`.

Принципиальная разница между ними заключается в том, что потоки, помещенные в долговременные хранилища, могут впоследствии изменяться, перезаписываться и удаляться. При удалении или замене старый поток лишь помечается как удаленный, хотя физически остается в файле. Поэтому для долговременных хранилищ характерна фрагментация, для борьбы с которой существует

операция сжатия. Использование хранилищ такого типа свойственно приложениям, часто изменяющим данные и загружающим лишь часть из них в память (например, таблицы баз данных и документы в редакторах).

Потоки в непрерывном хранилище после своего создания и сохранения могут лишь открываться для чтения. При необходимости обновить данные в хранилище весь файл должен быть заменен. Поэтому применение непрерывных хранилищ свойственно приложениям, использующим данные лишь для чтения (например, уровни в игре) либо хранящих полную рабочую копию данных в памяти (например, настройки).

Для создания нового файла хранилища классы `CPermanentFileStore` и `CDirectFileStore` предоставляют методы `CreateL()`, `ReplaceL()` и `TempL()`, а также их `xLC()`-варианты. По назначению и составу аргументов они аналогичны методам `Create()`, `Replace()` и `Temp()` класса `RFile`, поэтому подробно останавливаться на них мы не будем.

Сразу после создания в новом файле хранилища необходимо установить набор идентификаторов `UID`. Сделать это можно с помощью метода `SetTypeL()` базового класса `CFileStore`, принимающего аргумент типа `TUidType`. Значение `UID1` обязательно должно быть правильным идентификатором типа хранилища. Получить такой идентификатор можно с помощью метода `Layout()`. Его результатом для объектов `CPermanentFileStore` является `KPermanentFileStoreLayoutUid (0x10000050)`, а для объектов `CDirectFileStore` — `KDirectFileStoreLayoutUid (0x10000037)`. Впоследствии значения `UID` открытого файлового хранилища можно получить с помощью метода `Type()`.

```
RFs fs;
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);
_LIT(KStoreName, "c:\\store.dat");

// Создание нового долговременного хранилища
CPermanentFileStore* store = CPermanentFileStore::CreateLC(fs,
    KStoreName, EFileWrite);

// Создаем объект TUidType с заданным UID1
TUidType uids(store->Layout());
store->SetTypeL(uids);
```

Затем в хранилище необходимо создать корневой поток. Потоки создаются методами `CreateL()` и `CreateLC()` класса `RStoreWriteStream`. В них передается ссылка на файлоое хранилище, а результатом является идентификатор `TStreamId` нового потока. Если в поток были записаны данные, то перед его закрытием необходимо вызывать метод `CommitL()`. Чтобы сделать поток корневым, достаточно указать его идентификатор в методе `SetRootL()` хранилища, после чего его можно будет получить с помощью `Root()`.

```
// Создаем новый поток в хранилище
RStoreWriteStream root_strm;
TStreamId root_id = root_strm.CreateLC(*store);
// Помечаем поток как корневой
store->SetRootL(root_id);
root_strm.CommitL();
store->CommitL();
```

Классы потоков хранилища используют буферизацию при выполнении операций чтения и записи. Метод `CommitL()` класса `RStoreWriteStream` записывает содержимое буфера в хранилище и очищает его. Эта процедура не выполняется автоматически при закрытии потока. В случае если разработчик хочет создать в хранилище новый поток, но не желает закрывать уже открытые потоки записи, в них также должен быть предварительно вызван метод `CommitL()`.

Метод `CommitL()` имеется и в самом хранилище, но выполняет несколько иную функцию. Хранилище поддерживает механизм подтверждений, подобный транзакциям в базах данных. Все внесенные в файл изменения можно либо подтвердить с помощью методов `Commit()` или `CommitL()`, либо откатить до предыдущего подтвержденного состояния с помощью `Revert()` или `RevertL()`, если что-то пошло не так.

В корневом потоке следует сохранить идентификаторы всех прочих потоков хранилища. Для этого можно воспользоваться рекомендованным классом `CStreamDictionary`.

```
// Создание справочника
CStreamDictionary* dict = CStreamDictionary::NewLC();

// Создание еще одного потока
RStoreWriteStream int_strm;
TStreamId strm_id = int_strm.CreateLC(*store);

// Запись данных в поток
int_strm << TInt32(10);
int_strm.CommitL();

// Закрытие int_stream
CleanupStack::PopAndDestroy(&int_strm);

// Добавление потока в справочник
const TUid KMyIntegerStream = {0x101010FF};
dict->AssignL(KMyIntegerStream, strm_id);

// Сохранение справочника в корневом потоке
root_strm << *dict;
root_strm.CommitL();

// Удаление dict и закрытие root_strm
CleanupStack::PopAndDestroy(2, &root_strm);
```

```
// Подтверждение изменений в хранилище
store->CommitL();
```

```
// Закрытие хранилища
CleanupStack::PopAndDestroy(store);
```

Как видите, мы создали новый поток для хранения числа и поместили его идентификатор в справочник, добавив к нему соответствие значению KMyIntegerStream. Именно по этому значению KMyIntegerStream в будущем проще всего найти и открыть поток.

Открытие хранилища и чтение данных

Открыть хранилище можно с помощью методов OpenL() или OpenLC(), аналогичных методу Open() класса RFile, либо с помощью методов From() или FromL(), принимающих открытый объект RFile по ссылке. Открывать хранилище следует тем классом, с помощью которого оно было создано.

После открытия хранилища можно получить ID корневого потока, считать справочник и обратиться к другим содержащимся в хранилище потокам.

```
// Открываем существующее хранилище
CPermanentFileStore* store = CPermanentFileStore::OpenLC(fs,
    KStoreName, EFileRead);
```

```
// Открытие корневого потока для чтения
RStoreReadStream root_strm;
root_strm.OpenLC(*store, store->Root());
```

```
// Создание справочника
CStreamDictionary* dict = CStreamDictionary::NewLC();
```

```
// Считываем справочник
root_strm >> *dict;
const TUid KMyIntegerStream = {0x101010FF};
```

```
// Ищем id потока по ассоциированному идентификатору
TStreamId strm_id = dict->At(KMyIntegerStream);
```

```
// Удаляем справочник и закрываем поток
CleanupStack::PopAndDestroy(2, &root_strm);
```

```
RStoreReadStream int_strm;
int_strm.OpenLC(*store, strm_id);
TInt32 data;
int_strm >> data;
```

```
// Закрываем поток и хранилище
CleanupStack::PopAndDestroy(2, store);
```

Сжатие хранилища

В долговременном хранилище потоки могут удаляться с помощью методов `Delete()` и `DeleteL()` базового класса `CStreamStore`, а также заменяться с помощью `ReplaceL()` и `ReplaceLC()` класса `RStoreWriteStream`. При этом поток в файле помечается как несуществующий, но физически не удаляется, так как при этом пришлось бы перезаписывать часть данных. Поэтому с течением времени в долговременном файловом хранилище могут накапливаться неиспользуемые области. С помощью метода `ReclaimL()`, доступного для объектов `CPermanentFileStore`, эти области обнаруживаются и помечаются как готовые к использованию, поэтому их смогут вновь использовать новые потоки. Метод `CompactL()` позволяет удалить неиспользуемые области из файла и тем самым уменьшить его размер. И `ReclaimL()`, и `CompactL()` возвращают размер свободного пространства в байтах.

Операция сжатия может занять довольно продолжительное время. Существует класс `RStoreReclaim`, позволяющий выполнять ее пошагово.

Класс `CDictionaryFileStore`

Класс `CDictionaryFileStore`, объявленный в файле `s32file.h`, инкапсулирует класс `CPermanentFileStore` и позволяет автоматизировать работу с корневым потоком и справочником — они создаются в нем автоматически. Для чтения и записи потоков в таком хранилище используются классы `RDictionaryReadStream` и `RDictionaryWriteStream`. Их методы вместо идентификатора `TStreamId` потока принимают идентификатор `UID` и автоматически определяют нужный `TStreamId` в справочнике. Например, вот как будет выглядеть вышеприведенный пример с использованием класса `CDictionaryFileStore`.

```
// Открытие хранилища, если не существует - будет создано
_LIT(KStoreName, "c:\\store.dat");
CDictionaryFileStore* store = CDictionaryFileStore::OpenLC(fs,
    KStoreName, KNullUid);
// Корневой поток и справочник создаются автоматически

// Открытие потока, если не существует - будет создан
RDictionaryWriteStream dws;
const TUid KMyIntegerStream = {0x101010FF};
dws.AssignLC(*store, KMyIntegerStream);
// Новый поток автоматически регистрируется в справочнике

// Запись данных
dws << TInt32(10);
dws.CommitL();
CleanupStack::PopAndDestroy(&dws);
store->CommitL();
```

```
RDictionaryReadStream drs;  
// Открываем поток по UID  
drs.OpenLC(*store, KMyIntegerStream);  
  
// Чтение данных  
TInt32 data;  
drs >> data;  
CleanupStack::PopAndDestroy(2, store);  
CleanupStack::PopAndDestroy(&fs);
```

В отличие от `CPermanentFileStore`, класс `CDictionaryFileStore` предоставляет лишь один метод для открытия хранилища — `OpenL()`. Этот метод позволяет создать хранилище в случае, если оно не существовало. Кроме того, в нем можно указать идентификатор `UID3` хранилища, который будет проверяться при открытии либо сохраняться при создании. Класс `CDictionaryFileStore` предоставляет следующие полезные методы:

- `IsNullL()` — возвращает значение `ETrue`, если справочник в корневом потоке пуст;
- `IsPresentL()` — проверяет наличие в хранилище потока с заданным `UID`.

Для открытия потока в хранилище используется метод `AssignL()` класса `RDictionaryWriteStream`. В случае если поток с заданным `UID` не существовал, он будет создан. Во всем остальном `RDictionaryWriteStream` очень похож на класс `RStoreWriteStream`.

Класс TSwizzle

При работе со сложными объектами, размещенными в хранилище, часто возникает необходимость в их частичной загрузке. Например, при просмотре содержащего изображение документа само изображение может не загружаться в оперативную память до тех пор, пока пользователь не откроет страницу, на которой оно находится. Для представления таких объектов используются классы семейства `TSwizzle` (*swizzle* — англ., вид коктейля). Они способны хранить `ID` потока до тех пор, пока объект не загружен в память либо хранить указатель на него после завершения загрузки, а также предоставляют методы для проверки, в каком именно состоянии сейчас находится объект.

Подробное рассмотрение работы этого класса выходит за рамки нашей книги. Поэтому всю необходимую информацию вы сможете почерпнуть из справочника SDK.

Подготовка к сертификации ASD

- Умение распознать классы хранилищ и потоков, а также знание основных характеристик каждого из них (например, базовый класс, хранилище в памяти, долговременность, модифицируемость и т.д.).

-
- Понимание того, как используются метод `ExternalizeL()` и оператор `<<` в классе `RWriteStream` для записи объекта в поток, а также метод `InternalizeL()` и оператор `>>` в классе `RReadStream` для его чтения.
-

Базы данных

Базы данных незаменимы при работе с большими объемами данных и часто используются в разнообразных службах Symbian OS. Например, в виде базы данных хранится контактная книга и настройки коммуникационных сервисов. Поэтому с самых ранних версий в Symbian OS присутствует встраиваемая и клиент-серверная СУБД, позволяющая создавать локальные реляционные базы данных. Таблицы и индексы такой базы хранятся в одном файле. Для доступа и управления сервером СУБД предоставляется Symbian DBMS API. Работа с таблицами и данными ведется с помощью объектов. Язык запросов SQL поддерживается в СУБД Symbian, но его функциональность сильно ограничена.

Не так давно на Symbian OS была портирована популярная библиотека SQLite сообщества OpenSource, предоставляющая локальную встраиваемую СУБД с более полной поддержкой стандартов языка SQL. Этот продукт получил название Symbian SQL. К сожалению, библиотека Symbian SQL вошла в состав платформы S60 лишь с версии S60 3 FP2 (на базе Symbian 9.3). На устройствах под управлением S60 3 FP1 (на базе Symbian 9.2) ее необходимо устанавливать самостоятельно (включая в дистрибутив приложения или уведомляя об этом пользователя). А на платформе S60 3-й редакции (на базе Symbian 9.1) и более старых версиях она официально не поддерживается.

Сетевые СУБД для платформы Symbian представлены коммерческим продуктом Oracle Database Lite. Это чрезвычайно мощная СУБД с высокой степенью масштабируемости, которая обеспечивает работу приложений даже в отсутствие подключения к сети, осуществляя лишь периодическую синхронизацию с сервером Oracle Database. СУБД Oracle Database Lite поддерживает работу с устройствами под управлением операционных систем Symbian OS 7.x–9.x, Windows 2003/XP/Vista, Windows Mobile, Pocket PC, Linux, embedded Linux и некоторых других.

Здесь мы будем рассматривать работу только с Symbian DBMS API для Symbian 9.x.

Доступ к базе данных

Как уже отмечалось ранее, СУБД Symbian OS может использоваться и как встраиваемая (embedded) для прямого доступа к файлу базы данных из приложения, и как клиент-серверная база данных. Ее функциональность реализована в библиотеке `edbms.dll`. Поэтому для работы с СУБД в ММР-файле проекта необходимо подключить библиотеку импорта `edbms.lib`. Режим доступа к базе данных может быть в трех вариантах:

- исключительный (чтение и запись);
- разделяемый (только на чтение);
- разделяемый (на чтение и запись).

В случае когда приложению нужен исключительный доступ к базе данных либо разделяемый, но только для чтения, библиотеку `edbms.dll` можно использовать для работы с ней напрямую. Если же хотя бы одному из клиентов нужен разделяемый доступ с возможностью записи данных, то всем клиентам потребуется использовать клиент-серверный вариант той же СУБД. Для этого приложение устанавливает сессию с системным сервером СУБД (также использующим библиотеку `edbms.dll`), и в дальнейшем вся работа с базой данных ведется через него. Сервер СУБД является временным и запускается при первом подключении к нему (программа `edbsrv.exe`, UID3: 0x100012A5).

В большинстве случаев приложение может ограничиться встраиваемым вариантом работы СУБД. Тогда его следует предпочесть клиент-серверному способу, поскольку без использования механизмов IPC данные передаются быстрее.

Классы баз данных

Все классы, необходимые для работы с базами данных, объявлены в заголовочном файле `d32dbms.h`. Основные функции для работы с базой данных (задание схемы, поддержка транзакций, возможность выполнения SQL-запросов и пр.) определены в абстрактном классе `RDbDatabase`. Этот базовый класс реализуется двумя его конкретными потомками: `RDbStoreDatabase` и `RDbNamedDatabase`.

Класс `RDbStoreDatabase` используется для создания базы данных в одном из потоков хранилища. Хранилище должно поддерживать механизм подтверждений (транзакций). Фактически из существующих типов классов хранилищ использоваться может только класс `CPermanentFileStore`. Доступ через объект `RDbStoreDatabase` может осуществляться только напрямую, без участия сервера СУБД. Поэтому с помощью класса `RDbStoreDatabase` нельзя организовать разделяемый доступ с возможностью записи данных. Из “плюсов” класса `RDbStoreDatabase` следует отметить поддержку сжатия (метод `CompressL()`) и распаковки (метод `DecompressL()`) баз данных. Класс `RDbNamedDatabase` такую возможность не предоставляет. Сжиматься могут только незашифрованные базы данных. Индексы не сжимаются. После выполнения сжатия база данных доступна только для чтения и требует немного дополнительной памяти при извлечении данных, но на скорости работы это отражается незначительно. Довольно большой объем памяти требуется при выполнении самих операций сжатия и распаковки.

Для создания новой базы данных класс `RDbStoreDatabase` предоставляет метод `CreateL()`, принимающий указатель на объект реализующего хранилище класса и возвращающий ID потока, в котором были помещены данные. Для открытия в хранилище существующей базы данных класс `RDbStoreDatabase`

предоставляет метод `OpenL()`, принимающий указатель на него и ID потока. Следующий пример демонстрирует создание новой базы данных в хранилище.

```
// Имя базы данных, может быть любым.
// Рекомендованное расширение: ".db"
_LIT(KMydatabase, "c:\\mydb.db");
// Создание долговременного файлового хранилища
CPermanentFileStore* store =
    CPermanentFileStore::ReplaceLC(fs, KMydatabase, EFileWrite);
store->SetTypeL(TUidType(store->Layout()));

RDbStoreDatabase db;
// Создание базы данных в новом потоке
TStreamId strm_id = db.CreateL(store);
CleanupClosePushL(db);

// Назначаем поток корневым, чтобы найти в будущем
store->SetRootL(strm_id);
store->CommitL();
<...> // Работа с базой данных
CleanupStack::PopAndDestroy(2, store);
```

Работа с базой данных в классе `RDbStoreDatabase` ведется точно так же, как и в классе `RDbNamedDatabase` (поскольку большинство методов наследуются от общего предка), поэтому в дальнейшем мы будем рассматривать только его.

Класс `RDbNamedDatabase` также использует файловое хранилище для создания базы данных, но предоставляет несколько более высокий уровень API. При создании достаточно указать лишь имя файла базы данных, и хранилище будет создано автоматически. По умолчанию все данные хранятся в его корневом потоке. Такой формат называется “ерос”. Теоретически разработчик может создать собственную библиотеку расширения для сервера СУБД, добавляющую поддержку новых форматов. Класс `RDbNamedDatabase` позволяет работать с базой данных как напрямую, так и через сервер СУБД. В данный момент мы будем рассматривать только первый вариант, а работу через сервер изучим несколько позднее.

Начать работу с базой данных можно с помощью методов `Create()`, `Open()` и `Replace()`. Все они принимают в качестве аргументов ссылку на сессию файлового сервера, имя файла базы данных, а также строку с параметрами. В методе `Open()` можно также указать режим доступа к файлу (чтение/запись). Создание новой базы данных выполняется следующим образом.

```
_LIT(KMydatabase, "c:\\mydb.db");
_LIT(KDBFormat, "[101010FF]");
RDbNamedDatabase db;
User::LeaveIfError(db.Create(fs, KMydatabase, KDBFormat));
CleanupClosePushL(db);
```

Дескриптор `KDBFormat` содержит строку с параметрами. Параметр в данном случае всего один, а строка имеет формат “[UID]”. Передаваемый идентифика-

тор помещается в поле UID3 файла хранилища и используется системой при поиске баз данных определенного типа (например, содержащих номера телефонов “контактных книг”). При создании БД собственной схемы разработчик может использовать любое значение. Получить дескриптор со строкой в формате “[UID]” можно с помощью метода Name() класса TUid.

```
TUIdName b = TUid::Uid(0x101010FF).Name();
// b - это TBuf<10>, содержащий "[101010ff]"
```

Таблицы

Все таблицы и их столбцы в базе данных именованы. Имена таблиц должны быть уникальны в рамках базы данных, а столбцов — уникальны в рамках таблицы, к которой они принадлежат. Имена не зависят от регистра и на них распространяются следующие ограничения:

- должны начинаться с буквы алфавита;
- должны содержать только буквы алфавита, цифры и символ подчеркивания “_”;
- максимальный размер имени — 64 символа.

Вышеприведенные правила верны для всех именованных объектов базы данных (таблицы, столбцы, индексы и пр.).

Максимальное количество таблиц и строк в них ограничено лишь объемом свободных ресурсов системы. Количество столбцов в таблице косвенно ограничено максимальным размером записи — 8200 байт (чуть больше 8 Кбайт). Например, вы сможете создать только 32 столбца типа LongText8 или Text8[255], либо 100 столбцов типа Text8[80].

Столбец таблицы задается с помощью структуры типа TDbCol, хранящей ее имя (iName), тип (iType), максимальный размер (iMaxLength) и атрибуты (iAttributes). Все эти параметры также можно установить в различных перегруженных конструкторах TDbCol. Об ограничениях на формат имен столбцов уже говорилось выше. Для хранения имени часто используют дескриптор типа TDbColName, являющийся переопределением от TBuf<KDbMaxColName>, где KDbMaxColName — константа, равная 64. Тип столбца задается одним из значений перечисления TDbColType. Их описание приведено в табл. 6.2. Размер типа данных столбца не должен превышать значений, указанных в этой таблице. Атрибутом столбца является комбинация значений TDbCol::ENoNull и TDbCol::EAutoIncrement, полученная с помощью операции логического “ИЛИ”.

Таблица 6.3. Допустимые типы данных в столбцах таблицы

Тип	Описание	Область значений или длина	Длина (байт)	Поддержка автоматического увеличения	Может быть ключом индекса
EDbColBit	1 бит	0 ~ 1	1 бит	Да	Да
EDbColInt8	8-битовое знаковое число	$-2^7 \sim 2^7 - 1$	1	Да	Да

Окончание табл. 6.3

Тип	Описание	Область значений или длина	Длина (байт)	Поддержка автоматического увеличения	Может быть ключом индекса
EDbColUInt8	8-битовое беззнаковое число	$0 \sim 2^8 - 1$	1	Да	Да
EDbColInt16	16-битовое знаковое число	$-2^{15} \sim 2^{15} - 1$	2	Да	Да
EDbColUInt16	16-битовое беззнаковое число	$0 \sim 2^{16} - 1$	2	Да	Да
EDbColInt32	32-битовое знаковое число	$-2^{31} \sim 2^{31} - 1$	4	Да	Да
EDbColUInt32	32-битовое беззнаковое число	$0 \sim 2^{32} - 1$	4	Да	Да
EDbColInt64	64-битовое знаковое число	$-2^{63} \sim 2^{63} - 1$	8	Нет	Да
EDbColReal32	32-битовое число с плавающей точкой	$1,4 \times 10^{-45} \sim 3,40282 \times 10^{38}$	4	Нет	Да
EDbColReal64	64-битовое число с плавающей точкой	$2,2 \times 10^{-308} \sim 1,79769 \times 10^{308}$	8	Нет	Да
EDbColDateTime	Время и дата	01.01.0000 ~ 31.12.9999	8	Нет	Да
EDbColBinary	Небольшой массив байт произвольной длины	$0 \sim 255$	≤ 256	Нет	Нет
EDbColLongBinary	Большой массив байт произвольной длины	$0 \sim 2^{31}$	$\leq 2 \text{ Гб}$	Нет	Нет
EDbColText8	Небольшая не Unicode-строка произвольной длины	$0 \sim 256$	≤ 256	Нет	Да
EDbColText16	Небольшая Unicode-строка произвольной длины	$0 \sim 256$	≤ 512	Нет	Да
EDbColLongText8	Большая не Unicode-строка произвольной длины	$0 \sim 2^{30}$	$\leq 2 \text{ Гб}$	Нет	Ограниченно
EDbColLongText16	Большая Unicode-строка произвольной длины	$0 \sim 2^{31}$	$\leq 2 \text{ Гб}$	Нет	Ограниченно
EDbColText	Небольшая строка различной длины символов нейтрального размера. Для 8-битовых версий Symbian OS эквивалентен EDbColText8. Для Unicode версий– EDbColText16				
EDbColLongText	Большая строка различной длины символов нейтрального размера. Для 8-битовых версий Symbian OS эквивалентен EDbColLongText8. Для Unicode версий– EDbColLongText16				

Примеры.

```
// TInt32 с автоинкрементом
_LIT(KIdxCol, "IdxCol");
TDbCol col_Idx(KIdxCol, EDbColUInt32);
col_Idx.iAttributes = TDbCol::ENotNull |
                    TDbCol::EAutoIncrement;

// Unicode строка не более 20 символов
TDbCol col_Name(_L("StrCol"), EDbColText16, 20);
col_Name.iAttributes = 0;

// Массив байт размером 1Кбайт
TDbCol col_Data;
col_Data.iName = _L("DataCol");
col_Data.iType = EDbColLongBinary;
col_Data.iMaxLength = 0x1000;
col_Data.iAttributes = 0;
```

Обратите внимание на то, что поле `iAttributes` инициализируется всегда. К сожалению, нет возможности задать его значение в конструкторе, поэтому сразу после создания объекта в стеке в нем может находиться все что угодно. Если вы забудете присвоить полю `iAttributes` одно из допустимых значений, то при попытке создания или изменения таблицы с использованием такого столбца получите ошибку `KErrNotSupported`.

Для хранения объектов `TDbCol` используется класс `CDbColSet`, инкапсулирующий `CArrayPakFlat<TDbCol>` и предоставляющий следующие методы:

- `AddL()` — добавление объекта `TDbCol` в набор;
- `Remove()` — удаление столбца из набора по имени;
- `Clear()` — очищение массива;
- `Count()` — число столбцов в наборе;
- `Col()` — возвращает указатель на объект `TDbCol`, заданный его именем, или `NULL`, если такой объект не найден;
- `ColNo()` — возвращает индекс столбца в наборе, заданного его именем, или значение `KDbNullColNo` (равное 0) в случае, если он не найден;
- оператор `[]` — позволяет получить ссылку на объект `TDbCol` по индексу, причем индекс первого столбца равен 1, так как нулевое значение зарезервировано для `KDbNullColNo`.

Набор столбцов может использоваться для создания или изменения таблиц в базе данных. Для этого в базовом классе `RDbDatabase` существуют методы `CreateTable()` и `AlterTable()`. Оба принимают в качестве параметров имя таблицы и ссылку на объект `CDbColSet`. Примеры.

```
// Создаем набор ранее описанных столбцов
CDbColSet* col_set = CDbColSet::NewLC();
col_set->AddL(col_Idx);
```

```

col_set->AddL(col_Name);
col_set->AddL(col_Data);

// Создаем таблицу Test
_LIT(KTestTableName, "Test");
User::LeaveIfError(db.CreateTable(KTestTableName, *col_set));

// Объявим и добавим в набор новый столбец
TDbCol col_New(_L("NewCol"), EDbColBit);
col_set->AddL(col_New);

// Изменим существующую таблицу
User::LeaveIfError(db.AlterTable(KTestTableName, *col_set));

// Удаление набора столбцов
CleanupStack::PopAndDestroy(col_set);

```

Также существует перегруженный метод `CreateTable()`, позволяющий задать первичный ключ таблицы. Для этого в него передается ссылка на объект класса `CDbKey`, содержащий информацию о ключе. Более подробно использование класса `CDbKey` будет рассмотрено позднее, когда мы будем обсуждать работу с индексами.

Получить ссылку на объект `CDbColSet` с информацией о содержащихся в таблице столбцах можно, и не создавая объект `RDbTable`. Для этого необходимо воспользоваться методом `ColSetL()` класса `RDbDatabase`, передав ему в качестве аргумента имя таблицы.

Класс `RDbDatabase` также предоставляет метод `TableNamesL()` для получения списка имен всех существующих в базе данных таблиц. Метод возвращает указатель на объект класса `CDbTableNames`. Класс `CDbTableNames` является псевдонимом для класса `CDbNames`, инкапсулирующего массив дескрипторов `CArrayPakFlat<TDbNameC>`. Он очень прост, поэтому детально рассматривать работу с классом `CDbTableNames` мы не будем. Ограничимся лишь следующим примером, демонстрирующим удаление всех таблиц в базе данных.

```

// Получение списка имен таблиц
CDbTableNames* tables = db.TableNamesL();
if (tables)
{
    CleanupStack::PushL(tables);
    // Удаление всех таблиц
    for (TInt i = 0; i < tables->Count(); i++)
        User::LeaveIfError(db.DropTable((*tables)[i]));
    CleanupStack::PopAndDestroy(tables);
}

```

Для удаления из базы данных таблицы с заданным именем используется метод `DropTable()`.

```

User::LeaveIfError(db.DropTable(KTestTableName));

```

Индексы

Индексы создаются с помощью метода `CreateIndex()`, принимающего в качестве аргументов дескрипторы с именами индекса и связанной с ним таблицы, а также ссылку на объект `CDbKey`, содержащий описание формируемого индекса ключа.

Класс `CDbKey` инкапсулирует массив `CArrayPakFlat<TDbKeyCol>`. Каждый объект `TDbKeyCol` позволяет задать имя добавляемого в ключ столбца таблицы, порядок сортировки по нему, а также размер используемых данных (для столбцов, содержащих строки и бинарные массивы). В момент создания индекса столбец с указанным в `TDbKeyCol` именем должен существовать в таблице. Имя, размер и порядок сортировки (по умолчанию используется значение “по возрастанию”) можно задать в конструкторе класса `TDbKeyCol`.

Работа с массивом объектов `TDbKeyCol` в классе `CDbKey` ведется точно так же, как и с объектами `TDbCol` в `CDbColSet`. Класс `CDbKey` предоставляет методы `AddL()`, `Count()`, `Remove()` и `Clear()`, а также оператор `[]`, полностью соответствующие аналогам из `CDbColSet`. Нумерация объектов в массиве начинается с 1.

Помимо набора столбцов таблицы, ключ содержит информацию о том, является ли он первичным и уникальным, а также позволяет определить метод сравнения строковых значений. Эти данные могут редактироваться следующими методами класса `CDbKey`.

- `IsPrimary()` — проверяет, является ли ключ первичным.
- `MakePrimary()` — устанавливает флаг первичности ключа.
- `IsUnique()` — проверяет, является ли ключ уникальным.
- `MakeUnique()` — устанавливает флаг уникальности ключа.
- `SetComparison()` — устанавливает метод сравнения строк, используемый ключом. В качестве единственного аргумента передается одно из следующих значений перечисления `TDbTextComparison`:
 - `EDbCompareNormal` — стандартное посимвольное сравнение;
 - `EDbCompareFolded` — сравнение дескрипторов с помощью метода `CompareF()`;
 - `EDbCompareCollated` — сравнение дескрипторов с помощью метода `CompareC()`.
- `Comparison()` — возвращает текущий метод сравнения строк.

Пример создания индекса по составному уникальному ключу.

```
// Столбец индекса, сортировка по возрастанию
TDbKeyCol key_col_idx(_L("IdxCol"));
// Столбец со строковыми данными
// Используется не более 10 символов строки,
// сортировка по убыванию
TDbKeyCol key_col_str(_L("StrCol"), 10, TDbKeyCol::EDesc);

// Создание ключа
```

```

CDBKey * key = CDBKey::NewLC();
// Добавление столбцов в ключ
key->AddL(key_col_Idx);
key->AddL(key_col_Str);
// Ключ будет уникальным
key->MakeUnique();
// Создание индекса TestIdx для таблицы Test
_LIT(KTestIndexName, "TestIdx");
User::LeaveIfError(db.CreateIndex(KTestIndexName,
                                KTestTableName, *key));
CleanupStack::PopAndDestroy(key);

```

Получить ссылку на объект CDBKey с информацией о ключе индекса, не создавая объект RDBIndex, можно с помощью метода KeyL() класса RDBDatabase. Для этого ему необходимо передать в качестве аргументов имена таблицы и связанного с ней индекса.

Получить список имен всех существующих в базе данных индексов определенной таблицы можно с помощью метода IndexNamesL() класса RDBDatabase. Он возвращает указатель на объект класса CDBIndexNames. Этот класс, как и класс CDBTableNames, является псевдонимом для класса CDBNames, рассматривавшегося ранее.

Удалить индекс можно с помощью метода DropIndex(), принимающего в качестве аргументов дескрипторы с именами индекса и связанной с ним таблицы.

Чтение и запись данных

Методы для работы с данными, представленными набором строк (rowset), определены в базовом абстрактном классе RDBRawSet. Его конкретными потомками являются класс RDBTable, позволяющий работать со строками таблицы, и класс RDBView, служащий для получения данных из набора строк, возвращаемых в качестве результата SQL-запроса.

Для того чтобы открыть таблицу и получить доступ к ее данным через интерфейс RDBRawSet, необходимо воспользоваться методом Open() класса RDBTable, принимающим в качестве аргументов ссылку на объект базы данных, дескриптор с именем таблицы и режим доступа к ней. В качестве режима доступа используются следующие значения перечисления RDBRowSet::TAccess:

- EReadOnly — только чтение данных, строки не могут изменяться и добавляться;
- EInsertOnly — только добавление новых строк в набор;
- EUpdatable — все операции со строками разрешены (по умолчанию).

Пример открытия таблицы для чтения, записи и изменения данных.

```

RDBTable tbl;
User::LeaveIfError(tbl.Open(db, KTestTableName));
CleanupClosePushL(tbl);

```

```
<...> // Работа с данными таблицы Test
CleanupStack::PopAndDestroy(&tbl);
```

Набор строк таблицы имеет начало, конец, а также курсор, указывающий на текущую строку, с которой ведется работа. Строки в наборе не упорядочены, но они могут быть отсортированы в соответствии с ключом связанного с таблицей индекса. Чтобы выстроить строки таблицы в соответствии с ее индексом, необходимо активировать его, используя метод `SetIndex()` класса `RDbTable`. Курсор в этом случае вновь устанавливается в начало набора. Единственным аргументом метода `SetIndex()` является дескриптор с именем индекса. Отключить упорядочение по индексу можно с помощью метода `SetNoIndex()`. Данные методы не работают для таблиц, открытых в режиме доступа `EInsertOnly`. Пример.

```
// Упорядочить строки таблицы Test по индексу TestIdx
User::LeaveIfError(tbl.SetIndex(KTestIndexName));
// Отключение упорядочения по индексу
User::LeaveIfError(tbl.SetNoIndex());
```

По ключу активного индекса можно выполнять поиск нужной строки с помощью метода `SeekL()`. Для этого ему передается объект класса `TDbSeekKey` или `TDbSeekMultiKey`, содержащий искомые данные, а также метод сравнения. Класс `TDbSeekKey` подходит для поиска значения в простом ключе либо по первому полю составного ключа индекса. Значение передается в конструкторе класса либо устанавливается после создания с помощью метода `AddL()`. Класс `TDbSeekMultiKey` позволяет задать несколько значений для поиска в составном ключе. Очередность типов значений должна совпадать с типами столбцов в ключе. В качестве метода сравнения используются следующие значения перечисления `RDbTable::TComparison`:

- `ELessThan` — переход на последнюю строку, ключ которой строго меньше искомого;
- `ELessThan` — переход на последнюю строку, ключ которой меньше искомого или равен ему;
- `EEqualTo` — переход на первую строку, ключ которой совпадает с искомым;
- `EGreaterEqual` — переход на первую строку, ключ которой совпадает с искомым или больше его;
- `EGreaterThan` — переход на первую строку, ключ которой больше искомого.

Метод `SeekL()` возвращает значение `ETrue` в случае, если удовлетворяющая условиям строка была найдена, и перемещает на нее курсор. Например, для индекса `TestIdx`, содержащего числовой и строковый столбцы, можно выполнить следующий поиск.

```
// Переход на строку, значение столбца IdxCol
// в которой равно 10
TDbSeekKey seek_key(10);
if (tbl.SeekL(seek_key))
```

```

{
    // Строка найдена
}

// Переход на строку, с IdxCol >= 10 и StrCol >= "String"
TDbSeekMultiKey<2> seek_keys;
seek_keys.Add(10);
seek_keys.Add(_L("String"));
if (tbl.SeekL(seek_keys, RDbTable::EGreaterEqual))
{
    // Строка найдена
}

```

Набор строк (даже пустой) всегда имеет две позиции — начало (beginning) и конец (end). Не следует путать их с первой и последней строкой. Сразу после создания курсор указывает на начало.

Получить количество строк в наборе можно с помощью метода `CountL()`. Метод `IsEmptyL()` позволяет выяснить, пуст ли набор (он работает быстрее, чем `CountL() == 0`). Для перемещения курсора по набору строк используются следующие методы.

- `BeginningL()` — перемещает курсор в начало набора строк.
- `AtBeginning()` — позволяет проверить, находится ли курсор в начале.
- `FirstL()` — перемещает курсор на первую строку и возвращает значение `ETrue`. Если строк в наборе нет, то перемещает курсор в конец и возвращает значение `EFalse`.
- `EndL()` — перемещает курсор в конец набора строк.
- `AtEnd()` — позволяет проверить, находится ли курсор в конце.
- `LastL()` — перемещает курсор на последнюю строку и возвращает значение `ETrue`. Если строк в наборе нет, то перемещает курсор на начало и возвращает значение `EFalse`.
- `NextL()` — перемещает курсор на следующую строку в наборе и возвращает значение `ETrue`. Если строк больше нет, то курсор сдвигается на конец набора и результатом метода является `EFalse`. Будучи вызванным в момент, когда курсор указывает на начало, метод `NextL()` работает аналогично методу `FirstL()`.
- `PreviousL()` — перемещает курсор на предыдущую строку в наборе и возвращает значение `ETrue`. Если строк больше нет, то курсор сдвигается на начало набора и результатом метода является `EFalse`. Будучи вызванным в момент, когда курсор указывает на конец, метод `PreviousL()` работает аналогично методу `LastL()`.
- `AtRow()` — проверяет, указывает ли курсор на строку.

Следующий пример демонстрирует некоторые распространенные способы организации циклов, позволяющих перебрать все строки в наборе.

```
tbl.BeginningL();
while (tbl.NextL())
{
}

tbl.BeginningL();
for (tbl.FirstL(); tbl.AtRow(); tbl.NextL())
{
}
```

Возвратиться к уже пройденной курсором позиции можно с помощью закладок. Закладка представляет собой объект типа `TDbBookmark`, содержащий информацию о позиции. Получить такой объект для текущей позиции курсора можно с помощью метода `Bookmark()`, а перейти к строке, на которую он ссылается, с помощью метода `GotoL()`.

```
// Получение закладки для текущей позиции курсора
TDbBookmark bmk = tbl.Bookmark();
tbl.EndL(); // Курсор перемещен в конец набора строк
// Возвращение к позиции bmk
tbl.GotoL(bmk);
```

Считать содержимое текущей строки таблицы можно, предварительно выполнив метод `GetL()`, открывающий запись для чтения. Последующее получение данных из столбцов таблицы выполняется следующими методами.

- `ColSetL()` — возвращает указатель на объект класса `CDbColSet`, содержащий информацию о наборе столбцов, формирующем данную таблицу. Работа с этим классом нами уже рассматривалась. Метод может быть использован для получения номера столбца в таблице по имени. Ответственность за удаление полученного объекта несет разработчик.
- `ColCount()` — возвращает количество столбцов.
- `ColType()` — позволяет установить тип данных (класс `TDbColType`), содержащихся в столбце, по его номеру.
- `ColDef()` — возвращает объект с информацией о столбце (класс `TDbCol`) по его номеру.
- `ColSize()` — размер содержащего в столбце с заданным номером значения (в байтах). Для значения `NULL` возвращается 0.
- `ColLength()` — длина содержащего в столбце с заданным номером значения. Для значения `NULL` возвращается 0. Для чисел и времени результат равен 1. Для текста возвращается количество символов (8- или 16-битовых). Для бинарных массивов — размер в байтах.
- `IsColNull()` — проверяет значение столбца с заданным номером на равенство `NULL`.
- Методы `ColInt()`, `ColInt8()`, `ColInt16()`, `ColInt32()`, `ColInt64()`, `ColUInt()`, `ColUInt8()`, `ColUInt16()`, `ColUInt32()`, `ColReal()`,

`ColReal32()`, `ColReal64()`, `ColTime()`, `ColDes()`, `ColDes16()`, `ColDes8()` — служат для получения значения соответствующего типа из столбца с заданным номером.

Чтение данных из столбцов типа `EDbColBinary` и `EDbColLongBinary` выполняется с помощью класса потока `RDbColReadStream`, объявленного в файле `d32dbms.h`. Следующий пример чтения данных из последней строки таблицы **Test** демонстрирует его использование.

```
//Чтение данных из последней строки таблицы
//Предположим, что нам не известны порядковые номера столбцов
//Выясним их по известным нам именам столбцов
CDBColSet* cs = tbl.ColSetL();
TDbColNo no_idx = cs->ColNo(_L("IdxCol"));
TDbColNo no_str = cs->ColNo(_L("StrCol"));
TDbColNo no_data = cs->ColNo(_L("DataCol"));
//Не используем стек очистки, так как сброс произойти не может
delete cs;

//Попытаемся перевести курсор на последнюю строку
if (tbl.LastL())
{
    // Курсор изменил положение
    tbl.GetL();
    // Открываем строку для получения данных
    TUint idx = tbl.ColUint(no_idx);
    TBuf<20> buf = tbl.ColDes(no_str);
    // Если значение в столбце не равно NULL, считываем его
    if (!tbl.IsColNull(no_data))
    {
        // Чтение объекта TMyStreamData из бинарного массива
        RDbColReadStream db_rs;
        db_rs.OpenLC(tbl, no_data);
        TMyStreamData sample;
        db_rs >> sample;
        CleanupStack::PopAndDestroy();
    }
}
```

Добавить новую строку в набор можно с помощью методов `InsertL()` и `InsertCopyL()`. Курсор при этом блокируется и не может быть перемещен до завершения операции. При использовании метода `InsertL()` значения столбцов новой строки по умолчанию равны `NULL`. В отличие от него, метод `InsertCopyL()` позволяет инициализировать их значениями той строки, на которую указывал курсор. В обоих случаях исключением являются столбцы, тип данных которых поддерживает автоматическое увеличение — они сразу получают новые увеличенные значения. Для всех остальных столбцов значение следует устанавливать самостоятельно с помощью одного из перегруженных

методов `SetColL()`. Метод `SetColNullL()` может быть использован для записи значения `NULL` в столбец с заданным номером.

Запись данных в столбцы типа `EDbColBinary` и `EDbColLongBinary` происходит с помощью класса потока `RDbColWriteStream`, использование которого очень похоже на работу с классом `RStoreWriteStream`.

После того как все необходимые данные были внесены в новую строку, следует вызвать либо метод `PutL()` для вставки строки в таблицу, либо `Cancel()` для отмены операции. Во время работы метода `PutL()` проверяется соответствие значений типам столбцов (`NULL`, область значений, размер) и при необходимости проверка их уникальности. Затем выполняется запись данных и обновление связанных индексов. Если в этот момент ни один индекс таблицы не является активным, то строка вставляется в конец набора. В противном случае ее положение в наборе определяется активным индексом. Курсор автоматически перемещается на новую строку.

Следующий пример демонстрирует вставку двух строк в таблицу `Test`.

```
tbl.InsertL();
tbl.SetColL(no_str, _L("New String Value"));
// Значение IdxCol не изменяем, так как оно увеличивается
// автоматически
// Значение DataCol может содержать NULL, так и оставим
// Вставка новой строки
tbl.PutL();

// Курсор указывает на новую строку, создадим одну
// на ее основе
tbl.InsertCopyL();
// Значение StrCol скопировано и равно "New String Value"
// Значение IdxCol не изменяем
// Запишем объект TMyStreamData в бинарный массив
TMyStreamData sample;
sample.iDes = _L("Sample");
sample.iInt32 = 0x10;
sample.iReal64 = 3.14;

RDbColWriteStream dbws;
dbws.OpenLC(tbl, no_data);
dbws << sample;
dbws.CommitL();
CleanupStack::PopAndDestroy(&dbws);
tbl.PutL();
tbl.PutL();
```

Для изменения данных в уже существующей строке таблицы служит метод `UpdateL()`. После его вызова строка, на которую указывает курсор, открывается для редактирования. В дальнейшем работа ведется точно так же, как и при вставке новой строки. Изменения должны быть подтверждены с помощью метода `PutL()` или отменены вызовом метода `Cancel()`.

Удаление текущей строки из таблицы осуществляется методом `DeleteL()`.

```
// Удаление последней строки набора (если есть)
if (tbl.LastL())
    tbl.DeleteL();
```

При этом строка физически удаляется, но положение курсора не меняется. Если сдвинуть курсор на другую позицию (например, на следующую строку), то “призрак” удаленной строки пропадет из набора.

SQL-запросы

СУБД Symbian OS поддерживает выполнение SQL-запросов (DDL и DML). Изучение языка SQL выходит за рамки этой книги. Поэтому будем исходить из того, что его основы вам знакомы.

Еще одним проблемным вопросом является полнота поддержки возможностей языка SQL. Не приходится говорить о соответствии стандартной СУБД Symbian OS каким-либо стандартам ANSI SQL. Например, не поддерживаются процедуры, триггеры, операторы JOIN, UNION, DISTINCT, операции с несколькими таблицами (после ключевого слова FROM должна быть только одна таблица) и многое другое. В справочнике SDK есть специальный раздел, посвященный этой теме. Все это зачастую вынуждает разработчиков использовать другие поддерживаемые Symbian OS СУБД: SQLite (Symbian SQL) и продукты Oracle.

Таким образом, использование языка SQL — далеко не единственный способ работы с базой данных в СУБД Symbian. Тем не менее использование SQL-запросов в ряде случаев может быть очень удобно.

Текст SQL-запроса хранится в дескрипторе, как и любая другая строка. Иногда его помещают в объект класса `TDbQuery`, инкапсулирующий сам текст запроса и значение перечисления `TDbTextComparison`, который мы уже рассматривали при работе с классом `CDbKey`.

Для выполнения операций подмножества DDL (создание, изменение и удаление таблиц и индексов) и (частично) DML (изменение и удаление данных из таблицы) следует использовать метод `Execute()` класса `RDbDatabase`. В него передается дескриптор с текстом SQL-выражения и необязательный параметр класса `TDbTextComparison`. Результатом выполнения метода `Execute()` является код ошибки для DDL-операций или количество обработанных строк для DML-операций. В приведенном ниже примере создаются таблица **Test** и индекс **TestIdx**, практически аналогичные рассматриваемым ранее. Сведения о соответствии типов данных языка SQL и СУБД Symbian представлены в табл. 6.3.

```
// Создание таблицы Test
_LIT(KCreateTable, "CREATE TABLE Test (" \
    " IdxCol counter not null," \
    " StrCol varchar(20)," \
    " DataCol long varbinary," \
    " NewCol bit)");
```

```
User::LeaveIfError(db.Execute(KCreateTable));

// Создание индекса TestIdx
_LIT(KCreateIndex, "CREATE UNIQUE INDEX TestIdx on Test (" \
    " IdxCol ASC, " \
    " StrCol DESC)");
User::LeaveIfError(db.Execute(KCreateIndex));
```

Таблица 6.3. Соответствие типов данных языка SQL и СУБД Symbian OS

Тип SQL	Тип в СУБД Symbian
BIT	EDbColBit
TINYINT	EDbColInt8
UNSIGNED TINYINT	EDbColUInt8
SMALLINT	EDbColInt16
UNSIGNED SMALLINT	EDbColUInt16
INTEGER	EDbColInt32
UNSIGNED INTEGER	EDbColUInt32
COUNTER	EDbColUInt32 с атрибутом TDbCol::EAutoIncrement
BIGINT	EDbColInt64
REAL	EDbColReal32
FLOAT	EDbColReal64
DOUBLE	EDbColReal64
DOUBLE PRECISION	EDbColReal64
DATE	EDbColDateTime
TIME	EDbColDateTime
TIMESTAMP	EDbColDateTime
CHAR(<i>n</i>)	EDbColText, где <i>n</i> соответствует максимальной длине
VARCHAR(<i>n</i>)	EDbColText, где <i>n</i> соответствует максимальной длине
LONG VARCHAR	EDbColLongText
BINARY(<i>n</i>)	EDbColBinary, где <i>n</i> соответствует максимальной длине
VARBINARY(<i>n</i>)	EDbColBinary, где <i>n</i> соответствует максимальной длине
LONG VARBINARY	EDbColLongBinary
DECIMAL(<i>p</i> , <i>s</i>)	Не поддерживается
NUMERIC(<i>p</i> , <i>s</i>)	Не поддерживается

Выполнение обновления и удаления (UPDATE и DELETE) данных таблицы с помощью SQL-запросов может быть даже более эффективным, чем реализация этих операций на C++. Однако работа SQL-выражения INSERT существенно уступает в производительности аналогичному по функциональности коду на C++.

Для получения данных с помощью запроса SELECT используются SQL-представления (SQL views), реализуемые классом RDbView. Класс представления, также как и класс RDbTable, наследован от класса TDbRowSet, поэтому работа с данными в них аналогична.

Процесс получения новой строки или нескольких строк результата SQL-запроса называется **определением** (evaluation). Существует три типа представлений.

- **Без предварительного определения** (without pre-evaluation) — новые строки представления определяются автоматически “на лету” при перемещении курсора (при вызове метода `NextL()` на последней известной строке). Все полученные ранее строки хранятся в представлении. Если при очередном определении подходящей строки SQL-запрос ее не обнаружил, то курсор переходит в конец набора.
- **С полным предварительным определением** (with full pre-evaluation) — в этом случае разработчик должен самостоятельно вызывать метод `Evaluate()`, с помощью которого определяется следующий блок строк результата SQL-запроса. Количество получаемых одновременно строк устанавливает СУБД, обычно оно довольно велико. Как и в предыдущем варианте, все полученные ранее строки хранятся в представлении.
- **С частичным предварительным определением** (with limited pre-evaluation) — компромисс между двумя вышеназванными типами. Для получения очередного блока строк результата SQL-запроса необходимо использовать метод `Evaluate()`. При этом используется так называемое **окно определения** (evaluation window), позволяющее задать количество строк до и после текущей, которые должны быть определены. Представление не хранит все ранее полученные строки, а лишь те, что были возвращены при последней оценке.

Перед началом работы представление должно быть подготовлено с помощью одного из перегруженных методов `Prepare()`. Его аргументами являются ссылка на базу данных и инкапсулирующий SQL-запрос объект `TDbQuery`. Разработчик также может указать окно определения и режим доступа к содержимому представления (по умолчанию — `RDbRowSet::EUpdatable`).

Оценочное окно задается с помощью класса `TDbWindow`. Этот класс включает размер окна и позицию текущей строки в нем. Для подготовки представления с полным предварительным определением окно создается следующим образом.

```
TDbWindow window(TDbWindow::EUnlimited);
```

Можно также использовать константу `KDbUnlimitedWindow`, определяющую этот объект.

Для подготовки представления с частичным предварительным определением при создании окна необходимо указать количество строк до и после текущей, которые будут получаться при вызове метода `Evaluate()`.

```
// Окно с 5 строками до текущей и 3 после. Всего 9 строк
TDbWindow window(5, 3);
```

При подготовке представления без предварительного определения окно не создается вообще.

Класс `RDbView` также предоставляет следующие методы.

- `Unevaluated()` — возвращает `ETrue`, если еще не все строки результата определены и переданы представлению.
- `Evaluate()` — определение и получение следующего блока строк результата. Возвращает код ошибки. В представлении с полным предварительным определением также возвращает значение `>0`, если это не последний блок данных, который можно получить с помощью метода `Evaluate()`. Существует также асинхронный вариант метода.
- `EvaluateAll()` — позволяет произвести все оставшиеся определения. Эквивалент кода: `while (Unevaluated()) { Evaluate(); }`. Может использоваться только в представлениях с полным предварительным определением.

Чтобы окончательно разобраться с тем, как ведут себя три вышеназванных представления, мы рассмотрим небольшой пример. Поместим в таблицу **Test** 1000 строк.

```
for (TInt i = 0; i < 1000; i++)
{
    // В таблицу записывается новая строка,
    // содержащая лишь одно непустое поле - IdxCol
    tbl.InsertL();
    tbl.PutL();
}
```

Затем создадим представление на основе SQL-запроса “SELECT * FROM Test”, результатом которого являются 1000 строк. Рассмотрим, как выполняется получение этих строк и навигация по ним в представлении без использования предварительного определения.

```
RDbView view;
_LIT(KSQLQuery, "SELECT * FROM Test");
// Подготовка представления
User::LeaveIfError(view.Prepare(db, TDbQuery(KSQLQuery)));
CleanupClosePushL(view);

// Прямой проход
while (view.NextL())
{
    // Чтение строки
}

// Обратный проход (курсор в конце набора)
while (view.PreviousL())
{
    // Чтение строки
}

CleanupStack::PopAndDestroy(&view);
```

Код в обоих циклах `while` выполняется по 1000 раз. Методы `Evaluate()` не используются.

Теперь рассмотрим создание аналогичного представления с полным предварительным определением.

```

RDbView view;
_LIT(KSQLQuery, "SELECT * FROM Test");

User::LeaveIfError(view.Prepare(db, TDbQuery(KSQLQuery),
    KDbUnlimitedWindow));
// Вместо KDbUnlimitedWindow, можно было бы
// TDbWindow(TDbWindow::EUnlimited)
CleanupClosePushL(view);

// Пока все строки результата не будут определены
while (view.Unevaluated())
{
    // Определение блока строк
    view.Evaluate();

    if (view.AtBeginning())
        view.FirstL();

    do
    {
        // Чтение строки
    }
    while (view.NextL());
}

// Обратный проход (курсор в конце набора)
while (view.PreviousL())
{
    // Чтение строки
}

CleanupStack::PopAndDestroy(&view);

```

Заметьте, как изменился код чтения строк. Сразу после подготовки представление с полным предварительным определением пусто, поэтому необходимо вызывать метод `Evaluate()`, определяющий следующий блок кода. Размер этого блока выбирается СУБД. В моем эмуляторе он оказался равен 257 строкам. Таким образом, цикл `while` был вызван 4 раза. В условии `while` используется метод `Unevaluated()` вместо `Evaluate()`, так как последний (т.е. `Evaluate()`) вернул бы 0. При создании представления курсор указывает на начало набора. После вызова первого метода `Evaluate()` его местоположение не меняется. После того как курсор проходит первый полученный блок строк и достигает

конца набора, мы вновь вызываем метод `Evaluate()` (рис. 6.9). В этот момент курсор не может быть ни оставлен в конце набора строк, ни перемещен в его начало, так как разработчику пришлось бы использовать закладки, чтобы вернуться к началу второго блока строк. Поэтому при повторном вызове `Evaluate()` курсор помещается на первую строку нового блока результатов. Чтобы не пропустить первую строку второго блока, мы используем в качестве внутреннего цикла `do`, а не `while`. Поэтому необходима вставка с вызовом метода `FirstL()` в первом блоке, чтобы избежать чтения данных, когда курсор указывает на начало набора, а не на строку.

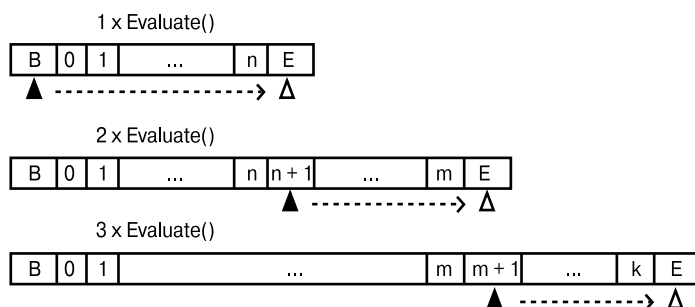


Рис. 6.9. Действие метода `Evaluate()` в представлении с полным предварительным определением

При обратном проходе мы считаем все 1000 строк, так как все они остаются в памяти.

Мы могли бы использовать и циклы из примера работы с представлением без использования предварительного определения. Для этого перед циклом чтения всего лишь необходимо вызвать метод `EvaluateAll()` и определить все строки результата сразу, но в ряде случаев это неэффективно. Например, если результаты SQL-запроса выводятся пользователю постранично, то, вполне вероятно, что он ограничится просмотром части страниц, и загружать в память все результаты бессмысленно.

Создать представление с частичным предварительным определением можно следующим образом.

```
User::LeaveIfError(view.Prepare(db,
    TDbQuery(KSQLQuery), TDbWindow(5, 4)));
```

В данном примере окно определения содержит 10 строк: 5 перед текущей и 4 после нее. Сразу после создания курсор указывает на начало набора, а само представление пусто. Получать новые блоки данных необходимо с помощью метода `Evaluate()`. Цикл, выполняющий чтение всех данных, можно оставить без изменений. Исполняться он будет следующим образом.

После первого вызова метода `Evaluate()` в представление будет передано 10 строк, и вложенный цикл `do` пройдет их с первой по десятую (рис. 6.10). При последующих вызовах `Evaluate()` также будет оцениваться по 10 строк, но 4 из них будут уже пройденными в предыдущем блоке, а курсор будет указывать

на 5-ю. Кроме того, полученный блок данных заменит имеющиеся в представлении строки. Таким образом, в памяти всегда находится только 10 строк.

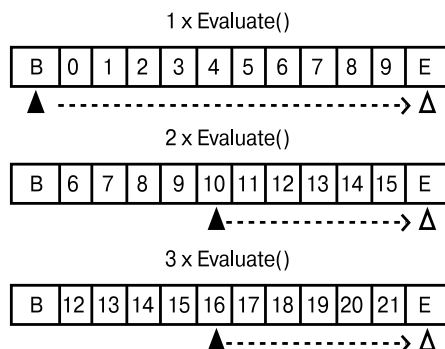


Рис. 6.10. Действие метода `Evaluate()` в представлении с частичным предварительным определением

Совершить обратный проход по всем записям результата можно с помощью следующего цикла.

```
do
{
    // Определение блока строк
    view.Evaluate();

    if (view.AtEnd())
        view.PreviousL();
    do
    {
        // Чтение строки
    }
    while (view.PreviousL());
}
while (view.Unevaluated());
```

Использование сессии сервера СУБД

Как уже отмечалось ранее, для разделяемого доступа, при котором хотя бы один клиент осуществляет запись, необходимо использовать клиент-серверный метод работы с базой данных. Для этого в классе `RDbNamedDatabase` существуют перегруженные методы `Create()` и `Open()`, принимающие вместо сессии файлового сервера `RFs` сессию сервера СУБД `RDBs`.

Перегруженный метод `Create()` также отличается аргументом, содержащим дескриптор для передачи параметров БД. Он обязательный и имеет вид `"SECURE[UID]"`, а не `"[UID]"`. Передаваемый с его помощью `UID` является идентификатором политики безопасности (security policy `UID`). Политика

безопасности задается объектом `TSecurityPolicy` и хранится в файле с именем `<UID>.spd` в подкаталоге `\policy\` приватного каталога сервера СУБД. С ее помощью задаются списки защищенных возможностей, доступ к которым должен декларировать клиент при чтении, записи и изменении схемы базы данных или таблицы. Политика безопасности также может ограничить доступ по `SID` или `VID` клиента. Список доступных на устройстве идентификаторов политики безопасности не документирован, а возможность задания собственной политики не предусмотрена. Поэтому мы будем исходить из того, что разработчик не может создать базу данных, используя сессию сервера СУБД, но может открыть уже существующую (в методе `Open()` строка параметров не является обязательным аргументом). Мы также не рассматриваем множество методов класса `RDBs` для работы с политикой безопасности.

Итак, для того чтобы открыть созданную нами ранее базу данных для разделяемого доступа с возможностью записи, все клиенты должны использовать следующий код.

```
RFs fs;
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);
// Имя базы данных
_LIT(KMydatabase, "c:\\mydb2.db");

// Установка сессии с сервером СУБД
RDBs dbs;
User::LeaveIfError(dbs.Connect());
CleanupClosePushL(dbs);

// Открытие базы данных
RDbNamedDatabase db;
User::LeaveIfError(db.Open(dbs, KMydatabase));
CleanupClosePushL(db);
<...> // Работа с базой данных
CleanupStack::PopAndDestroy(3, &fs);
```

Далее работа с базой данных ведется абсолютно так же, как и без использования сервера СУБД.

Транзакции

СУБД Symbian поддерживает два вида блокировок, на основе которых реализованы транзакции: **блокировки для чтения** (`read-lock`) и **блокировки для записи** (`write-lock`). Объектом блокировки может быть только база данных целиком. Блокировка для записи может быть установлена только в том случае, если база данных не заблокирована. Блокировка для чтения может быть установлена во всех случаях, кроме ситуации, когда база данных уже заблокирована для записи. Поэтому база данных может иметь сколько угодно блокировок для

чтения, запрещающих запись данных, либо только одну блокировку для записи, запрещающую любые другие блокировки.

Начать новую транзакцию можно с помощью метода `Begin()` класса `RDbDatabase`. При этом устанавливается блокировка для чтения. Если в рамках транзакции используются методы редактирования данных таблицы (`InsertL()`, `InsertCopyL()` и `UpdateL()`), то уровень блокировки повышается до блокировки для записи. После вызова метода `PutL()` или `Cancel()` она вновь заменяется на блокировку для чтения. Транзакция может быть либо подтверждена с помощью метода `Commit()`, либо отменена с помощью метода `Rollback()` класса `RDbDatabase`. Блокировка для чтения после этого снимается.

Если методы редактирования данных таблицы вызываются вне транзакции, то они создают свою собственную. Она будет автоматически закрыта при вызове метода `PutL()` или `Cancel()`. Таким образом, во время добавления или изменения данных всегда устанавливается блокировка для записи. То же происходит при выполнении SQL-запросов подмножества DML.

Несколько операций по добавлению или изменению данных в рамках одной транзакции выполняются быстрее, чем если бы для каждой из них транзакция создавалась бы автоматически. Поэтому код, использовавшийся нами ранее:

```
for (TInt i = 0; i < 1000; i++)
{
    // В таблицу записывается новая строка,
    // содержащая лишь одно непустое поле - IdxCol
    tbl.InsertL();
    tbl.PutL();
}
```

будет выполняться значительно быстрее, будучи заключен в одну транзакцию.

```
// Начало транзакции
User::LeaveIfError(db.Begin());

for (TInt i = 0; i < 1000; i++)
{
    tbl.InsertL();
    tbl.PutL();
}

// Подтверждение транзакции
db.Commit();
```

Клиент не может использовать вложенные транзакции — это приведет к панике. Если клиент считывает данные вне транзакции (а значит, не используя блокировку чтения) в тот момент, когда другой клиент осуществляет их запись, то он может получить неверную, частично измененную информацию.

Поэтапное выполнение операций

API СУБД Symbian предлагает два класса для поэтапного выполнения операций: `RDbIncremental` и `RDbUpdate`. Их можно использовать в том случае, если во время работы приложение должно отображать информацию о ходе выполнения операции или реагировать на пользовательский ввод. Класс `RDbIncremental` предоставляет методы `AlterTable()`, `CreateIndex()`, `DropIndex()`, `DropTable()`, `Execute()`, `Compact()`, `Recover()` и `UpdateStats()`, аналогичные соответствующим методам класса `RDbDatabase`. Отличие в их использовании заключается в том, что все эти методы имеют дополнительный аргумент — ссылку на числовую переменную. В эту переменную после вызова метода помещается начальное количество шагов, на которые разбита операция. Далее необходимо передавать эту переменную в метод `Next()`, выполняющий один шаг операции. Количество шагов при этом может уменьшаться. Операция считается завершенной, когда значение переменной станет равным нулю. Пример.

```
RDbIncremental incr;
TInt steps;
User::LeaveIfError(incr.DropTable(db, KTestTableName, steps));
CleanupClosePushL(incr);
if (steps > 0)
    do
    {
        // Отображение хода выполнения операции
        User::LeaveIfError(incr.Next(steps));
    } while(steps > 0);
CleanupStack::PopAndDestroy(&incr);
```

Существует также асинхронный вариант метода `Next()`.

Метод `Execute()` позволяет выполнять только DDL-запросы. Метод `Close()` класса `RDbIncremental` освобождает занимаемые объектом ресурсы и откатывает состояние базы данных в случае, если текущая операция не была выполнена до конца.

Класс `RDbUpdate` предоставляет возможность поэтапного выполнения SQL-запросов с оператором `UPDATE`. Работа с ним ведется точно так же, как и с классом `RDbIncremental`. Класс `RDbUpdate` имеет один дополнительный метод — `RowCount()`, возвращающий количество строк, измененных в результате выполнения операции. Его можно вызывать между этапами.

Сокеты

Сокет (socket) — это абстрактный объект, представляющий собой конечную точку межпроцессного соединения. Стандартный интерфейс сокета был разработан в институте Беркли еще в 1983 году. С помощью сокетов можно единообразно осуществлять передачу данных, используя различные протоколы и ка-

налы связи. Например, на базе сокетов реализуются службы Bluetooth (протоколы RFCOMM, L2CAP, AVCTP, ACDTP, Link Manager), передача данных через инфракрасный порт (протоколы Tiny TP, IrMUX), отправка SMS-сообщений, а также интернет-коммуникации (протоколы TCP, UDP, ICMP и др.).

Сервер сокетов

Работу с сокетами в Symbian OS реализует сервер ESock (файл `esock.exe`). Он поддерживает различные протоколы передачи данных, содержащиеся в библиотеках PRT. Сервер загружает и выгружает данные библиотеки по мере необходимости. Для работы с новым протоколом создается отдельный поток.

Все используемые для работы с сервером ESock классы объявлены в заголовочном файле `es_sock.h` и требуют подключения к проекту библиотеки импорта `esock.lib`. Сессия с сервером ESock устанавливается с помощью класса `RSocketServ`, предоставляющего следующие методы.

- `Connect()` — установление сессии с сервером. Имеет необязательный параметр, позволяющий зарезервировать необходимое количество слотов сообщений.
- `NumProtocols()` — позволяет узнать количество протоколов, которые в данный момент известны серверу.
- `GetProtocolInfo()` и `FindProtocol()` — позволяют получить информацию о протоколе по его индексу или имени. Информация передается в структуре `TProtocolDesc`.
- `StartProtocol()` и `StopProtocol()` — асинхронные методы, указывающие серверу на необходимость загрузить или выгрузить определенный протокол. Использование этих методов требует доступа к защищенной возможности `NetworkControl`. Аргументами являются идентификаторы семейства, типа сокета и протокола в семействе.
- `InstallExtension()` — устанавливает новую библиотеку расширения в сервер ESock. Применяется редко. Использование требует доступа к защищенной возможности `NetworkControl`.
- `SetExclusiveMode()` и `ClearExclusiveMode()` — асинхронные методы, позволяющие получить эксклюзивный доступ к серверу или отказаться от него. Применяются редко. Использование требует доступа к защищенной возможности `NetworkControl`.
- `Version()` — возвращает версию сервера.

Обычно сервер самостоятельно загружает нужный протокол при создании использующего его сокета. Кроме того, многие часто применяемые протоколы загружаются при обращении использующих их системных сервисов и готовы к использованию практически постоянно. Тем не менее для ряда приложений время открытия сокета может быть критично и имеет смысл убедиться в том, что оно не будет потрачено на инициализацию протокола, если предварительно загрузить его с помощью метода `StartProtocol()`.

Одну сессию с сервером могут разделять сокет, использующие различные протоколы. Так как все загруженные протоколы исполняются в разных потоках, то сервер тратит некоторое время на определение нужного потока. Если поток, с которым преимущественно будет использоваться данная сессия, известен заранее, то об этом можно сообщить серверу `ESock`. Для этого существует перегруженный вариант метода `Connect()`, принимающий объект класса `TSessionPref`. Данный класс инкапсулирует идентификаторы семейства протоколов и самого протокола.

Протоколы

Структура `TProtocolDesc` содержит различную информацию о протоколе. Ниже перечислены основные характеристики протокола, представленные в этой структуре.

- `iAddrFamily` — идентификатор семейства протоколов. Чаще всего означает формат записи адресов объектов коммуникации. Например, для интернет-протоколов это совокупность IP-адреса и порта.
- `iProtocol` — идентификатор протокола в семействе.
- `iName` — символьное имя протокола.
- `iSockType` — тип используемого с данным протоколом сокета. Может принимать одно из следующих значений:
 - `KReadStream` — сокет как **поток данных**; означает, что этот сокет представляет устойчивый канал для надежной передачи блоков данных произвольного размера;
 - `KSockDatagram` — сокет **для передачи датаграмм**; данные передаются пакетами ограниченного размера, и если размер пакета превышает допустимый, то он либо отбрасывается, либо считывается частями (поведение зависит от протокола), а протоколы, использующие такой тип сокета, могут реализовывать как надежную, так и ненадежную (см. `iServiceInfo`) передачу данных;
 - `KSockSeqPacket` — сокет **для передачи последовательности датаграмм**; протоколы, использующие данный тип сокета, осуществляют надежную передачу данных пакетами ограниченного размера.
 - `KSockRaw` — сокет **предоставляет низкоуровневый доступ** для чтения и записи пакетов протокола.
- `iServiceInfo` — содержит беззнаковое число, представляющее собой битовую комбинацию различных характеристик протокола. Наиболее важными из них являются надежность (присутствие `KSIReliable`) и использование соединений (присутствие `KSIConnectionLess`).

Надежные (reliable) протоколы предоставляют гарантии доставки данных в правильном порядке, без потерь и дублирований, а также способны определять и исправлять (по возможности) возникающие при передаче ошибки. Связь,

осуществляемая по **ненадежному** (unreliable) протоколу, частично или полностью не соответствует таким характеристикам.

Протоколы, **ориентированные на использование соединений** (connection-oriented), перед началом передачи данных требуют установки абстрактной сессии с принимающим сокетом. При этом происходит обмен различной служебной информацией для определения общих параметров передачи данных, а также оповещения ее начале и окончании. В отличие от них, **не использующие соединения** (connectionless) протоколы не предполагают организации такой сессии с принимающей стороной.

Следующий код позволяет вывести на экран консольного приложения основную информацию обо всех известных серверу ESocket протоколах. Результаты для наиболее часто используемых из них представлены в табл. 6.4.

```
// Вспомогательная функция
const TDesc& GetSocketType(const TUint aSockType);

void MainL()
{
    //Установка соединения с сервером ESocket
    RSocketServ sock_serv;
    User::LeaveIfError(sock_serv.Connect());
    CleanupClosePushL(sock_serv);

    // Получение количества установленных протоколов
    TUint num(0);
    User::LeaveIfError(sock_serv.NumProtocols(num));

    // Получение информации о протоколах по индексу
    for (TUint i = 1; i <= num; i++)
    {
        TProtocolDesc info;
        User::LeaveIfError(sock_serv.GetProtocolInfo(i, info));

        //Отображение информации
        _LIT(KDescName, "Name: %S\n");
        _LIT(KDescFamily, "Family ID: 0x%x\n");
        _LIT(KDescId, "Protocol ID: 0x%x\n");
        _LIT(KDescSocket, "Socket type: %S\n");
        _LIT(KDescReliable, "Reliable\n");
        _LIT(KDescUnreliable, "Unreliable\n");
        _LIT(KDescConnectionless, "Connectionless\n\n");
        _LIT(KDescConnectionOr, "Connection-oriented\n\n");

        TBuf<50> buf;

        // Символьное имя
        buf.Format(KDescName, &info.iName);
        console->Write(buf);
    }
}
```

```

// ID семейства
buf.Format(KDescFamily, info.iAddrFamily);
console->Write(buf);

// ID протокола в семействе
buf.Format(KDescId, info.iProtocol);
console->Write(buf);

// Тип сокета
buf.Format(KDescSocket, &GetSocketType(info.iSockType));
console->Write(buf);

// Является ли протокол надежным
if (info.iServiceInfo & KSIReliable)
    console->Write(KDescReliable);
else
    console->Write(KDescUnreliable);

// Устанавливает ли протокол соединения
if (info.iServiceInfo & KSIConnectionLess)
    console->Write(KDescConnectionless);
else
    console->Write(KDescConnectionOr);

console->Getch();
}

//Заккрытие сессии с сервером ESock
CleanupStack::PopAndDestroy(&sock_serv);
}

const TDesC& GetSocketType(const TUint aSockType)
{
    // Вспомогательная функция, возвращающая
    // Дескриптор с типом сокета по значению
    _LIT(KSockStreamDes, "stream");
    _LIT(KSockDatagramDes, "datagram");
    _LIT(KSockSeqPacketDes, "seq packet");
    _LIT(KSockRawDes, "raw");
    _LIT(KSockNotApplicable, "not applicable");

    switch (aSockType)
    {
        case KSockStream:
            return KSockStreamDes;
        case KSockDatagram:
            return KSockDatagramDes;
        case KSockSeqPacket:

```

```

        return KSockSeqPacketDes;
    case KSockRaw:
        return KSockRawDes;
    case KUndefinedSockType:
    default:
        return KSockNotApplicable;
    }
}

```

Таблица 6.4. Характеристики наиболее часто используемых протоколов

Семейство (ID)	Протокол	Имя (ID)	Тип сокета	Надежность	Использование соединения
IPv4 (KAfInet) и IPv6 (KAfInet6)	TCP	tcp (KProtocol-InetTcp)	Поток	Да	Да
	UDP	udp (KProtocol-InetUdp)	Датаграмма	Нет	Нет
	ICMP	icmp (KProtocol-InetIcmp) icmp6 (KProtocol-Inet6Icmp)	Датаграмма	Нет	Нет
Bluetooth (KBTAddr-Family)	L2CAP	L2CAP (KL2CAP)	Последовательность датаграмм	Да	Да
	SDP	SDP (KSDP)	Поток	Нет	Да
	AVCTP	AVCTP (KAVCTP)	Датаграмма	Да	Нет
	AVDTP	AVDTP (KAVDTP)	Датаграмма	Нет	Да
	RFCOMM	RFCOMM (KRFCOMM)	Поток	Да	Да
	Link Manager	BTLinkManager (KBTLink-Manager)	-	Нет	Да
IrDA (KIrdaAddr-Family)	Tiny TP	IrTinyTP (KIrTinyTP)	Последовательность датаграмм	Да	Да
	IrMUX	Irmux (Kirmux)	Датаграмма	Да	Да

Указанные в скобках константы семейства Bluetooth-протоколов объявлены в заголовочном файле `bt_sock.h`, интернет-протоколов — в `in_sock.h`, протоколов инфракрасного порта — в `ir_sock.h`.

Более подробную информацию о прочих характеристиках протоколов, предоставляемых классом `TProtocolDesc`, можно получить в справочнике SDK.

Адреса, класс `RHostResolver`

При передаче данных и установке соединений необходимо знать адреса сторон коммуникации. В различных протоколах адресация выполняется по-разному. Базовым классом, представляющим адрес сокета, является `TSockAddr`. В нем хранятся лишь идентификатор семейства протоколов и номер порта. Именно этот класс используется в качестве аргумента различных методов, требующих указания адреса. На практике, в зависимости от применяемого протокола или реализуемой им службы, используются классы, порожденные от `TSockAddr`. Например, `TBTSockAddr` и `TInquirySockAddr` для Bluetooth, `TIrdaAddr` для связи через инфракрасный порт и `TInetAddr` для интернет-протоколов. Они могут передаваться в любой метод, принимающий объект `TSockAddr`, и содержат дополнительную информацию об адресе, специфичную для используемого протокола передачи данных.

Для использования служб разрешения имен (например, DNS для интернет-соединений) или поиска адресов доступных для коммуникации устройств (в Bluetooth и IrDa) служит класс `RHostResolver`. Он содержит базовые методы, принимающие объекты класса `TSockAddr`. Не все протоколы поддерживают предоставляемую ими функциональность. В случае если метод не может быть использован для данного вида соединений, он возвращает код ошибки `KErrNotSupported`.

Информацию о найденных объектах класс `RHostResolver` возвращает в объекте класса `TNameEntry`, который является запакованным в дескриптор объектом `TNameRecord`. В объекте `TNameRecord` инкапсулировано имя объекта, его адрес (`TSockAddr`) и некоторые флаги.

Мы не будем подробно рассматривать все предоставляемые классом `RHostResolver` методы и их работу с различными протоколами, а лишь ограничимся примерами его наиболее частого применения. Следующий код демонстрирует работу DNS при использовании интернет-протоколов.

```
// Работа с протоколом TCP
// Для работы с TInetAddr требуется подключение insock.lib
// Константы KAFInet и KProtocolInetTcp объявлены в in_sock.h

RHostResolver resolver;
User::LeaveIfError(resolver.Open(sock_serv, KAFInet,
    KProtocolInetTcp));
CleanupClosePushL(resolver);
```

```

TBuf<0x100> buf;
// Получение имени текущего устройства
User::LeaveIfError(resolver.GetHostName(buf));
// buf = "localhost"

TNameEntry name;
// Получение адреса по имени сайта
User::LeaveIfError(resolver.GetByName(_L("devmobile.ru"),
                                     name));

TInetAddr addr(name().iAddr);
addr.Output(buf); // buf = "81.176.226.188"

// Получение имени сайта по адресу
addr.Input(_L("77.88.21.8"));
User::LeaveIfError(resolver.GetByAddress(addr, name));
buf = name().iName; // buf = "ya.ru"

CleanupStack::PopAndDestroy(&resolver);

```

Следующий пример позволяет обнаружить и вывести на экран консоли имена всех доступных Bluetooth-устройств.

```

// Работа с протоколом Bluetooth Link Manager
// Для работы с TInquirySockAddr требуется подключение
// bluetooth.lib
// Константы KBTAddrFamily и KBTLinkManager объявлены
// в bt_sock.h

RHostResolver resolver;
// Требуется доступ к защищенной возможности LocalServices
// Bluetooth должен быть включен на устройстве,
// иначе вернет ошибку KErrHardwareNotAvailable
TInt err = resolver.Open(sock_serv, KBTAddrFamily,
                        KBTLinkManager);

User::LeaveIfError(err);
CleanupClosePushL(resolver);

TBuf<0x100> buf;
// Получение имени текущего устройства
User::LeaveIfError(resolver.GetHostName(buf));
// buf = "Nokia 5800 XpressMusic"

//Получение имен всех доступных Bluetooth-устройств
TInquirySockAddr addr;
addr.SetIAC(KGIAC); // General Unlimited Inquiry Access Code
addr.SetAction(KHostResInquiry | KHostResName );

TNameEntry entry;

```

```
while (KErrNone == resolver.GetByAddress(addr, entry))
{
    console->Write(entry().iName);
}

CleanupStack::PopAndDestroy(&resolver);
```

Работа с сокетами

Открытие сокета, установка соединения

Создание сокетов и работа с ними ведутся с помощью объектов класса `RSocket`, определенного в заголовочном файле `es_sock.h`. Для того чтобы создать сокет на стороне сервера и получить доступ к нему, необходимо воспользоваться одним из многочисленных перегруженных методов `Open()` класса `RSocket`. Класс `RSocket` является субсессией от `RSocketServ`, и поэтому одним из аргументов метода `Open()` является ссылка на установленную сессию с сервером сокетов. Помимо этого, в качестве аргументов может указываться идентификатор типа сокета (поток данных, передача датаграмм и пр.) и протокол (по имени либо по идентификаторам семейства и протокола в нем). Следующий пример демонстрирует создание и открытие сокета с протоколом TCP.

```
RSocket sock;
User::LeaveIfError(sock.Open(sock_serv, KAF_INET,
    KSOCK_STREAM, KPROTOCOL_INET_TCP));
/* Либо так:
 * _LIT(KTCPProt, "tcp");
 * User::LeaveIfError(sock.Open(sock_serv, KTCPProt));
 */
```

В зависимости от того, требует ли установки соединения применяемый протокол, сокеты также делятся на две группы: **ориентированные на использование соединений** (connection-oriented) и **не использующие соединения** (connectionless). Тип сокетов, позволяющий передавать данные как потоки, обычно соответствует первой группе, а типы сокетов, передающие информацию в форме датаграмм, чаще всего применяются сокетами второй группы. Данные характеристики в большой степени определяют способ работы с сокетом.

После открытия сокета он может использоваться либо для установки соединения (в этом случае сокет называют клиентским, а соединение активным), либо для ожидания входящих соединений (сокет называют серверным, а соединение пассивным).

Для установки активного соединения необходимо воспользоваться одним из асинхронных методов `Connect()`. В качестве аргумента ему передается адрес объекта `TSockAddr` серверного сокета. Существует также перегруженный вариант, позволяющий передать некоторые данные сразу во время установки соединения, но не все протоколы поддерживают его. Следующий пример де-

монстрирует подключение TCP-сокета к соответствующему порту веб-сайта, выполняемое для простоты синхронно.

```
// Получаем IP веб-сайта с помощью DNS
RHostResolver resolver;
resolver.Open(sock_serv, KAfInet, KProtocolInetTcp);
CleanupClosePushL(resolver);
TNameEntry name_entr;
_LIT(KSite, "devmobile.ru");
User::LeaveIfError(resolver.GetByName(KSite, name_entr));
TInetAddr addr(name_entr().iAddr);
CleanupStack::PopAndDestroy(&resolver);
// Можно было бы задать IP с помощью addr.SetAddress()

addr.SetPort(80); //порт HTTP

// Выполнение подключения (синхронно)
TRequestStatus status;
sock.Connect(addr, status);
User::WaitForRequest(status);
User::LeaveIfError(status.Int());
// Сокет подключен
<...> // Передача данных
```

Для установки пассивного соединения необходимо выполнить следующие действия.

1. Связывание сокета с локальным адресом.
2. Создание очереди ожидания входящий соединений.
3. Запуск асинхронного метода для получения входящего соединения.

Связать локальный адрес и сокет можно с помощью синхронного метода `Bind()`, принимающего единственный аргумент — ссылку на объект базового класса `TSockAddr`. Данный адрес должен использоваться для установки активных соединений.

Очередь ожидания входящих соединений создается с помощью синхронного метода `Local()`, аргументом которого является ее размер.

Для получения входящего соединения используется асинхронный метод `Accept()`, принимающий в качестве аргумента ссылку на так называемый “пустой” (blank) сокет. Пустой сокет создается с помощью метода `Open()`, без указания протокола и типа. В момент появления входящего соединения, оно помещается в очередь ожидания. Если в этот момент выполнялся асинхронный запрос `Accept()`, то он будет завершен, а переданный ему в качестве аргумента сокет будет связан с входящим соединением. После этого соединение удаляется из очереди ожидания, а связанный с ним сокет может использоваться для передачи данных. Если по каким-то причинам (разработчик забыл запустить метод `Accept()`) очередь ожидания окажется переполненной, то входящие соединения будут отклоняться.

Далее приводится простой пример создания серверного TCP-сокета с использованием порта 2000, способного работать лишь с одним входящим соединением.

```

RSocket sock;
_LIT(KTCPProt, "tcp");
User::LeaveIfError(sock.Open(sock_serv, KTCPProt));
CleanupClosePushL(sock);

// Зададим локальный адрес
// IP: 0.0.0.0; Порт: 2000
TInetAddr addr(KInetAddrAny, 2000);
User::LeaveIfError(sock.Bind(addr));
// Создание очереди ожидания на 1 входящее соединение
// Очередь большего размера не требуется,
// так как все операции выполняются синхронно
sock.Listen(1);

// Создание пустого сокета
RSocket incom;
User::LeaveIfError(incom.Open(sock_serv));
CleanupClosePushL(incom);

// Получение сокета входящего соединения
TRequestStatus status;
sock.Accept(incom, status);
// Ожидание входящего соединения
User::WaitForRequest(status);

if (status.Int() == KErrNone)
{
    <...> // Работа с сокетом incom
}

CleanupStack::PopAndDestroy(2, &sock);

```

Метод `Bind()` может также вызываться и для задания локального адреса в клиентских сокетах. Если он не вызывается, используется адрес по умолчанию.

При использовании протоколов TCP и UDP вместо метода `Bind()` можно использовать метод `SetLocalPort()`, позволяющий изменить лишь порт сокета. Для отмены выполнения асинхронного запроса `Connect()` класс `RSocket` предоставляет метод `CancelConnect()`.

Как и для метода `Open()`, методы `Listen()` и `Accept()` имеют перегруженные варианты, позволяющие указать в 8-битовом дескрипторе передаваемые при установке соединения данные. Однако эта возможность поддерживается не всеми протоколами и на практике применяется редко.

При использовании протоколов, не устанавливающих соединения, сокет может передавать или получать данные сразу после создания и связывания

с локальным адресом. Вызов же метода `Connect()` для сокетов такого типа не устанавливает соединения, но позволяет задать адрес, на который будет вестись передача данных по умолчанию.

Передача данных

Передача данных между сокетами осуществляется с помощью 8-битовых дескрипторов. Класс `RSocket` предоставляет несколько асинхронных методов для записи и чтения данных из сокета, работа которых зависит как от типа соединения, так и от типа сокета. Все они принимают в качестве аргументов 8-битовый дескриптор с данными для передачи и беззнаковое число, содержащее комбинацию специфичных для протокола флагов.

Для обмена информацией по протоколам, не устанавливающим соединения, используются асинхронные методы `SendTo()` и `RecvFrom()`. В них необходимо дополнительно передать ссылку на объект `TSockAddr`. При использовании метода `SendTo()` в нем указывается адрес серверного сокета. Отправляется все содержимое дескриптора. Существует перегруженный вариант метода `SendTo()`, позволяющий получить размер действительно отправленных данных. Он может быть полезен для возобновления передачи в случае возникновения ошибок связи. При получении данных с помощью метода `RecvFrom()` в объект `TSockAddr` помещается адрес клиентского сокета. Данные записываются в 8-битовый дескриптор. Перегруженный вариант метода позволяет получить размер данных, которые в действительности были прочитаны, для всех типов сокетов, кроме датаграмм. Ниже приводятся примеры синхронного использования этих функций для отправки сообщений (например, по протоколу UDP).

```
TInt SendDataTo(RSocket& aSocket, TDesC8& aData,
               TInetAddr& aAddr)
{
    TRequestStatus status;
    aSocket.SendTo(aData, aAddr, 0, status);
    User::WaitForRequest(status);
    return status.Int();
}

TInt ReceiveDataFrom(RSocket& aSocket, TDes8& aData,
                    TInetAddr& aAddr)
{
    TRequestStatus status;
    aSocket.RecvFrom(aData, aAddr, 0, status);
    User::WaitForRequest(status);
    return status.Int();
}
```

Сокеты, использующие в работе соединения, могут воспользоваться асинхронными методами `Send()` и `Recv()`, аналогичными методам `SendTo()` и `RecvFrom()`. Их отличие состоит в том, что эти методы не имеют аргумента типа `TSockAddr`. Кроме того, метод `Recv()` ведет себя по-разному в зависимо-

сти от типа сокета. Для сокетов-поточков он завершает свою работу лишь когда переданный ему дескриптор будет целиком наполнен передаваемыми данными либо когда связь будет разорвана. При этом дескриптор заполняется до своего максимального размера, не взирая на длину текущей строки в нем. Для сокетов, передающих данные в виде датаграмм, работа метода завершается сразу по получению очередной датаграммы. Если ее размер превысит максимальный размер дескриптора, часть данных будет потеряна.

Для сокетов, реализующих обмен данными в виде потоков, также может использоваться асинхронный метод `RecvOneOrMore()`. Его отличие от метода `Recv()` в том, что он завершает свою работу при получении любого количества данных, даже если они не заполняют дескриптор целиком. От также принимает в качестве аргумента числовую переменную, в которую помещается размер полученных данных.

Для отмены асинхронных операций `Send()` и `Recv()` служат методы `CancelSend()` и `CancelRecv()`.

Несколько упрощенной альтернативой методам `Send()` и `Recv()` являются методы `Write()` и `Read()`. Они отличаются тем, что не имеют аргумента для указания флагов. Кроме того, их могут использовать сокеты, не устанавливающие соединения, но лишь в том случае, если для них был задан адрес по умолчанию с помощью метода `Connect()`. Отменить выполнение этих асинхронных методов можно с помощью методов `CancelWrite()` и `CancelRead()` соответственно.

Заккрытие сокета

В тот момент, когда разработчику понадобится закрыть сокет, он все еще может выполнять некоторые операции и иметь непустые буферы данных ввода-вывода. Поэтому, прежде чем вызвать стандартный для R-классов метод освобождения `Close()`, необходимо прервать данные операции. В случае если сокет служит для передачи данных по протоколу, не использующему соединения, для этого достаточно вызвать метод `CancelAll()`.

Если же протокол сокета использует соединения, то перед его закрытием необходимо вызвать асинхронный метод `Shutdown()`. Детали его работы зависят от протокола, но его основное назначение — уведомление удаленного сокета о разрыве соединения и прекращение всех операций. Метод `Shutdown()` принимает в качестве аргумента одно из следующих значений перечисления `RSocket::TShutdown`:

- `ENormal` — разрыв соединения по завершению операций чтения и записи;
- `EStopInput` — принудительное завершение всех операций чтения из сокета и разрыв соединения по окончании всех операций записи в него;
- `EStopOutput` — принудительное завершение всех операций записи в сокет и разрыв соединения по окончании всех операций чтения из него;
- `EImmediate` — принудительное завершение всех операций и разрыв соединения.

Существует также перегруженный вариант метода `Shutdown()`, позволяющий передать дополнительные данные при разрыве соединения. Удаленный сокет может получить такую информацию с помощью метода `GetDisconnectData()`. Данная возможность поддерживается не всеми протоколами.

Прочие операции

Так как нюансы исполнения многих операций сокета зависят от его протокола и настроек, класс `RSocket` предоставляет множество методов для получения различной информации о нем. Кроме того, он содержит ряд базовых методов для выполнения специфичных для протокола операций. Обычно они принимают в качестве аргумента идентификатор команды или параметра, описание которых следует искать в информации о протоколе.

- Метод `Info()`, подобно `RSocketServ::GetProtocolInfo()`, позволяет получить структуру `TProtocolDesc` с полной информацией о протоколе сокета.
- Методы `GetOpt()` и `SetOpt()` служат для получения и изменения различных параметров сокета. Помимо значения параметра, они принимают в качестве аргументов его идентификатор и уровень. Уровни служат для разграничения стандартных параметров, задаваемых сервером `ESock`, и параметров, порожденных от `RSock`-классов.
- `LocalName()` — локальный адрес сокета, задаваемый в методе `Bind()` или `Connect()`.
- `LocalPort()` — возвращает порт сокета.
- `RemoteName()` — адрес удаленного сокета.
- Асинхронный метод `Ioctl()` и отменяющий его `CancelIoctl()` служат для выполнения специфичных для протокола запросов.
- `Transfer()` — позволяет передать сокет другой сессии сервера `ESock`.

Подключения

Некоторые интернет-протоколы (например, TCP и UDP) допускают использование различных физических каналов передачи данных. Например, подключение к сети может быть выполнено с помощью GPRS, WiFi, Bluetooth, инфракрасного порта, и даже локальной сети (в эмуляторе на ПК). Если в момент открытия сокета на устройстве отсутствует подключение, подходящее для передачи его данных, то оно устанавливается. В том случае, если на устройстве содержатся несколько настроек, позволяющих выполнить подключение, то пользователю предварительно будет предложено выбрать одну из них. В частности, при установке подключения к сети Интернет предлагается выбрать одну из точек доступа.

Сервер ESock предоставляет API для создания, поиска и отслеживания состояния используемых устройством подключений. Оно реализуется классом RConnection. Методы Open () классов RHostResolver и RSock имеют перегруженные варианты, позволяющие указать в качестве аргумента ссылку на объект RConnection. Эта возможность позволяет использовать предварительно установленное подключение и часто применяется на практике для подавления диалогового окна выбора точки доступа при работе с Интернет.

Более подробную информацию, а также примеры использования класса RConnection вы можете найти в справочнике SDK и Википедии сообщества Forum Nokia.

Подготовка к сертификации ASD

- Умение задать правильные утверждения, определяющие и описывающие сетевой сокет.
 - Понимание назначения и роли модулей протоколов PRT в Symbian OS.
 - Понимание разницы между использующими и не использующими соединения сокетом.
 - Понимание разницы между передачей данных с помощью потока и в виде датаграмм, а также их связь с использующими и не использующими соединения сокетом.
 - Знание характеристик классов RSocketServ, RSocket и RHostResolver.
 - Понимание базовых принципов использования сокетов в Symbian OS.
 - Умение определять правильный шаблон кода открытия и конфигурации сокета для протоколов с использованием соединений и без них.
 - Знание того, какие методы класса RSocket должны применяться для приема и передачи данных при работе с использованием соединений и без них.
 - Знание характеристик синхронного и асинхронного способа закрытия сессии сокета.
 - Умение задать правильные утверждения о транспортной независимости.
-

Сервер окон

Сервер окон (window server, или WSERV: файл wserv.exe, SID 0x10003b20) является одним из базовых компонентов системы. Он загружен постоянно и запускается при включении устройства. Сервер окон предназначен для обеспечения разделяемого доступа различных приложений к экрану и различным средствам ввода данных устройства (клавиатура, сенсоры прикосновения). Он обрабатывает некоторые предоставляемые ему ядром аппаратные события (например, нажатия клавиш) и рассылает уведомления о них. Он также осуществляет вывод изображений на экран с учетом его ориентации.

Разделяемый доступ к экрану устройства организуется сервером WSERV с помощью **окон** (window), объединенных в группы. Под *окном* в Symbian C++

понимается область экрана, в которую выводится изображение. Приложение может иметь множество окон. Они также могут быть вложенными и полностью или частично перекрывать друг друга. Сервер отвечает за определение того, содержимое каких окон необходимо отобразить на экране. Обычно на экран выводится изображение сразу нескольких окон, принадлежащих различным программам. Например, совокупность окон, формирующих панель статуса (отображающую часы, индикаторы и пр.), принадлежит системному приложению и видна на экране устройства практически всегда, тогда как в остальной части экрана выводятся окна прочих, в том числе и пользовательских приложений. Использование окон также позволяет перейти от абсолютных координат экрана к относительным (в границах окна), что упрощает рисование в нем.

Существует несколько типов окон, но все они подчиняются строгой иерархии, изображенной на рис. 6.11. На самом верхнем уровне находится **корневое окно** (root window) — его создает сам сервер. Все остальные окна образуют дерево с корнем в нем. В случае если устройство использует сразу несколько экранов (например, коммуникатор Nokia E90), сервер создает корневое окно для каждого из них (тогда все окна, согласно теории графов, будут организованы в виде леса).

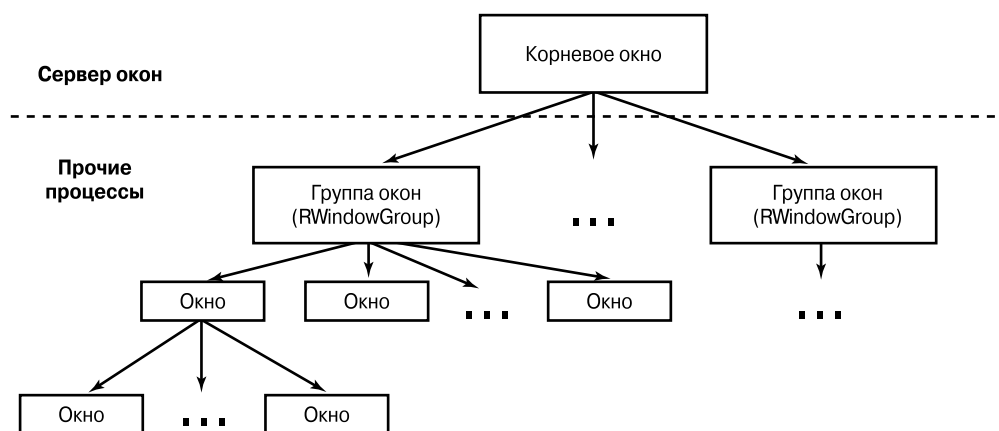


Рис. 6.11. Дерево окон в системе

На втором уровне находятся **группы окон** (window group), работа с которыми осуществляется с помощью класса `RWindowGroup`. Они служат для задания общих характеристик входящих в них окон, а также могут подписываться на рассылаемые сервером извещения о событиях. Группы окон являются псевдоокнами, определяют область экрана и не используются для вывода изображений.

Потомками групп окон могут быть окна различных типов. Они соответствуют некоторой области экрана и используются для вывода графики.

Все классы окон являются потомками базового класса `RWindowTreeNode`. Он содержит методы определения родителей, потомков и соседей окна, позволяющие перебирать элементы дерева. Каждый элемент дерева содержит

связный список своих потомков. Они упорядочены по старшинству. Первый потомок окна считается самым старшим. При отображении на экране полностью или частично перекрывающихся окон одной группы действуют следующие правила.

- Окно, являющееся прямым потомком группы окон, может иметь любой допустимый размер и положение на экране. Если же оно является потомком другого окна, то его границы не должны выходить за границы области родителя. Не попадающие в регион родительского окна части не отображаются.
- Если окна связаны отношением “родитель-потомок”, то потомок выводится поверх родительского окна.
- Если окна являются соседними, то сверху выводится старшее окно.

Группы окон также связаны отношениями старшинства, но, помимо этого, могут иметь различный приоритет. Старшинство учитывается лишь при определении порядка отображения групп окон одного приоритета.

Корневое окно отрисовывается на экране позади всех остальных, в качестве заднего фона. Оно заливает сплошным цветом неиспользуемые области экрана. В действительности устройства используют всю доступную поверхность экрана, и корневое окно никогда не отображается.

На рис. 6.12 приводится диаграмма, демонстрирующая иерархию классов окон. Как уже упоминалось ранее, все они порождены от класса `RWindowTreeNode`, позволяющего организовать окна в виде дерева. От него наследуются классы `RWindowGroup` и `RWindowBase`. Класс `RWindowGroup` служит для объединения окон в группы и не используется для вывода изображений на экран.

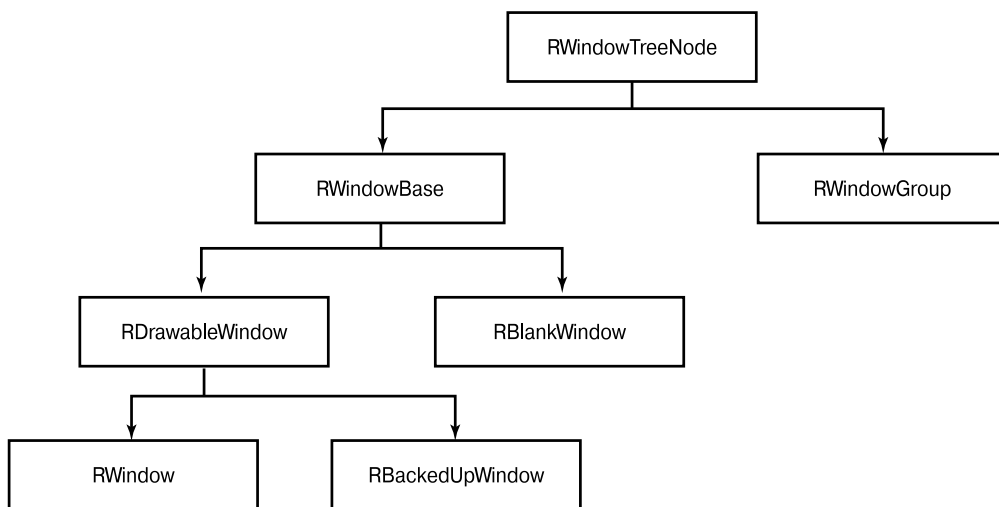


Рис. 6.12. Иерархия классов окон

Класс `RWindowBase` является базовым и не предназначен для использования разработчиком (его конструктор объявлен в секции `protected`). В нем определены методы, позволяющие задать размер и положение окна. Порожденный от него класс `RBlankWindow` практически не добавляет новой функциональности и может служить контейнером для других типов окон. Объекты `RBlankWindow` не могут использоваться для вывода графики.

Наследуемый от `RWindowBase` класс `RDrawableWindow` также является базовым. От `RDrawableWindow` наследуются классы окон, которые могут использоваться для вывода изображений на экран. Классы `RWindowBase` и `RDrawableWindow` можно часто встретить в качестве аргументов функций.

Классы `RWindow` и `RBackedUpWindow` непосредственно используются для вывода графики. На практике чаще всего применяется `RWindow`. Класс `RBackedUpWindow` отличается от него тем, что при работе с его объектом в системе хранится изображение с копией содержимого окна. Оно может использоваться для быстрой перерисовки. Это значительно быстрее, чем отправка запроса на перерисовку содержимого окна создавшему его приложению. Но в то же время объекты класса `RBackedUpWindow` занимают больше памяти.

К сожалению, мы не сможем подробно рассмотреть в рамках этой книги все функции вышеописанных классов. Их слишком много, поэтому в дальнейшем речь пойдет лишь о наиболее часто используемых методах, рассматриваемых в контексте решения конкретных задач.

Получение уведомлений о нажатиях клавиш

Сессия сервера окон реализуется классом `RWsSession`, объявленным в заголовочном файле `w32std.h`. Его использование требует подключения библиотеки импорта `ws32.lib`. Чтобы установить сессию с сервером, необходимо воспользоваться методом `Connect()`, принимающим ссылку на объект `RFs`. Существует также перегруженный вариант без параметров. При его использовании объект `RWsSession` установит собственную сессию с файловым сервером и будет отвечать за ее завершение. Разорвать сессию с сервером окон и освободить все занимаемые ею ресурсы можно с помощью метода `Close()`.

Для того чтобы получать уведомления о нажатиях клавиш, необходимо создать группу окон. Для работы с группами окон служит класс `RWindowGroup`. При создании экземпляра этого класса необходимо передать ссылку на сессию сервера окон в его конструктор.

```
// Установка сессии с сервером окон
RWsSession wserv;
User::LeaveIfError(wserv.Connect());
CleanupClosePushL(wserv);
// Создание объекта RWindowsGroup
RWindowGroup wg(wserv);
```

Создать новую группу окон можно с помощью метода `Construct()`. Он принимает в качестве аргумента значение хендла группы. Это значение должно быть уникальным в рамках сессии сервера окон. На практике в качестве значения хендла группы используют адрес объекта `RWindowGroup`.

```
// Создание группы окон
User::LeaveIfError(wg.Construct((TUint32) &wg));
CleanupClosePushL(wg);
```

Созданная таким образом группа окон помещается в начало списка всех существующих групп. Ее порядковый номер и приоритет равны нулю. Если групп с более высоким приоритетом не существует (чаще всего так оно и есть), то именно эта группа будет получать все события от сервера. Так как окон в ней пока не создано, на экране ничего не изменится. Но если бы эти окна существовали, то они выводились бы поверх всех остальных. В дальнейшем изменить порядковый номер и приоритет группы можно с помощью метода `SetOrdinalPositionErr()`, а для получения их значений служат методы `OrdinalPosition()` и `OrdinalPriority()`.

Еще одной характеристикой созданной нами группы окон является ее способность получать фокус ввода. Существует перегруженный вариант метода `Construct()`, позволяющий указать значение этой характеристики. По умолчанию оно равно `ETrue`. В дальнейшем способность получать фокус можно включать или отключать с помощью метода `EnableReceiptOfFocus()`. Только способные получать фокус ввода группы уведомляются о нажатиях клавиш. В каждый момент времени фокус имеет только одна группа окон в системе. Если получение фокуса в группе окон, отображающейся на экране самой верхней, отключено, то сообщения о нажатиях клавиш будут переправляться следующей за ней.

Создать группу окон можно также с помощью метода `ConstructChildApp()`. В отличие от метода `Construct()`, он позволяет поместить новую группу окон в “цепочку”, указав идентификатор родительской группы. Такой идентификатор можно получить с помощью метода `Identifier()`. При изменении положения одной из входящей в цепочку групп все остальные группы той же цепочки также меняют свое положение, сохраняя при этом порядок относительно друг друга. Цепочки применяются при встраивании одного приложения в другое в тех случаях, когда при переключении между программами они должны вести себя как одно целое.

Итак, мы выяснили что, несмотря на то, что созданная нами группа никак не отображается на экране, именно она является самой верхней, а также имеет фокус. Теперь сервер окон будет отправлять все стандартные сообщения именно ей, — до тех пор, пока другая группа не вытеснит ее с первой позиции списка. Под стандартными сообщениями понимаются все типы сообщений, кроме уведомлений о необходимости перерисовки и нажатии специальных “приоритетных” клавиш (для них уведомления можно получать с помощью отдельных асинхронных методов `RedrawReady()` и `PriorityKeyReady()`). Подписаться

на уведомления о стандартных событиях можно с помощью асинхронного метода `EventReady()` класса `RWsSession`. После того как асинхронный запрос будет исполнен, информацию о наступившем событии необходимо извлечь с помощью метода `GetEvent()`. Отменить выполнение асинхронного запроса можно с помощью метода `EventReadyCancel()`. Повторный вызов метода `EventReady()` может быть сделан только после того, как был использован метод `GetEvent()` или `EventReadyCancel()`. Следующий пример демонстрирует синхронный (для краткости) вариант получения уведомлений от сервера окон.

```
TRequestStatus status;
// Асинхронный запрос на уведомление о событии
wserv.EventReady(&status);

// Синхронно ожидаем его завершения
User::WaitForRequest(status);

// Получаем информацию о событии
TWsEvent e;
wserv.GetEvent(e);

// Если была нажата клавиша ...
if (status.Int() == KErrNone)
    if (e.Type() == EEventKeyDown && e.Key())
        switch (e.Key()->iScanCode)
        {
            // Обработка нажатой клавиши
        }

// Объекты были помещены в стек в предыдущих примерах
CleanupStack::PopAndDestroy(2, &wserv); // wserv, wg
```

Информацию о стандартных событиях сервер окон передает с помощью объекта класса `TWsEvent`. Ее структура зависит от типа события, узнать который можно с помощью метода `Type()`. Он может принимать одно из значений перечисления `TEventCode`. Их около трех десятков, и здесь мы рассмотрим лишь наиболее часто используемые.

- `EEventNull` — пустое событие, должно игнорироваться клиентом.
- `EEventKeyDown` — клавиша нажата, но еще не отпущена.
- `EEventKeyUp` — клавиша, которая была нажата, отпущена.
- `EEventKey` — нажатие клавиши. В зависимости от того, какая клавиша нажата, может посылаться после `EEventKeyDown` (чаще всего) или `EEventKeyUp` (реже — обычно для клавиш-модификаторов).
- `EEventModifiersChanged` — изменение состояния модификаторов (например, `<Alt>` или `<Shift>`). Отслеживается сервером окон.

- `EEventPointer` — событие сенсорного экрана. Используется вместе с `EEventPointerEnter`, `EEventPointerExit` и `EEventDragDrop`.
- `EEventFocusGained` — ваша группа окон получила фокус.
- `EEventFocusLost` — ваша группа окон утратила фокус.
- `EEventFocusGroupChanged` — теперь фокус принадлежит другой группе окон.
- `EEventWindowGroupsChanged` — сигнализирует об удалении или изменении имени группы окон.
- `EEventWindowGroupListChanged` — изменения в списке групп окон: добавление новой группы, удаление существующей или изменение их порядка.
- `EEventWindowVisibilityChanged` — изменение видимости окна.

Многие типы уведомлений по умолчанию не отсылаются клиенту. Например, для того чтобы получать уведомления типа `EEventWindowGroupListChanged`, необходимо предварительно включить эту возможность с помощью метода `EnableGroupListChangeEvents()` объекта `RWindowGroup`. Поэтому, прежде чем использовать их, ознакомьтесь с соответствующей информацией справочника SDK.

Обычно приложение использует только одну группу окон. Если это не так, то определить, какой из групп предназначалось сообщение, можно с помощью метода `Handle()` класса `TWsEvent`. Он возвращает уникальный хендл группы окон, использованный при ее создании.

Для определения времени, в которое произошло событие, служит метод `Time()`, возвращающий объект `TTime`.

В зависимости от типа события, разработчик может получить указатель на дополнительную информацию о нем с помощью различных методов. Например, метод `ModifiersChanged()` возвращает указатель на объект класса `TModifiersChangedEvent`, а `Pointer()` — на указатель `TPointerEvent`. Интересующую нас информацию о нажатой клавише можно получить с помощью метода `Key()`, и передана она будет в объекте `TKeyEvent`.

Структура `TKeyEvent` содержит следующие четыре поля.

- `iCode` — код символа, генерирующегося нажатием клавиши. Используется только в событиях типа `EEventKey`. Содержит значение перечисления `TKeyCode` (например, `EKeyEnter`, `EKeyRightArrow` или `EKeyMenu`).
- `iScanCode` — стандартный код клавиши. Содержит значение перечисления `TStdScanCode` (например, `EStdKeyEnter`, `EStdKeyLeftArrow` или `EStdKeyDevice0`).
- `iModifiers` — состояние клавиш-модификаторов на момент нажатия.
- `iRepeats` — количество “повторений” нажатия клавиши в случае, если она удерживалась некоторое время.

Перечисления `TKeyCode` и `TStdScanCode` объявлены в заголовочном файле `e32keys.h` и в данной книге не приводятся. Во-первых, они содержат слишком

много значений. Во-вторых, производители устройств, к сожалению, не всегда используют для клавиши тот код, который предназначен для нее в ОС Symbian. Кроме того, для некоторых клавиш подходящих значений не объявлено. В качестве примера я приведу некоторые часто используемые, но неочевидные коды клавиш из перечисления `TKeyCode`, применяющиеся на устройствах фирмы Nokia:

- `EKeyDevice0`, `EKeyDevice1`, `EKeyDevice3` — левая, правая и средняя клавиши “soft key” (кнопки выбора команд меню);
- `EKeyDevice2` — кнопка включения/выключения устройства (power button);
- `EKeyYes` и `EKeyNo` — кнопки выполнения и отмены звонка (часто с изображением зеленой и красной телефонной трубки соответственно);
- `EKeyApplication0` — кнопка вызова главного меню устройства.

К сожалению, официального документа, содержащего коды клавиш различных моделей, не существует, но в большинстве случаев коды кнопок цифровых клавиатур и QWERTY-клавиатур одного производителя совпадают.

Еще более проблематичным может оказаться перехват различных специфичных для конкретной модели кнопок (например, управление воспроизведением мультимедиа). Вполне возможно, что они не обрабатываются сервером окон, и для получения уведомлений об их нажатии потребуется использовать специальное API (чаще всего, не входящее в SDK, а являющееся частью SDK Extension Pack или специального опубликованного набора дополнений к SDK для работы с данной моделью устройства). В отдельных случаях такое API для пользовательских приложений может и вовсе не существовать.

Класс `RWindowGroup` также предоставляет методы для получения уведомлений о нажатии клавиш даже в том случае, если группа окон не имеет фокуса и не является первой в списке. Это методы `CaptureKey()`, `CaptureKeyUpAndDowns()` и `CaptureLongKey()`. При их использовании необходимо указать код перехватываемой клавиши, а также модификаторы, которые должны быть включены в этот момент. Метод `CaptureKey()` регистрирует клиент на получение уведомлений типа `EEventKey`, а метод `CaptureKeyUpAndDowns()` — на события `EEventKeyDown`. Метод `CaptureLongKey()` позволяет получать сообщения об удерживании определенной клавиши. В качестве результата работы вышеупомянутых методов возвращается хендл на регистрацию об уведомлении на данное событие либо код ошибки. В дальнейшем такой хендл может быть использован для отмены уведомлений с помощью методов `CancelCaptureKey()`, `CancelCaptureKeyUpAndDowns()` и `CancelCaptureLongKey()` соответственно. Использование методов `CaptureXXX()` требует наличия доступа к защищенной возможности `SwEvent`. Чаще всего эти методы применяются в сервисах, не имеющих GUI или выполняющихся в фоновом режиме, — скажем, это может быть утилита, сохраняющая изображение экрана (screen snap) в файл

при нажатии на определенную клавишу. Для того чтобы отправить группу окон в конец списка групп, достаточно в качестве ее порядкового номера указать -1. Пример, код которого приведен в листингах 6.22 и 6.23, демонстрирует класс, позволяющий перехватывать нажатие заданных клавиш из выполняющегося в фоновом режиме приложения.

Листинг 6.22. Файл KeyCatcher.h

```
#include <e32base.h>
#include <w32std.h> // RWsSession

// Класс MKeyObserver позволяет другим классам приложения
// реализовать методы обратного вызова и обрабатывать события,
// полученные CKeyCatcher

class MKeyObserver
{
public:

    // Значения перечисления TCaptureRes включаются в результат
    // метода обработчика для управления поведением CKeyCatcher

    enum TCaptureRes
    {
        EKeyRepeat = 1,    // Переслать событие окну с фокусом
        ECancel = 2        // Прекратить перехват нажатия клавиш
    };

    // Методы-обработчики событий
    virtual TInt ProcessKeyEvent(const TInt aKeyCode) = 0;
    virtual TInt ProcessKeyUpDown(const TInt aKeyStdCode) = 0;
};

// Структура, хранящая хендл, возвращаемый
// методами RWsSession::CaptureXXX(), и код клавиши

struct TKeyCatchInfo
{
    inline TKeyCatchInfo(TBool aUpDown, TInt aKeyCode);
    TInt iHandle;
    TInt iKeyCode;
    TBool iUpDown; // CaptureKey() или CaptureKeyUpAndDowns()
};

// Конструктор структуры, выполняющий инициализацию ее членов
inline TKeyCatchInfo::TKeyCatchInfo(TBool aUpDown,
    TInt aKeyCode) : iHandle(0), iUpDown(aUpDown),
    iKeyCode(aKeyCode)
```

```

    {
    }

// Класс активного объекта, осуществляющего регистрацию
// и отмену перехвата нажатия клавиш

class CKeyCatcher : public CActive
{
public:
    // Деструктор и часть двухфазного конструктора
    ~CKeyCatcher();
    static CKeyCatcher* NewL(MKeyObserver& aObserver);
    static CKeyCatcher* NewLC(MKeyObserver& aObserver);

private:
    // C++ конструктор
    CKeyCatcher(MKeyObserver& aObserver);
    // Часть двухфазного конструктора
    void ConstructL();

private:
    // Методы активного объекта
    void DoCancel();
    void RunL();

public:
    // Регистрация на получение уведомлений о нажатии
    TInt CaptureKeyEventL(TBool aUpDown, TInt aKeyCode);
    // Прекращение получения уведомлений о нажатии
    TInt CancelCaptureKeyEvent(TBool aUpDown, TInt aKeyCode);
    // Запуск асинхронного запроса
    void Start();

private:
    // Универсальный метод для регистрации на нажатие
    void CaptureKey(TKeyCatchInfo& aInfo);
    // Универсальный метод для отмены регистрации на нажатие
    void CancelCaptureKey(const TKeyCatchInfo aInfo);

private:
    // Интерфейс обработчика событий
    MKeyObserver& iObserver;
    // Сессия сервера окон
    RWsSession iWServ;
    // Группа окон
    RWindowGroup* iWg;
    // Массив хендлов RWsSession::CaptureXXX()
    RArray<TKeyCatchInfo> iKeyHandles;
};

```

Листинг 6.23. Файл KeyCatcher.cpp

```

#include "KeyCatcher.h"

CKeyCatcher::CKeyCatcher(MKeyObserver& aObserver):
    CActive(EPriorityNormal), iObserver(aObserver)
{
}

CKeyCatcher* CKeyCatcher::NewLC(MKeyObserver& aObserver)
{
    CKeyCatcher* self = new (ELeave) CKeyCatcher(aObserver);
    CleanupStack::PushL(self);
    self->ConstructL();
    return self;
}

CKeyCatcher* CKeyCatcher::NewL(MKeyObserver& aObserver)
{
    CKeyCatcher* self = CKeyCatcher::NewLC(aObserver);
    CleanupStack::Pop(); // self;
    return self;
}

void CKeyCatcher::ConstructL()
{
    // Установка сессии с файловым сервером
    User::LeaveIfError(iWServ.Connect());

    // Создание объекта группы окон
    // Используется указатель, т.к. ссылка на RWsSession
    // должна быть передана в конструкторе
    iWg = new (ELeave) RWindowGroup(iWServ);

    CleanupStack::PushL(iWg);
    // Создание группы окон, не принимающей фокуса
    User::LeaveIfError(iWg->Construct(TUint32(&iWg), EFalse));
    CleanupStack::Pop(iWg);

    // Группа окон отправляется в конец списка и больше
    // не является верхней на экране
    iWg->SetOrdinalPosition(-1);

    CActiveScheduler::Add(this);
}

CKeyCatcher::~CKeyCatcher()
{
}

```

```

// Отмена асинхронного запроса (если необходимо)
Cancel();

// Отмена регистрации на все нажатия
for (TInt i = 0; i < iKeyHandles.Count(); i++)
    CancelCaptureKey(iKeyHandles[i]);

// Удаление группы окон
if (iWg)
{
    iWg->Close();
    // Уничтожение объекта для работы с группами окон
    delete iWg;
}

// Закрытие сессии с оконным сервером
iWServ.Close();
}

void CKeyCatcher::DoCancel()
{
    // Выполнение отмены асинхронного запроса
    iWServ.EventReadyCancel();
}

void CKeyCatcher::CaptureKey(TKeyCatchInfo& aInfo)
{
    // Выполнение регистрации на уведомление о нажатии
    if (!aInfo.iUpDown)
        aInfo.iHandle = iWg->CaptureKey(aInfo.iKeyCode, 0, 0);
    else
        aInfo.iHandle =
            iWg->CaptureKeyUpAndDowns(aInfo.iKeyCode, 0, 0);
}

void CKeyCatcher::CancelCaptureKey(const TKeyCatchInfo aInfo)
{
    // Выполнение отмены регистрации на уведомление о нажатии
    if (aInfo.iUpDown)
        iWg->CancelCaptureKeyUpAndDowns(aInfo.iHandle);
    else
        iWg->CancelCaptureKey(aInfo.iHandle);
}

TInt CKeyCatcher::CaptureKeyEventL(TBool aUpDown,
                                   TInt aKeyCode)
{
    // Универсальная регистрация на нажатие

```

```

// Создание объекта перечисления для хранения данных
TKeyCatchInfo info(aUpDown, aKeyCode);

// Выполнение регистрации
CaptureKey(info);

// Если регистрация выполнена с ошибкой - сброс
User::LeaveIfError(info.iHandle);

// Сохраняем данные об успешной регистрации
iKeyHandles.AppendL(info);

return KErrNone;
}

TInt CKeyCatcher::CancelCaptureKeyEvent(TBool aUpDown,
                                         TInt aKeyCode)
{
    // Универсальная отмена регистрации на нажатие

    // Находим информацию о регистрации на нажатие в массиве
    for (TInt i = 0; i < iKeyHandles.Count(); i++)
        if (iKeyHandles[i].iUpDown == aUpDown &&
            iKeyHandles[i].iKeyCode == aKeyCode)
        {
            // Отмена перехвата данной клавиши
            CancelCaptureKey(iKeyHandles[i]);
            // Удаление из массива перехватываемых событий
            iKeyHandles.Remove(i);
            return KErrNone;
        }

    // Данный тип события или клавиша не перехватывались
    return KErrNotFound;
}

void CKeyCatcher::Start()
{
    // Запуск асинхронного запроса для получения уведомлений

    // Отмена предыдущих асинхронных запросов, если они были
    Cancel();

    // Если группа окон зарегистрировалась на перехват
    // хотя бы одного события
    if (iKeyHandles.Count() >= 0)
    {

```

```

        // Запуск асинхронного запроса
        iWServ.EventReady(&iStatus);
        SetActive();
    }

    // В дальнейшем проверить, удалось ли запустить цикл
    // ожидания, можно с помощью CKeyCatcher::IsActive(),
    // а остановить его с помощью CKeyCatcher::Cancel().
    // Это методы базового класса CActive.
}

void CKeyCatcher::RunL()
{
    // Обработчик результатов асинхронного запроса

    if (iStatus != KErrNone) return; // Ошибка или отмена
                                    // запроса

    // Получение информации о событии
    TWsEvent event;
    iWServ.GetEvent(event);

    TInt res(KErrNone);

    // Если событие является нажатием на клавишу
    if (event.Key())
        switch (event.Type())
        {
            // Вызов соответствующего метода обратной связи
            case EEventKeyDown:
            case EEventKeyUp:
            {
                res = iObserver.ProcessKeyUpDown(
                    event.Key()->iScanCode);
            }
            break;

            case EEventKey:
            {
                res = iObserver.ProcessKeyEvent(
                    event.Key()->iCode);
            }
            break;

            default:
                break;
        }
}

```

```

// Если слушатель вернул ошибку - сброс
User::LeaveIfError(res);

// Если в результате присутствует EKeyRepeat,
// отправляем событие группе окон с фокусом
if (res & MKeyObserver::EKeyRepeat)
{
    TInt id = iWServ.GetFocusWindowGroup();
    iWServ.SendEventToWindowGroup(id, event);
}

// Если в результате не присутствует ECancel,
// продолжаем перехватывать события
if (res & MKeyObserver::ECancel)
    Start();
}

```

Вышеприведенный класс `CKeyCatcher` самостоятельно устанавливает сессию с сервером `WSERV` и создает группу окон. Сразу после этого эта группа отправляется в конец списка с помощью вызова `SetOrdinalPosition(-1)`. Так как созданная группа не способна получать фокус, то, даже если при каких-то условиях она вновь окажется на вершине списка, разработчик может быть уверен, что ей не будут переданы никакие уведомления о нажатых клавишах, кроме тех, на которые она была зарегистрирована.

Объект класса `RWindowGroup` для управления создается в куче с помощью оператора `new`. Он не может быть просто объявлен членом класса `CKeyCatcher`, так как в его конструкторе необходимо указать ссылку на сессию сервера окон, а она еще не установлена на момент вызова конструктора. В целях экономии ресурсов разработчик может изменить класс `CKeyCatcher` и при создании его объекта передавать ссылку на объект `RFs` для использования перегруженного конструктора `RWsSession` либо ссылку на уже установленную сессию окон (актуально для GUI-приложений, в которых сессия с сервером окон устанавливается автоматически и может быть получена из окружения `Eikon`).

Для хранения хендлов, возвращаемых методами `CaptureXXX()`, и их соответствия кодам перехватываемых клавиш определена структура `TKeyCatchInfo`. Класс `CKeyCatcher` инкапсулирует динамический массив таких структур, позволяя отменять получения уведомлений о нажатии некоторых клавиш во время выполнения программы.

Информация о полученных событиях передается в методы обратного вызова клиента, реализовавшего интерфейс `MKeyObserver`. Ссылка на такой класс указывается при вызове статических методов `CKeyCatcher::NewXXX()`. Такой способ связи позволяет разделить логику получения событий и логику их обработки, а также использовать однажды написанный класс `CKeyCatcher` в различных приложениях без изменений. Методы-обработчики должны возвращать стандартный код ошибки либо комбинацию элементов перечисления `MKeyObserver::TCaptureRes`, служащих для управления поведением объекта

класса `CKeyCatcher`. Их значения являются степенями двойки, поэтому создание такой комбинации и проверка на вхождение в нее элементов перечисления могут осуществляться с помощью логических операций. В вышеприведенном примере реализованы только две команды: `ERepeat` (пересылка полученного события группе окон, имеющей фокус) и `ECancel` (прекращение получения событий). Следующий код демонстрирует класс, реализующий интерфейс `MKeyObserver`.

```
#include "KeyCatcher.h"

// Объявление класса

class CKeyObserver : public CBase, public MKeyObserver
{
public:
    TInt ProcessKeyEvent(const TInt aKeyCode);
    TInt ProcessKeyUpDown(const TInt aKeyStdCode);
};

// Реализация методов интерфейса

TInt CKeyObserver::ProcessKeyEvent(const TInt aKeyCode)
{
    <...> // Обработка кода клавиши
    return MKeyObserver::EKeyRepeat;
}

TInt CKeyObserver::ProcessKeyUpDown(const TInt aKeyStdCode)
{
    <...> // Обработка кода клавиши
    return MKeyObserver::EKeyRepeat;
}
```

В этом случае использовать класс `CKeyCatcher` в приложении без GUI можно следующим образом.

```
// Создание объекта, обрабатывающего уведомления о нажатиях
CKeyObserver* observer = new (ELeave) CKeyObserver();
CleanupStack::PushL(observer);

// Создание объекта, получающего уведомления о нажатиях
CKeyCatcher* key_catcher = CKeyCatcher::NewLC(*observer);

// Регистрация для всех типов событий
// при нажатиях на левый и правый soft key
User::LeaveIfError(key_catcher->CaptureKeyEventL(EFalse,
    EKeyDevice0));
User::LeaveIfError(key_catcher->CaptureKeyEventL(EFalse,
    EKeyDevice1));
```

```
User::LeaveIfError(key_catcher->CaptureKeyEventL(ETrue,
    EStdKeyDevice0));
User::LeaveIfError(key_catcher->CaptureKeyEventL(ETrue,
    EStdKeyDevice1));

// Запуск цикла ожидания событий сервера окон
key_catcher->Start();

// Запуск цикла ожидания планировщика активных объектов
CActiveScheduler::Start();

// Завершение работы приложения
CleanupStack::PopAndDestroy(2, observer);
```

Обратите особое внимание на возможность пересылать события другим группам окон. Если какая-либо группа окон зарегистрировалась на получение уведомлений о нажатии определенной клавиши с помощью метода `CaptureXXX()`, то это событие будет отправляться только ей, вне зависимости от того, получила группа фокус или нет. Если данный метод перехвата нажатий на клавиши применяется в сервисах, выполняемых в фоновом режиме (чаще всего так и есть), то это может привести пользователя в замешательство. Обычно выполняемым в фоне процессам достаточно простого мониторинга нажатий клавиш. Поэтому сразу после получения уведомления об их нажатии они должны самостоятельно переслать эту информацию той группе окон, которая в действительности должна была ее получить. Для этого в классе `RWsSession` определены следующие методы:

- `SendEventToWindowGroup()` — отправка события группе окон с указанным идентификатором;
- `SendEventToAllWindowsGroups()` — отправка события всем существующим группам окон;
- `SendEventToOneWindowGroupsPerClient()` — отправка события лишь одной группе окон каждой клиентской сессии сервера.

Для поиска групп окон предназначены методы:

- `GetFocusWindowGroup()` — возвращает идентификатор группы, имеющий фокус;
- `NumWindowGroups()` — возвращает количество всех существующих групп окон;
- `WindowGroupList()` — позволяет получить список идентификаторов всех групп;
- `FindWindowGroupIdentifier()` — поиск группы по имени и получение ее идентификатора.

К сожалению, необходимость самостоятельной трансляции события имеющей фокус группе окон — не единственный недостаток использования методов `CaptureXXX()`. Если в системе имеются два приложения, зарегистрировавшие-

ся таким образом на получение одних и тех же клавиш, то они будут мешать друг другу — уведомление получит лишь одно из них (зарегистрировавшееся последним). Кроме того, ряд приложений не способно правильно обработать события, посланные их группе окон, — например, некоторые мидлеты Java.

Рисование на экране

Для вывода графики на экран используются лишь два класса окон: `RWindow` и `RBackedUpWindow`. Мы ограничимся рассмотрением лишь одного из них — `RWindow`.

Объект класса `RWindow` создается точно так же, как объект `RWindowGroup`: в его конструктор передается ссылка на сессию сервера `WSERV`. Затем необходимо вызвать метод `Construct()` для создания окна. Аргументами метода являются ссылка на объект базового класса `RWindowTreeNode`, позволяющий связать создаваемое окно с родительским, а также уникальное значение хендла. Пример.

```
// Создание окна
// ws — установленная сессия RWSession,
// wg — объект RWindowGroup
RWindow wnd(ws);
User::LeaveIfError(wnd.Construct(wg, TInt32(&wnd)));
```

В качестве родителя может выступать не группа окон, а другое окно (объект классов `RBlankWindow`, `RWindow` или `RBackedUpWindow`). В этом случае следует помнить о том, что на экране отображается только область окна потомка, не выходящая за рамки родительского окна. Окна, имеющие одного родителя, могут перекрывать друг друга. Их порядок отображения регулируется с помощью метода `SetOrdinalPosition()`. В качестве уникального идентификатора окна рекомендуется использовать адрес управляющего им объекта.

Сразу после создания окно не отображается и не получает сообщений о перерисовке содержимого до тех пор, пока не будет вызван метод `Activate()`. Это позволяет разработчику задать его характеристики (размер, положение, прозрачность и пр.) с помощью следующей группы методов.

- `SetPosition()` — задает положение относительно родительского окна. Началом отсчета системы координат служит левый верхний угол. В случае когда родительским окном является группа окон (объект `RWindowGroup`), положение определяется относительно левого верхнего угла экрана. В процессе работы определить местоположение окна можно с помощью методов `Position()` (относительно родительского окна) и `AbsPosition()` (относительно экрана).
- `SetSize()` — задает размер окна. Получить текущий размер окна можно с помощью метода `Size()`.
- `SetExtent()` — позволяет одновременно задать положение и размер окна.

- `SetBackgroundColor()` — позволяет изменить цвета фона окна. Окно заливается этим цветом при перерисовках, инициированных со стороны сервера.
- `EnableRedrawStore()` — включает или отключает механизм Redraw Store. По умолчанию включен. Данный механизм реализуется сервером в целях оптимизации и позволяет ему запоминать операции, производящиеся при перерисовке окна. Отключение механизма Redraw Store приводит к некорректному отображению содержимого окон, перекрытых полупрозрачными окнами.
- `SetRequiredDisplayMode()` — устанавливает режим отображения для окна. Режим отображения влияет на качество цветопередачи. По умолчанию используется самый лучший из поддерживаемых аппаратным обеспечением. Определить режим отображения окна можно с помощью метода `DisplayMode()`.
- `SetShape()` — позволяет изменить форму окна.
- `SetCornerType()` — позволяет изменить форму углов окна.
- `SetTransparentRegion()`, `SetNonTransparent()` и ряд методов `SetTransparencyXXX()` — позволяют определить параметры прозрачности окна.
- `SetShadowDisabled()` — включает и отключает отображение тени для самого верхнего окна (по умолчанию отображается). Тень представляет собой затемненную область, по размеру и форме равную окну и находящуюся позади него. За отрисовку тени отвечает система.
- `SetShadowHeight()` — задает смещение между окном и его тенью. По умолчанию оно равно нулю, и тень не видна.
- `FadeBehind()` — включает и отключает затемнение всех окон той же оконной группы, когда данное окно оказывается самым верхним. Проверить, включена ли данная возможность, можно с помощью метода `IsNonFading()`, а для того чтобы определить, затемнено ли окно в данный момент, необходимо воспользоваться методом `IsFaded()`.

Все вышеперечисленные методы могут вызываться и в процессе работы для динамического изменения характеристик окна. Для включения и отключения отображения окна после его активации методом `Activate()` служит функция `SetVisible()`.



Полупрозрачные окна поддерживаются далеко не всеми устройствами. На данный момент ни одна модель под управлением S60 3, 3 FP1 и 3 FP2 не реализует эту возможность, а методы `SetTransparencyXXX()` возвращают ошибку `KErrNotSupported`.

После того как окно активировано, разработчик должен обеспечить получение и обработку уведомлений о необходимости его перерисовки. Для этого служит метод `RedrawReady()` класса `RWsSession`. Вызванный с его помощью асинхронный запрос завершается в тот момент, когда содержимое окна необ-

ходимо частично или полностью перерисовать. После его завершения должен вызываться метод `GetRedraw()`, помещающий в переданный ему по ссылке объект `TWsRedrawEvent` дополнительные сведения. Класс `TWsRedrawEvent` содержит хендл окна, содержимое которого необходимо перерисовать (полезно в том случае, если вы создали несколько окон для одной сессии `RWsSession`), и регион, который должен быть перерисован. Отменить асинхронный запрос `RedrawReady()` можно с помощью метода `RedrawReadyCancel()`.

Следующий пример демонстрирует фрагменты кода класса, схожего с классом `CKeyCatcher` активного объекта, обеспечивающего создание окна `RWindow` и обработку событий.

```
class CDrawer : public CActive
{
    <...> // Объявление стандартных методов активного объекта
private:
    RWsSession iWServ;
    RWindowGroup* iWg;
    // Окно для отображения графики
    RWindow * iWindow;
}

void CDrawer::ConstructL()
{
    <...> // Установка сессии с файловым сервером
    <...> // Создание группы окон

    // Создание окна
    iWindow = new (ELeave) RWindow (iWServ);
    User::LeaveIfError(iWindow->Construct(*iWg,
                                         (TUint32)&iWindow));

    // Установка местоположения и размера
    iWindow->SetExtent(TPoint(10, 10), TSize(100, 100));
    // Установка цвета фона окна (белый)
    iWindow->SetBackgroundColor(TRgb(255, 255, 255));
    // Отображение небольшой тени
    iWindow->SetShadowHeight(1);

    // Активация окна
    iWindow->Activate();
}

void CDrawer::StartL()
{
    Cancel();
    // Регистрация на уведомления о перерисовках
    iWServ.RedrawReady(&iStatus);
    SetActive();
}
```

```

    }

void CDrawer::RunL()
{
    if (iStatus == KErrNone)
    {
        // Получение информации о событии
        TWsRedrawEvent event;
        iWServ.GetRedraw(event);
        // event.Handle() – идентификатор окна
        // event.Rect() – область, которая должна быть перерисована
        <...> // Выполняем перерисовку
        // Регистрируемся на следующее сообщение
        iWServ.RedrawReady(&iStatus);
        SetActive();
    }
}

void CDrawer::DoCancel()
{
    iWServ.RedrawReadyCancel();
}

CDrawer::~CDrawer()
{
    Cancel();

    if (iWindow)
    {
        iWindow->Close();
        delete iWindow;
    }
    <...> // Удаление iWg и закрытие iWServ
}

```

Во время выполнения программы вызвать перерисовку всего окна или его части можно самостоятельно. Для этого используется один из перегруженных методов `Invalidate()` класса `RWindow`. Данный метод сообщает серверу окон, что содержимое окна устарело и нуждается в обновлении. Сервер в свою очередь пошлет соответствующее сообщение указанному окну. Таким образом, весь код для вывода графики на экран всегда должен находиться в обработчике результатов запроса `RedrawReady()`. А сами сообщения о перерисовке окна принято делить на **инициированные сервером** (*server-initiated redraw*), посланные в результате переключения пользователем приложений в своем устройстве и прочих внешних причин, и **инициированные клиентом** (*client-initiated redraw*), вызванные созданием окна приложением с помощью метода `Invalidate()`.

Итак, теперь мы можем создавать окна нужного размера и получать уведомления о необходимости перерисовки их содержимого, но, как видите, в методе `RunL()` отсутствует код, который мог бы что-либо нарисовать в окне. Далее мы подробно рассмотрим использующийся для этого механизм.

Прежде всего, необходимо создать объект класса `CWsScreenDevice`, предоставляющий доступ к экрану устройства. В его конструктор передается ссылка на сессию сервера окон. Затем следует вызвать метод `Construct()`, завершающий создание объекта. Данный метод также имеет перегруженный вариант, принимающий номер экрана устройства, к которому подключается объект. Класс `CWsScreenDevice` предоставляет большое количество методов для получения информации о различных характеристиках экрана (размер, палитра, ориентация), работы со шрифтами и доступа к отдельным пикселям. Он также позволяет сохранить изображение всего экрана в файл. Но нас в данный момент интересует лишь один метод — `CreateContext()`. С его помощью создается объект класса `CWindowGc`. Данный класс используется для работы с графическим контекстом и содержит инструменты для рисования на экране.

Следующий пример демонстрирует подготовку объектов `CWsScreenDevice` и `CWindowGc`.

```
// Создание объекта CWsScreenDevice
CWsScreenDevice* screen = new (ELeave)
                                CWsScreenDevice(iWServ);
CleanupStack::PushL(screen);

// Подключение к экрану по умолчанию
screen->Construct();

// Создание графического контекста
CWindowGc* gc;
User::LeaveIfError(screen->CreateContext(gc));
CleanupStack::PushL(gc);
<...> // Рисование на экране
CleanupStack::PopAndDestroy(2, screen);
```

После того как графический контекст создан, его необходимо активировать для обновляемого окна с помощью метода `Activate()`, принимающего ссылку на объект базового класса `RDrawableWindow`. В дальнейшем, по завершении работы с окном, графический контекст нужно либо деактивировать с помощью метода `Deactivate()` и использовать повторно для другого окна, либо уничтожить.

Затем необходимо вызвать метод `BeginRedraw()` объекта окна. Он уведомляет сервер, что окно собирается отреагировать на его последнее сообщение о необходимости обновления содержимого. Существует также перегруженный вариант метода `BeginRedraw()`, принимающий объект `TRect`, задающий область обновления. Получить его можно из объекта `TWsRedrawEvent`. После того как содержимое окна будет обновлено, нужно проинформировать об этом сервер `WSERV` с помощью вызова метода `EndRedraw()`.

Между вызовами методов `BeginRedraw()` и `EndRedraw()` объекта `RWindow` выполняется собственно вывод графики. Для этого используются следующие методы графического контекста класса `CWindowGc`.

- `SetOrigin()` — устанавливает начало относительной системы координат, используемой при рисовании. По умолчанию ее начало совпадает с левым верхним углом обновляемого окна.
- `SetClippingRegion()` — позволяет указать регион, для которого будут применяться операции рисования. Все нарисованное вне этой области игнорируется. По умолчанию совпадает с видимой частью окна.
- `SetPenColor()`, `SetPenSize()` и `SetPenStyle()` — задают цвет, толщину линии и стиль абстрактного карандаша. Данные параметры используются при рисовании линий.
- `MoveBy()` и `MoveTo()` — перемещают карандаш в относительной системе координат. Метод `MoveBy()` сдвигает его по заданному вектору, а метод `MoveTo()` переносит в указанную позицию.
- `DrawLine()`, `DrawLineBy()` и `DrawLineTo()` — позволяют рисовать прямые линии. Методы `DrawLineBy()` и `DrawLineTo()` рисуют линию от текущей позиции карандаша по заданному вектору либо до указанной точки в относительной системе координат соответственно. Метод `DrawLine()` служит для рисования линии между двумя заданными позициями. Все три метода изменяют текущее положение карандаша.
- `DrawPolyLine()` — рисует ломаную линию, последовательно соединяя множество точек, указанных в массиве.
- `DrawEllipse()` — рисует эллипс, вписанный в заданный прямоугольник.
- `DrawArc()` — позволяет рисовать дуги (часть эллипса).
- `DrawPie()` — рисует секторы эллипса (возможно закрашивание).
- `DrawRect()` и `DrawRoundRect()` — позволяют нарисовать прямоугольник. С помощью метода `DrawRoundRect()` его углы могут быть скруглены.
- `DrawPolygon()` — позволяет рисовать многоугольники (возможно закрашивание).
- `SetBrushColor()`, `SetBrushStyle()` и `SetBrushOrigin()` — служат для задания характеристик абстрактной кисти (цвет, стиль и смещение шаблона стиля соответственно), используемой для закрашивания областей.
- Перегруженные методы `Clear()` — позволяют закрашивать все окно или указанную область.
- `Reset()` — выполняет сброс всех параметров кисти и карандаша, заменяя их значениями по умолчанию.
- `SetOpaque()` — позволяет задать параметры прозрачности при рисовании, используется только при рисовании на полупрозрачном окне.
- `SetFaded()` и `SetFadingParameters()` — используются для затемнения окна.

- `SetDrawMode()` — позволяет определить, будет ли цвет текущего содержимого окна комбинироваться с цветом линий и заливки, и как это должно происходить.
- `MapColors()` — позволяет переназначить цвета для всех пикселей в заданной области окна. Соответствие цветов задается парами.

В качестве примера ниже приводится фрагмент кода, позволяющий нарисовать в окне случайный многоугольник на белом фоне. Результат выполнения этого кода представлен на рис. 6.13.

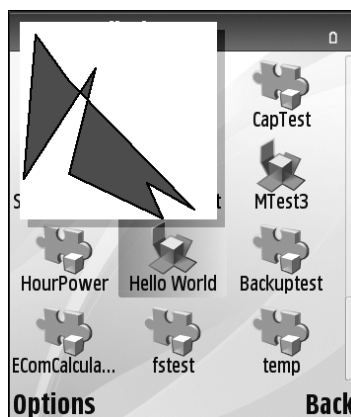


Рис. 6.13. Небольшое окно с тенью, отображающее красный многоугольник

```
// Активируем графический контекст окна
gc->Activate(*iWindow);
// Уведомляем сервер о начале рисования
iWindow->BeginRedraw(rect);

gc->SetPenSize(TSize(2,2)); // Размер карандаша: 2x2
gc->SetBrushColor(KRgbWhite); // Цвет кисти: белый
gc->SetBrushStyle(CGraphicsContext::ESolidBrush);
// Кисть: сплошная
gc->Clear(); // Очистка перерисовываемой области
gc->SetBrushColor(KRgbRed); // Цвет кисти: красный

// Помещаем в массив polygon до десяти случайных
// координат в пределах размера окна
// (для использования Random()) нужно подключить e32math.h
CArrayFixFlat<TPoint>* polygon =
    new (ELeave) CArrayFixFlat<TPoint>(10);
CleanupStack::PushL(polygon);
TInt cnt = Math::Random() % 10;
for (TInt i = 0; i < cnt; i++)
{
    TUint x(Math::Random());
```

```

    x %= event.Rect().Width();
    TUint y(Math::Random());
    y %= event.Rect().Height();
    polygon->AppendL(TPoint(x, y));
}

// Рисуем многоугольник
gc->DrawPolygon(polygon);
CleanupStack::PopAndDestroy(polygon);
// Уведомляем сервер об окончании рисования
iWindow->EndRedraw();

```



Для задания наиболее часто используемых цветов можно воспользоваться одним из шестнадцати псевдонимов `KRgbXXX` для объектов `TRgb`, объявленных в заголовочном файле `gdi.h`.

На практике для увеличения быстродействия объекты классов `CWScreenDevice` и `CWindowGc` не создают каждый раз заново при перерисовке, а объявляют в качестве членов класса и инициализируют вместе с окном в методе `ConstructL()`.



По умолчанию объект `RWsSession` использует буферизацию при передаче некоторых команд на сервер (например, команд рисования). Отправка всех хранящихся в буфере данных и его очистка может происходить как автоматически, так и принудительно при выполнении некоторых операций или при вызове метода `Flush()`. Буферизация команд может быть отключена с помощью метода `SetAutoFlush()`, однако в большинстве случаев разработчику нет необходимости вмешиваться в процесс передачи данных объектом сессии сервера.

Отображение текста

Symbian OS поддерживает два типа шрифтов: TTF и GDR (собственный формат Symbian). GDR-шрифт можно создать из файла BDF (формат шрифтов компании Adobe) с помощью утилиты `fntrtran.exe`, входящей в состав SDK. Подробную информацию о ней можно найти в справочнике SDK. Кроме того, в Symbian OS реализована подсистема Open Font System, позволяющая разрабатывать и подключать модули, обеспечивающие поддержку других форматов шрифтов.

Работу со шрифтами осуществляет специальный **сервер шрифтов и изображений** (Font and Bitmap Server: файл `fbserve.exe`, SID `0x10003A16`). Им осуществляется регистрация файлов шрифтов и их загрузка в память. Это позволяет всем приложениям, отображающим текст на экране одним и тем же шрифтом, использовать единожды помещенный в память экземпляр. Шрифт выгружается только тогда, когда его перестает использовать последнее приложение.

Функционирование сервера окон тесно связано с работой сервера шрифтов и изображений. Для удобства разработчика он дублирует наиболее часто ис-

пользуемые функции сервера шрифтов, выступая в качестве прослойки над ним. Благодаря этому необходимость в использовании сервера FBSErv напрямую возникает крайне редко.

Абстрактные базовые классы, предоставляющие интерфейс для работы со шрифтами, реализуются в разделяемой библиотеке `gdi.dll`. Их объявление находится в заголовочном файле `gdi.h`, а при использовании требуется подключение библиотеки импорта `gdi.lib`.

И эмулятор, и устройства под управлением Symbian OS всегда содержат некоторый набор предустановленных шрифтов. Кроме того, разработчик может установить собственные шрифты поддерживаемого формата, включив их файлы в дистрибутив приложения. Получить информацию обо всех зарегистрированных в системе и доступных для использования гарнитурах (шрифтов одного рисунка, но разных размеров (кеглей) и начертаний) можно через сервер окон с помощью класса `CWsScreenDevice`. Для этого в нем содержатся следующие методы.

- `NumTypefaces()` — возвращает количество известных системе гарнитур.
- `TypefaceSupport()` — позволяет получить информацию о гарнитуре с заданным индексом. Информация возвращается в объекте класса `TTypefaceSupport` (объявлен в файле `gdi.h`), предоставляются следующие характеристики:
 - `iIsScalable` — масштабируемость;
 - `iMinHeightInTwips` и `iMaxHeightInTwips` — минимальный и максимальный размер шрифта для данной гарнитуры в **твипах** (`twip`);
 - `iNumHeights` — количество различных размеров, которые может принимать шрифт данной гарнитуры;
 - `iTypeface` — объект `TTypeface`, содержащий символьное имя и атрибуты (пропорциональность, символьность, сериф) шрифтов гарнитуры.
- `FontHeightInPixels()` и `FontHeightInTwips()` — позволяют определить размер шрифта по индексу гарнитуры и номеру поддерживаемого ею размера в пикселях или твипах соответственно.
- Множество методов `GetNearestFontXXX()` — служат для получения указателя на базовый класс `CFont`, позволяющий работать с конкретным шрифтом гарнитуры. В качестве аргумента методов необходимо передать объект `TFontSpec`, инкапсулирующий характеристики (гарнитура, стиль, размер) искомого шрифта. Некоторые методы позволяют задать приблизительный размер и получить наиболее близкий к необходимому шрифт. В случае если подходящего шрифта в системе не существует, методы возвращают ошибку.
- `ReleaseFont()` — позволяет уведомить сервер шрифтов о том, что данный шрифт больше не используется клиентом. Единственным аргументом метода является указатель на объект базового класса шрифта `CFont`. Если данный шрифт больше не используется ни одним клиентом, то он выгружается из памяти.

- `AddFile()` — регистрация нового файла шрифтов в системе и получение его уникального идентификатора.
- `RemoveFile()` — отмена регистрации файла шрифтов по полученному из метода `AddFile()` идентификатору. В момент вызова ни один из шрифтов файла не должен использоваться.
- `GetFontById()` — получение указателя на объект базового класса `CFont` шрифта по его идентификатору. Такой идентификатор имеется у каждого шрифта в файлах BDF и GDR.



Единица измерения *twip* не зависит от характеристик устройства и равна одной двадцатой пункта (1/567 сантиметра, или 1/1440 дюйма), тогда как размер пикселя — аппаратно-зависимая величина. Соотношение размеров твипа и пикселя изменяется при различных разрешениях экрана.

Итак, получить список всех поддерживаемых гарнитур на устройстве и их размеров вам поможет следующий код.

```
// Создание и подключение объекта screen
// класса CWScreenDevice
TInt tf_cnt = screen->NumTypefaces();
TTypefaceSupport tf_supp;
for (TInt i = 0; i < tf_cnt; i++)
{
    screen->TypefaceSupport(tf_supp, i);
    // Определение названия гарнитуры
    TBuf<50> name = tf_supp.iTypeface.iName;
    for (TInt j = 0; j < tf_supp.iNumHeights; j++)
    {
        // Определение поддерживаемого размера шрифта
        TInt height = screen->FontHeightInTwips(i, j);
    }
}
```

После того как вы определились с гарнитурой и подходящим размером шрифта, его необходимо загрузить с помощью одного из методов `GetNearestFontXXX()`.

```
_LIT(KTypeFace, "Series 60 Sans");
TFontSpec fspec(KTypeFace, 220); // Размер: 220 twips
CFont* font;
User::LeaveIfError(screen->GetNearestFontInTwips(font,
                                                    fspec));

<...> // Работа с font
```

Класс `CFont` является абстрактным и предоставляет лишь базовый интерфейс для работы со шрифтом, но в большинстве случаев этого достаточно.

Полученный шрифт можно использовать для вывода текста на экран. Для этого его следует назначить текущим шрифтом графического контекста окна

с помощью метода `UseFont()` класса `CWindowGC`, после чего текст может быть отображен с помощью одного из описываемых ниже методов класса `CWindowGC`.

- `DrawText()` — рисование заданного текста текущим шрифтом в виде горизонтальной строки, начиная с заданной точки экрана. Существует также перегруженный вариант, позволяющий задать прямоугольную область окна, в которой должен отображаться текст, а также его выравнивание в ней. Эта область предварительно автоматически очищается с помощью операции `Clear()`.
- `DrawTextVertical()` — аналог метода `DrawText()`, позволяющий рисовать текст вертикально. Содержит дополнительный параметр булевого типа, регулирующий направление текста.
- `DrawTextExtended()` — вариант метода `DrawText()`, принимающий дополнительный параметр типа `CGraphicsContext::TDrawTextExtendedParam`, позволяющий задать некоторые характеристики отображения текста (например, направление).

Класс `CWindowGC` также предоставляет методы `SetStrikethroughStyle()` и `SetUnderlineStyle()` для вывода зачеркнутого или подчеркнутого текста, а также методы `SetCharJustification()` и `SetWordJustification()` для точного выравнивания символов и слов.

По умолчанию графический контекст не имеет назначенного шрифта для рисования текста, поэтому использование методов `DrawText()` и `DrawTextVertical()` без предварительного вызова метода `UseFont()` приведет к панике. По окончании вывода текста текущий шрифт контекста можно сбросить с помощью метода `DiscardFont()`.

Ниже приводится небольшой пример, позволяющий вывести слово `Symbian` на экран несколькими способами с помощью ранее полученного шрифта. Результат его выполнения представлен на рис. 6.14.

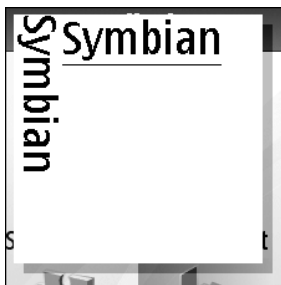


Рис. 6.14. Вывод текста в окно

```
_LIT(KText, "Symbian");
// Назначение шрифта контексту
gc->UseFont(font);
// Отображение вертикальной строки сверху вниз
```

```
gc->DrawTextVertical(KText, TPoint(font->DescentInPixels(),
                                   0), EFalse);
// Подчеркивание включено
gc->SetUnderlineStyle(EUnderlineOn);
// Отображение горизонтальной строки
TInt h = font->HeightInPixels();
gc->DrawText(KText, TPoint(h, h));
// Освобождение шрифта
gc->DiscardFont();
screen->ReleaseFont(font);
```

Работа с изображениями

Графический контекст содержит ряд методов для вывода изображений на экран, принимающих в качестве аргументов объекты класса `CFbsBitmap`. Класс `CFbsBitmap` предназначен для создания новых или загрузки существующих изображений из файлов формата MBM. При этом для их размещения используется память сервера шрифтов и изображений. Класс `CFbsBitmap` инкапсулирует сессию `RFbsSession` этого сервера, которую он устанавливает и завершает автоматически. Сервер отвечает за управление выделенной для хранения изображений кучей (в том числе, за ее дефрагментацию), а также отслеживает количество работающих с каждым из них клиентов. Изображение удаляется из памяти только тогда, когда его перестанет использовать последний объект `CFbsBitmap`. Таким образом, не только экономится память, но и достигается высокая производительность при отображении графики на экране. Ведь серверу окон достаточно получить от клиентского приложения хендл изображения в памяти сервера `FBSErv`, которое он желает отобразить, а не все его данные.



Загрузить изображение в память сервера `FBSErv` из MIF-файлов можно средствами статического класса `AknIconUtils`. Для получения изображений из файлов формата JPG, GIF и прочих популярных растровых форматов необходимо воспользоваться классом `CImageDecoder`. Кроме того, разработчик имеет возможность создать модуль для добавления поддержки новых форматов.

Класс `CFbsBitmap` объявлен в заголовочном файле `fbs.h`, и для его использования необходимо подключить библиотеку импорта `fbscli.lib`. Он предоставляет множество методов, среди которых чаще всего используются следующие.

- `Create()` — создание нового пустого изображения с заданным размером и режимом отображения (цветовой гаммой).
- `Load()` — загрузка из указанного MBM-файла изображения с заданным индексом. Последний параметр типа `TBool` позволяет определить, может ли данный экземпляр изображения в памяти быть использован другими клиентами либо им придется создавать собственную копию.

- `Duplicate()` — подключение объекта `CFbsBitmap` к уже находящемуся в памяти изображению с заданным хендлом.
- `Compress()` — уведомление сервера о возможности проведения сжатия памяти, занимаемого изображением (например, с помощью RLE). Существует также асинхронный вариант метода — `CompressInBackground()`.
- `LoadAndCompress()` — аналогичен вызовам методов `Load()` и `Compress()`.
- `IsCompressedInRAM()` — проверка, хранится ли изображение в памяти в сжатом виде.
- `IsLargeBitmap()` — позволяет узнать, считает ли сервер FBSEVR изображение большим (обычно при размере больше 64 Кбайт). При хранении больших изображений сервер применяет специальные механизмы оптимизации использования памяти.
- `SizeInPixels()` и `SizeInTwips()` — получение размеров изображения в пикселях и твипах соответственно. Изменить эти параметры можно с помощью методов `SetSizeInTwips()` и `Resize()`. Размеры в пикселях и твипах являются независимыми значениями и могут быть конвертированы методами `VerticalPixelsToTwips()`, `VerticalTwipsToPixels()` и `HorizontalPixelsToTwips()`, `HorizontalTwipsToPixels()`.
- `DisplayMode()` и `SetDisplayMode()` — получение и изменение режима отображения. Режим отображения может только ухудшаться по отношению к значению, использованному при создании объекта. Уточнить изначальный режим можно с помощью метода `InitialDisplayMode()`.
- `IsMonochrome()` — проверка, является ли изображение черно-белым.
- `GetPixel()` — получение цвета пикселя в заданной точке изображения.
- `GetScanLine()`, `SetScanLine()`, `GetVerticalScanLine()` и `GetVerticalScanLine()` — получение и изменение цвета целой линии пикселей изображения (горизонтальной или вертикальной).
- `Save()` и `StoreL()` — сохранение изображения или нескольких изображений в MBM-файл.
- `Reset()` — отключение объекта `CFbsBitmap` от изображения на сервере. Его хендл обнуляется. Если это был единственный клиент, использовавший изображение, то оно будет выгружено из памяти. Объект `CFbsBitmap` может быть использован повторно.

Несмотря на то, что класс `CFbsBitmap` имеет методы для получения и изменения целых линий пикселей изображения, при выполнении значительных преобразований (например, поворотов или масштабирования) рекомендуется использовать более эффективный вспомогательный класс `TBitmapUtil`. Он объявлен в том же заголовочном файле, что и `CFbsBitmap`, и принимает указатель на его объект в конструкторе. Класс `TBitmapUtil` позволяет обращаться к отдельным пикселям и выполняет операции быстрее. Все операции должны выполняться между вызовами методов `Begin()` и `End()`, уведомляющими сервер о начале и окончании работы с изображением. В этот период сервер гаран-

тирует, что данные изображения не будут перемещаться в памяти (например, при дефрагментации). Это позволяет избежать лишних проверок и тем самым увеличить производительность.

Для того чтобы рисовать на изображении так же как и в окне, существуют классы `CFbsBitmapDevice` и `CFbsBitGc`, использование которых во многом похоже на работу с классами `CWsScreenDevice` и `CWindowGc`. Они реализованы в библиотеке `bitgdi.dll`, объявлены в файле `bitstd.h` и требуют подключения к проекту файла `bitgdi.lib`. Следующий пример демонстрирует загрузку изображения из MBM-файла и отрисовку на нем рамки.

```
const TInt KImageIdInMBM = 0;
_LIT(KMBMFile, "z:\\HelloWorld.mbm");

// Создание объекта CFbsBitmap
CFbsBitmap* bitmap = new (ELeave) CFbsBitmap();
CleanupStack::PushL(bitmap);

// Загрузка изображения из MBM-файла в память сервера FBSERV
User::LeaveIfError(bitmap->Load(KMBMFile, KImageIdInMBM));

// Создание графического устройства на основе изображения
CFbsBitmapDevice* bdev = CFbsBitmapDevice::NewL(bitmap);
CleanupStack::PushL(bdev);

// Создание графического контекста
CFbsBitGc* bgc = CFbsBitGc::NewL();
CleanupStack::PushL(bgc);

// Подключение графического контекста к устройству
bgc->Activate(bdev);
// Рисуем красную рамку на изображении
bgc->SetPenColor(KRgbRed);
bgc->SetPenSize(TSize(10,10));
bgc->DrawRoundRect(TRect(bitmap->SizeInPixels()), TSize(50,50));
// Уничтожаем объекты, используемые для рисования
CleanupStack::PopAndDestroy(2, bdev);
<...> // Продолжаем работу с bitmap
```

После того как изображение будет подготовлено к отображению, его можно будет вывести в окно во время обработки его перерисовки. Для этого класс `CWindowGc` предоставляет следующие методы.

- `BitBlt()` — рисование изображения или его части в заданной точке окна.
- `BitBltMasked()` — аналог метода `BitBlt()`, позволяющий указать еще одно черно-белое изображение, выступающее в роли маски для задания областей прозрачности.
- `DrawBitmap()` — рисование изображения или его части в заданной точке окна в соответствии с его размером в твипах (величинах, не зависящих от

характеристик экрана). Если размер в твипах объекта `CFbsBitmap` не задан, то изображение не будет отображено. Существует также перегруженный метод, позволяющий нарисовать изображение или его часть в заданной прямоугольной области окна. При этом осуществляется его масштабирование.

- `DrawBitmapMasked()` — аналог метода `DrawBitmap()`, позволяющий указать изображение-маску для задания областей прозрачности.

Вот небольшой пример использования ранее подготовленного изображения (масштабируется в соответствии с размерами окна) — результат его выполнения представлен на рис. 6.15.

```
// rect — перерисовываемая область окна
iWindow->BeginRedraw(rect);
gc->Clear();
gc->DrawBitmap(rect, bitmap);
iWindow->EndRedraw();
```



Рис. 6.15. Рисование изображения в окне



MBM-файл создается с помощью утилиты `bmconv` в составе SDK. Чтобы автоматизировать работу с `bmconv`, запись о MBM-изображении можно добавить в MMP-файл — в этом случае он будет генерироваться при сборке проекта. IDE Carbide.c++2.x предоставляет удобный мастер для создания и редактирования таких записей, вызываемый с помощью команд `Add MBM Entry` или `Edit MBM` в контекстном меню MMP-файла в окне дерева проекта.

Другие возможности сервера окон

Сервер окон предоставляет несколько API, которые подробно в этой книге не рассматриваются, но могут быть очень полезны. Их использование не должно вызывать сложностей у читателя, освоившего ранее изложенный материал, поэтому далее я привожу лишь их краткое описание.

Приложения, требующие высокой производительности при отображении графики, могут получить прямой доступ к экрану (в англоязычной литературе часто используется акроним DSA (direct screen access)) с помощью классов `CDirectScreenAccess` или `RDirectScreenAccess`. Это позволяет минимизировать использование сервера окон, а все, что вы нарисуете, будет отображаться

немедленно. Принципиальная разница между классами `CDirectScreenAccess` и `RDirectScreenAccess` заключается в способе получения уведомлений от сервера окон — в частности, сообщений о принудительном прекращении прямого доступа к области экрана (например, при ее перекрытии окном другого приложения). Для связи с сервером в классе `CDirectScreenAccess` используется интерфейс методов обратного вызова `MDirectScreenAccess`, а в классе `RDirectScreenAccess` — асинхронный запрос.

Помимо окон, сервер `WSERV` позволяет отображать на экране **спрайты** (sprite), управляемые с помощью классов `RWsSprite`. *Спрайтом* является изображение (или группа изображений), которое может перемещаться по экрану, не вызывая перерисовку окон под ним. На основе спрайтов может быть создана простая анимация, а также разнообразные курсоры.

При реализации анимации разработчику необходимо использовать таймеры, основанные на активных объектах. Но как вы помните, их срабатывание может происходить с задержкой, связанной с недостаточно высоким приоритетом. Устранить этот недостаток можно с помощью полиморфной DLL специального вида, загружаемой непосредственно в процесс сервера окон. Это позволит реализованному в ней объекту, порожденному от базового класса `CAnim`, использовать таймеры высокоприоритетного сервера `WSERV` и отображать анимацию в окне или в спрайте. Связь с такой библиотекой осуществляется с помощью клиент-серверной архитектуры.



Помимо вывода анимации, полиморфные библиотеки `CAnim` могут использоваться для перехвата нажатий клавиш. Они получают подобные сообщения в виде объектов `TRawEvent` от сервера окон, но до того как он сам их обрабатывает. Такой способ перехвата нажатий сложнее реализовать, но он лишен недостатков методов `CaptureXXX()`. В том случае, если библиотека не заблокирует событие, то оно будет обработано сервером и отправлено по назначению (окну с фокусом).

Примеры использования вышеназванных API можно найти в Википедии сообщества `Forum Nokia` и справочнике SDK.

ГЛАВА 7

Сертификация приложений

Безусловно, изложенная в этой книге информация не может быть достаточна для создания коммерческого ПО. В ней представлены лишь базовые знания по Symbian C++, необходимые начинающему программисту. Поэтому пройдет еще длительное время, потраченное на исследования, разработку и тестирование, прежде чем вам удастся получить приложение, которое заслуживало бы сертификации и публикации. Тем не менее я считаю, что некоторые сведения о процессе сертификации должны быть известны разработчику изначально. Дело в том, что программист должен еще до начала работы над проектом оценить, доступ к каким защищенным возможностям потребуется приложению. Ведь от этого может зависеть стоимость, и даже возможность его реализации. Кроме того, разработчик должен знать критерии, предъявляемые к приложению при сертификации. Некоторые требования к безопасности и устойчивости программы могут серьезно повлиять на первоначальное ТЗ, и опытный разработчик вносит такие корректировки до начала, а не во время работы над проектом.

С другой стороны, не представляется возможным дать исчерпывающую информацию о прохождении сертификации. Дело в том, что предъявляемые требования и сама ее процедура периодически претерпевают изменения. Меняться может многое: появляются новые программы сертификации, изменяются цены, список центров тестирования, совершенствуются критерии. Однажды сменилась даже организация, выдающая идентификаторы издателя. Поэтому процесс сертификации будет изложен в довольно обобщенном виде, и вы должны иметь в виду, что эта информация могла устареть. Лучше всего либо перепроверить ее перед началом проекта (если вы довольно редко пишете программы для Symbian OS), либо следить за происходящими изменениями в сообществе Symbian Foundation (если вы занимаетесь разработкой профессионально).

Способы сертификации

Сертификация заключается в подписывании приложения специальным цифровым сертификатом, содержащим информацию об авторе приложения и разрешающим доступ к определенным защищенным возможностям Symbian OS. Более подробно о том, что такое защищенные возможности и как они проверяются, рассказывалось в *главе 1*, раздел “Платформа безопасности Symbian OS”. Здесь мы лишь напомним, что они делятся на четыре типа (табл. 7.1).

Таблица 7.1 . Категории защищенных возможностей

Пользовательские возможности	LocalServices Location NetworkServices ReadUserData UserEnvironment WriteUserData	Наиболее значимые для владельца телефона функции. В зависимости от политики безопасности производителя устройства, пользователи могут самостоятельно давать приложению разрешение на доступ к этим возможностям
Системные возможности	PowerMgmt ProtServ ReadDeviceData SurroundingsDD SwEvent TrustedUI WriteDeviceData	Защищают сервисы системы, настройки устройства и некоторые функции аппаратного обеспечения
Ограниченные возможности	CommDD DiskAdmin NetworkControl MultimediaDD	Защищают файловую систему, коммуникационные и сервисы мультимедиа устройства
Возможности производителя устройств	AllFiles DRM TCB	Защищают наиболее важные сервисы системы

Мы уже упоминали два вида сертификатов, которые можно получить бесплатно: сертификат-пустышка (self-generated) и сертификат разработчика (DevCert).

Сертификат-пустышку разработчик может создать с помощью инструментов SDK (процесс описан в *главе 3*, раздел “Подписывание SIS-файла сертификатом”) и применять для сертификации приложения, не использующего защищенные возможности либо ограничивающегося доступом к пользовательской категории возможностей. При установке подписанного этим типом сертификатов дистрибутива система выводит предупреждающее сообщение о том, что такому приложению нельзя доверять. Многие интернет-магазины программного обеспечения отказываются от распространения приложений с сертификатом self-signed, опасаясь некачественных и вредоносных программ. В связи с этим многие разработчики предпочитают использовать полноценные платные программы сертификации и подписывать дистрибутивы более доверенным сертификатом даже в том случае, если для их установки достаточно сертификата типа self-generated.

Что же касается сертификата разработчика, то при его использовании дистрибутив также выдает предупреждения во время установки. Кроме того, он ограничивает список устройств, на которые возможна установка, по их IMEI-номеру. Перечень защищенных возможностей, доступ к которым он может подтвердить, задается при его создании.

SIS-пакет также может распространяться неподписанным, но в этом случае забота о его сертификации ложится на плечи пользователя. Конечно же, такое ПО не бывает коммерческим.

Сертификацию приложений осуществляет компания Symbian Signed, являющаяся частью Symbian Foundation. Ее портал расположен по адресу <http://www.symbiansigned.com>. Она предлагает различные способы сертификации (табл. 7.2 и 7.3).

- **Open Signed** — позволяет бесплатно подписать SIS-пакет сертификатом разработчика, действительным в течение трех лет. Есть две версии этой программы: Open Signed Online и Open Signed Offline.
 - Вариант **Online** доступен всем желающим и позволяет подписать пакет для одного номера IMEI, подтверждая доступ к группе пользовательских и системных защищенных возможностей. Коды UID всех исполняемых файлов в пакете должны быть из тестового диапазона значений (0xE0000000... 0xEFFFFFFF) либо быть зарезервированы пользователем. Процедура подписания сертификатом происходит на сайте (разработчик загружает неподписанный SIS-пакет и получает подписанный по почте). Скачать сам файл сертификата разработчика для последующего использования нельзя.
 - Вариант **Offline**, в отличие от Online, доступен только владельцам идентификатора издателя (Publisher ID, PubId). Он заключается в отправке на сайт Symbian Signed специально сформированного запроса, на основании которого генерируется файл сертификата разработчика. Этот файл затем можно скачать и использовать для подписывания дистрибутивов до истечения срока его действия сколько угодно раз. Это не единственное преимущество Offline-варианта. В запросе на создание сертификата можно указать список IMEI-номеров устройств (до 1000 номеров и даже больше, если связаться с Symbian Signed), на которых будут устанавливаться подписанные им дистрибутивы. Это очень удобно. Кроме того, такой сертификат разработчика может подтвердить доступ ко всем защищенным возможностям, кроме группы возможностей производителя. В отличие от Online-варианта программы, значения UID всех исполняемых файлов в пакете должны быть из защищенного диапазона значений.
- Программы **Express Signed** и **Certified Signed** позволяют сертифицировать дистрибутив сертификатом, не накладывающим ограничения на IMEI-номера устройств, и действительным в течение десяти лет. Они доступны только владельцам идентификатора издателя. Кроме того, они являются платными. Оплачивается каждая сертификация SIS-файла. Дистрибутив после этого может копироваться и распространяться как угодно. Но если вы подготовите новую версию пакета (или измените существующую), то его вновь придется сертифицировать и, соответственно, платить за это. Сертификация по программе Certified Signed (от 160 EUR, стоимость зависит от выбранного центра тестирования) существенно дороже Express Signed (20\$).
 - **Express Signed** — быстрый и экономный способ сертификации, подходящий для большинства приложений. Накладывает ограничения на возможность доступа к некоторым защищенным возможностям, но проходит без обязательного независимого тестирования.

- **Certified Signed** — программа сертификации приложения, позволяющая ему получить доступ ко всем функциям устройства, не требующим разрешения производителя. Требуется обязательного прохождения процедуры независимого тестирования. Приложения, сертифицированные при помощи программы Certified Signed, могут использовать логотип “for Symbian OS” в рекламных целях.

Доступ ко всем вышеперечисленным программам, кроме Open Signed Online, доступен только владельцам **идентификатора издателя** (Publisher ID, PubId). Это цифровой сертификат сроком действия 1 год, который может приобрести только юридическое лицо или ПБОЮЛ. Срок действия идентификатора издателя определяет период, на протяжении которого владелец может воспользоваться соответствующими программами сертификации, но никак не влияет на срок действия сертификата, которым подписывается дистрибутив. Он предназначен для идентификации автора дистрибутива. Сертифицировавшее SIS-файл юридическое лицо всегда можно легко определить. Это является сдерживающим фактором для тех, кто желает каким-то образом сертифицировать вредоносный код. Обычным наказанием в этом случае является занесение идентификатора издателя в черный список и прекращение обслуживания его владельца. Стоимость сертификата издателя составляет около 200\$.

Таблица 7.2. Характеристики различных программ сертификации Symbian Signed

	Open Signed Online	Open Signed Offline	Express Signed	Certified Signed
Требуется наличия идентификатора издателя	Нет	Да	Да	Да
Бесплатна	Да	Да ¹	Нет	Нет
Независимое тестирование	Нет	Нет	Нет	Да
Ограничения по IMEI	Да	Да	Нет	Нет
Подходит для коммерческого распространения	Нет	Нет	Да	Да

Таблица 7.3. Доступные способы сертификации при использовании защищенных возможностей различных категорий

	Self-signed	Open Signed Online	Open Signed Offline	Express Signed	Certified Signed
Пользовательские возможности	Да	Да	Да	Да	Да
Системные возможности	Нет	Да	Да	Да	Да
Ограниченные возможности	Нет	Нет	Да	Нет	Да
Возможности производителя устройств	Нет	Нет	Да ²	Нет	Да ³

¹ Доступна только после покупки идентификатора издателя.

² Требуется подтверждение от производителя устройств.

³ Требуется подтверждение от производителя устройств.

Open Signed Online

Для того чтобы подписать SIS-файл сертификатом разработчика по программе Open Signed Online, необходимо на сайте `symbiansigned.com` перейти по соответствующей ссылке⁴. Регистрация учетной записи не требуется. В открывшейся в окне браузера форме (рис. 7.1) следует указать IMEI-номер устройства, на которое вы планируете устанавливать дистрибутив, ваш адрес электронной почты, выбрать загружаемый с ПК SIS файл, а также ввести код теста Тьюринга и согласиться с лицензионным соглашением.

Рис. 7.1. Форма Open Signed Online

Номер IMEI можно узнать, введя значение ***#06#** на экране смартфона. При регистрации в форме указывать следует только один номер. В SIS-пакете не должно содержаться исполняемых файлов, декларирующих доступ к защищенным возможностям из категории ограниченных или возможностей производителей устройств. Используемые идентификаторы должны быть из незащищенного диапазона (0xE0000000...0xFFFFFFFF). Исключением является случай, когда вы имеете учетную запись на сайте, идентификатор издателя и зарегистрировали за собой некоторый диапазон номеров UID из защищенной области значений. В этом случае указываемый в форме адрес электронной почты должен совпадать с адресом в учетной записи, на которую выделены идентификаторы.

Если все эти условия соблюдены, то, после того, как вы щелкнете в форме на кнопке **Send**, выбранный SIS-файл будет загружен на сайт Symbian Signed, а через некоторое время вы получите его подписанную версию на указанный в форме почтовый адрес. Его можно использовать для установки приложения на одно устройство в течение трех лет с момента сертификации.

⁴ <https://www.symbiansigned.com/app/page/public/openSignedOnline.do>

Покупка идентификатора издателя

Цифровые сертификаты, называемые идентификаторами издателя, выпускаются **центрами сертификации** (certificate authority) и содержат информацию о своем владельце. Ими подписываются запросы на получение сертификата разработчика по программе Open Signed Offline и SIS-файлы перед отправкой на сертификацию по программам Express Signed и Certified Signed. Сам по себе идентификатор издателя не подтверждает доступа приложения к каким-либо защищенным возможностям.

В настоящее время центром сертификации, выпускающим идентификаторы издателя для Symbian Signed, является TC TrustCenter. Приобрести его можно за 200\$ (без НДС), перейдя по следующей ссылке: <http://www.trustcenter.de/order/publisherid/dev>. Оплата осуществляется при помощи кредитной карты. Мы не будем подробно рассматривать заполнение различных форм данными о компании. На этом этапе браузер (рекомендую Internet Explorer), который вы используете для заполнения форм, генерирует пару встраивающихся в него ключей. В дальнейшем получить идентификатор издателя можно будет только при помощи этого браузера. Убедитесь, что в момент получения идентификатора издателя будете иметь доступ к тому ПК и браузеру, с которого была выполнена его покупка. Операционную систему и браузер на данном ПК в этот период переустанавливать нельзя.

Сразу после оплаты идентификатора издателя на указанный вами в данных о покупателе почтовый адрес придет письмо с подтверждением факта покупки и перечнем документов, отсканированные копии которых вы должны предоставить центру сертификации для подтверждения существования юридического лица. Обычно этими документами являются заверенная выписка из ЕГРЮЛ и паспорт осуществившего покупку сотрудника. Сотрудники TC TrustCenter предпочитают получать эти данные с электронного почтового адреса с доменного имени, принадлежащего проверяемой компании.

После того как центр сертификации убедится в существовании юридического лица, на ваш почтовый адрес будет отправлено письмо со ссылкой для установки идентификатора издателя в браузер. Открывать ее следует только в браузере, содержащем сгенерированные ключи. После установки сертификата в браузер вам следует экспортировать его вместе с закрытым ключом в формате PKCS#12 в файл с расширением “.pfx”. При этом необходимо задать пароль, который впоследствии будет использоваться при доступе к идентификатору издателя.

Затем из PFX-файла необходимо сделать пару файлов: KEY (приватный ключ в формате PKCS#8) и CER (сертификат в формате X.509), подходящих для использования с утилитой `signsis.exe`. Для этого необходимо скачать утилиту TC-ConvertP12⁵, поместить PFX-файл в ее папку и запустить файл `tcp12p8.bat` из командной строки следующим образом.

⁵ http://developer.symbian.org/wiki/index.php/Symbian_Signed_Tools#Publisher_ID_export_tool

```
tsrpl2p8.bat <название pfx-файла> <пароль к pfx-файлу>
mykeyfile.key mycerfile.cer
```

Полученные ключ и сертификат должны использоваться для подписания SIS-файлов перед отправкой на сертификацию по программам Express Signed и Certified Signed. Чтобы получить доступ к программам Open Signed Offline, Express Signed и Certified Signed, необходимо связаться с сотрудниками Symbian Signed и сообщить им имя вашей учетной записи на сайте symbiansigned.com и данные сертификата (содержимое CER-файла). Обратиться к ним можно на форуме Symbian Signed Support портала сообщества разработчиков Symbian⁶.

Open Signed Offline

Чтобы получить сертификат разработчика по программе Symbian Signed Offline, необходимо создать специальный файл запроса формата PKCS#10 с расширением “.csr”, подписанный идентификатором издателя. Он генерируется при помощи утилиты DevCertRequest (текущая версия 2.3), скачать которую можно с сайта сообщества разработчиков Symbian⁷. После установки и запуска утилиты DevCertRequest откроется окно мастера, состоящего из пяти этапов.

1. Выберите папку и имя для создаваемого CSR-файла.
2. Укажите CER- и KEY-файлы вашего идентификатора издателя, а также пароль к нему.
3. На третьем этапе мастер отображает информацию о владельце, полученную из идентификатора издателя. Убедившись, что выбран правильный идентификатор, перейдите к следующему этапу.
4. На данном этапе необходимо ввести номера IMEI, для которых будет действителен сертификат разработчика, и указать защищенные возможности, доступ к которым он должен подтверждать (рис. 7.2). Список IMEI может содержать до 1000 номеров. Предоставляются функции его сохранения и восстановления из файла. В качестве удостоверяемых защищенных возможностей я советую выбирать все, кроме возможностей производителя. Так как программа Open Signed Offline является бесплатной, то нет никаких причин ограничиваться при запросе сертификата разработчика лишь теми защищенными возможностями, которые используются программой. Поэтому, получив сертификат сразу для 17 защищенных возможностей, вы с большей долей вероятности сможете использовать его в дальнейшем для других проектов.

⁶ <http://developer.symbian.org/forum/forumdisplay.php?f=5>

⁷ http://developer.symbian.org/wiki/index.php/Symbian_Signed_Tools#DevCertRequest

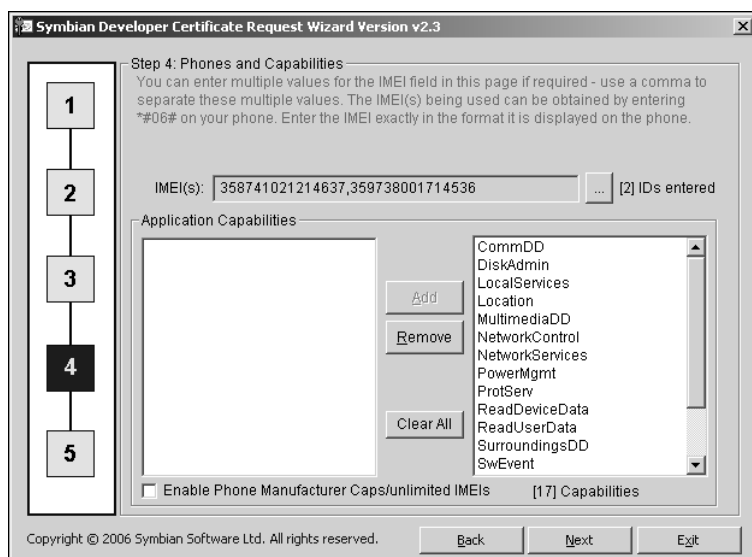


Рис. 7.2. Мастер DevCertRequest — четвертый этап

- На последнем этапе отображается вся информация, собранная на предыдущих этапах. Удостоверившись, что все верно, щелкните на кнопке **Finish** для генерации файла запроса.

Получив CSR-файл запроса, войдите в свою учетную запись на сайте symbiansigned.com и на вкладке **My Symbian Signed** откройте страницу **Open Signed\Request**. В ней укажите ваш CSR-файл и, введя код теста Тьюринга, щелкните на кнопке **Send**. В том случае если код введен верно и CSR-файл сформирован правильно, вы будете немедленно переадресованы на страницу, отображающую информацию о выданном вам сертификате разработчика. Скачать его CER-файл можно при помощи кнопки **Download**. Этот файл должен использоваться в паре с KEY-файлом вашего идентификатора издателя. В дальнейшем вы всегда сможете просмотреть список всех выданных вам сертификатов разработчика на странице **Open Signed⇒My DevCerts** (рис. 7.3) и скачать любой из них снова.



Рис. 7.3. Информация о полученном сертификате

Резервирование идентификаторов

В приложениях, сертифицируемых по программам Express Signed и Symbian Signed, должны использоваться значения идентификаторов из защищенного диапазона значений. Это позволяет не только быстро определить создателя приложения, но и обеспечить уникальность его идентификаторов.

Разработчик может бесплатно запросить некоторое количество идентификаторов из защищенного диапазона. Для этого необходимо войти в свою учетную запись на сайте symbiansigned.com и на вкладке **My Symbian Signed** перейти к окну **UIDs⇒Request** (рис. 7.4). В нем указывается количество идентификаторов, которое вам необходимо, название вашей компании, ваш электронный почтовый адрес, а также (необязательно) произвольное описание. Во избежание злоупотреблений разработчик может запросить не более 20 номеров в день. Если это ограничение вас не устраивает — свяжитесь с сотрудниками Symbian Signed, и они сами выделяют вам необходимое количество идентификаторов. В качестве названия организации должно использоваться значение, указанное при получении идентификатора издателя.

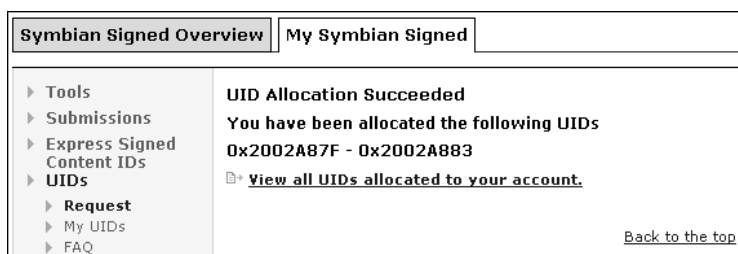


Рис. 7.4. Назначенный диапазон идентификаторов

Если все поля заполнены верно, то после отправки запроса вы немедленно будете переадресованы на страницу, отображающую выделенный вам диапазон значений из защищенного диапазона. В дальнейшем вы всегда можете просмотреть все назначенные вашей учетной записи идентификаторы в окне **UIDs⇒My UIDs**.

Критерии тестирования Symbian Signed

Перед прохождением сертификации по программам Express Signed и Symbian Signed должно проводиться тестирование работы приложения в соответствии с официально утвержденными Symbian Signed критериями. Они спроектированы для того, чтобы убедиться, что все программы сторонних разработчиков, написанные для Symbian OS, достигают минимального уровня качества. На текущий момент Symbian Signed использует критерии версии 3.0.3, но уже готовится к публикации версия 4.x. Поэтому я настоятельно советую найти на

сайте symbiansigned.com ссылку на текущую версию этого документа⁸ и внимательно с ним ознакомиться. Далее излагаются лишь общие положения критериев тестирования.

Тестовые испытания не направлены на субъективную оценку или проверку содержания приложения: удобство в использовании, локализацию или стиль графического интерфейса. Также не тестируется работоспособность на различных моделях. Приложение проверяется только на тех моделях, которые указаны разработчиком, либо в PKG-файле.

Тесты делятся на две главных группы:

- универсальные тесты (с префиксом UNI в названии) — для проверки надежности и устойчивости приложения;
- тесты на обращение к защищенным возможностям (с префиксом CAP) — проверяют правильность работы с некоторыми специфичными функциями.

Примеры универсальных тестов:

- приложение должно корректно устанавливаться (в том числе с внешних накопителей, например, карт памяти), удаляться, переустанавливаться и не оставлять в устройстве никаких файлов после своего удаления;
- если в приложении реализована возможность резервного копирования, то оно должно корректно восстанавливаться;
- приложение должно использовать для создания файлов предназначенные для этого каталоги;
- приложение должно проходить нагрузочные тесты, обрабатывать системные исключения (например, нехватку памяти), внезапное выключение устройства и многократные переключения между программами;
- приложение должно корректно обрабатывать системные события и правильно работать с диспетчером задач;
- приложение должно корректно отображаться вне зависимости от разрешения и ориентации экрана устройства;
- приложение, реализующее механизм автозапуска при старте системы, обязано предоставлять способ его включения и отключения.

Примеры тестов на обращение к защищенным возможностям:

- приложения мультимедиа не должны мешать выполнению голосовых звонков;
- приложения, выполняющие переадресацию звонков, обязаны иметь графический интерфейс;
- приложения интернет-телефонии должны отображать сообщение “Об отказе производителя устройства от ответственности” и не мешать GSM-функциям телефона, включая возможность выполнения экстренного вызова.

⁸ На данный момент действительные критерии тестирования публикуются здесь: http://developer.symbian.org/wiki/index.php/Symbian_Signed_Test_Criteria_%28Symbian_Signed%29

Большинство критериев имеет несколько схем проведения тестирования. Кроме того, они содержат многочисленные нюансы, а также исключения. Например, невозможно проверять корректность закрытия программы через диспетчер задач, если она в этом диспетчере не отображается.

Программа сертификации Express Signed

Программа сертификации Express Signed является платной. Каждое подписание SIS-файла стоит 1 Content ID. Content ID является своеобразным виртуальным талоном на прохождение сертификации. Для того чтобы определить, сколько Content ID на данный момент доступно для вашей учетной записи, нужно войти в нее на сайте symbiansigned.com и на вкладке **My Symbian Signed** перейти на страницу **Express Signed Content IDs**. Там же можно пополнить текущий счет, приобретя 1, 5, 10, 50 или 100 единиц Content ID. Каждая единица Content ID стоит 20\$ и оплачивается по системе PayPal или кредитной картой.

После того как на счету вашей учетной записи появится хотя бы один Content ID, вам станет доступна страница **Submissions**⇒**Express Signed**. При переходе на нее открывается первое окно мастера сертификации.

1. В первом окне указывается ZIP-архив, содержащий сертифицируемое приложение, и информация о его авторе. В ZIP-архиве должны содержаться:
 - SIS-пакет приложения, подписанный идентификатором издателя;
 - PKG-скрипт, на основании которого был создан данный SIS-файл;
 - TXT-файл с кратким описанием программы, особенностей ее установки и работы;
 - в том случае, если сертифицируемое приложение является условно-бесплатным, вы должны предоставить несколько ключей, для того чтобы ее могли зарегистрировать во время тестирования.
2. Следующее окно — описание приложения. Указываются его название, версия, тип (игра, персональное и т.д.), краткое описание, лицензия (коммерческое, бесплатное, демонстрационное и т.д.), язык приложения (можно выбрать несколько), язык программирования, использованный при создании. Вам также необходимо указать одну из моделей смартфонов, поддерживаемых приложением. Именно эта модель устройства будет использована для тестирования в том случае, если оно понадобится.
3. В третьем окне приводятся результаты тестирования. Предполагается, что вы самостоятельно протестировали ваше приложение согласно критериям Symbian Signed и на этом этапе должны указать его результаты (пройдено/не пройдено/исключение). В том случае, если по какому-либо критерию к приложению применимо исключение — вы должны указать его код.

4. Последнее окно содержит обоснование использования защищенных возможностей. На этом этапе вам предстоит для каждой декларируемой в SIS-файле защищенной возможности указать используемое в приложении API, требующее доступа к ней, а также (в некоторых случаях) пояснить необходимость использования этого API.

После того как все этапы работы мастера будут пройдены, выполняется сертификация вашего приложения. Идентификатор издателя, которым оно было подписано, при этом удаляется. Сертификация может занять несколько минут, а после ее окончания вам предложат скачать свой ZIP-архив с уже подписанным SIS-пакетом. В дальнейшем вы всегда сможете получить доступ ко всем ранее сертифицированным пакетам на странице **Submissions⇒My applications**.

Периодически компания Symbian Signed проводит аудит сертифицированных по программе Express Signed приложений. Пакеты для проверки выбираются случайным образом. В том случае, если вы заполнили формы мастера заведомо неверными данными, обманули с результатами самостоятельного тестирования или необоснованно наделили приложение доступом к некоторым защищенным возможностям, то к вам будут применены штрафные санкции. Жесткость этих санкций зависит от серьезности вашего нарушения. Например, сертификация вредоносной программы в лучшем случае закончится отзывом идентификатора издателя и его занесением в черный список. При менее серьезных проступках вам запретят пользоваться возможностями Express Signed до тех пор, пока вы несколько раз не пройдете программу Certified Signed с обязательной проверкой в независимом центре тестирования.

Программа сертификации Certified Signed

Программа сертификации Certified Signed принципиально отличается от Express Signed тем, что при ее прохождении проводится обязательная проверка приложения в независимом центре тестирования в соответствии с текущими критериями Symbian Signed. На данный момент Symbian Signed работает с тремя центрами тестирования: NSTL, MphasiS и Sogeti HT. Они отличаются по стоимости предоставляемых услуг и срокам (обычно 3-4 дня), в течение которых выполняется тестирование приложения. Ознакомиться с условиями оказания услуг различных центров тестирования можно, войдя в свою учетную запись на сайте **Symbian Signed** и перейдя на страницу **Test houses** на вкладке **Symbian Signed Overview**.

Виртуальные талоны Content ID для оплаты услуг центров тестирования не используются. Установленная сумма должна перечисляться вами выбранному центру напрямую.

Мастер для отправки приложения на сертификацию по программе Certified Signed находится на странице **Submissions⇒Certified Signed** вкладки **My Symbian Signed** и, в целом, аналогичен мастеру Express Signed. Разница между ними лишь в том, что на втором этапе вам необходимо будет дополнительно

выбрать в раскрывающемся списке центр тестирования, который будет проверять ваше приложение, а этап три отсутствует.

После завершения работы мастера ваш ZIP-архив с подписанным идентификатором издателя SIS-пакетом, PKG-файлом и описанием отправляется выбранному на втором этапе центру тестирования. Дальнейшее общение с сотрудниками центра и оплата их услуг выполняется напрямую через сайт центра тестирования.

Отслеживать состояние поданного на сертификацию приложения можно на странице **Submissions⇒My applications** сайта symbiansigned.com. Через несколько дней, по окончании всех проверок, центр тестирования самостоятельно свяжется с Symbian Signed, и статус вашей заявки изменится на **Accepted** (в случае успешного прохождения тестов) или **Rejected** (приложение не соответствует критериям). Здесь же можно будет скачать подписанный SIS-файл. Кроме того, центр тестирования предоставит вам документ с отчетом о проведенной проверке и ее результатами.

В том случае если центр тестирования счел приложение не соответствующим действующим критериям, вам ничего не останется, кроме как ознакомиться с его отчетом, внести необходимые исправления и снова подать заявку на прохождение сертификации по программе Certified Signed. Услуги центра тестирования придется оплачивать еще раз.

ПРИЛОЖЕНИЕ А

Акронимы и сокращения

AO (Active object) — активный объект в Symbian C++.

GUI (Graphical User Interface) — графический интерфейс пользователя.

DBMS (Data Base Management System) — система управления базой данных (СУБД).

DDL (Data Definition Language) — язык описания данных, подмножество языка SQL, предназначенное для построения DDL-запросов, связанных с определением структуры сохраняемых в базе данных.

DLL (Dynamic-Link Library) — динамически подключаемая библиотека.

DML (Data Manipulation Language) — язык управления данными, подмножество языка SQL, предназначенное для построения DML-запросов, связанных с процедурами манипуляции данными (вставка, удаление или изменение данных в базе).

GDI (Graphical Device Interface) — один из компонентов Symbian OS, реализующий операции с графикой, графическим контекстом и изображениями.

IDE (Integrated Development Environment) — интегрированная среда разработки.

ITC (Inter Thread Communication) — межпоточное взаимодействие.

IPC (Inter Process Communication) — межпроцессное взаимодействие.

MBM (Multi-Bitmap File) — формат файла, способного содержать несколько изображений.

MIF (Multi-Icon File) — формат файла, способного содержать несколько пиктограмм.

MIME (Multipurpose Internet Mail Extension) — стандарт, описывающий передачу различных типов данных по электронной почте. Включает в себя спецификацию наиболее часто используемых типов данных.

POSIX (Portable Operating System Interface for Unix) — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой. Стандарт создан для обеспечения совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода, но может быть использован и для не-Unix-систем.

P.I.P.S. (P.I.P.S. Is POSIX on Symbian) — портированные библиотеки POSIX для Symbian OS.

RAM (Random-Access Memory) — память, позволяющая получить доступ к участку данных за постоянное время независимо от его физического расположения в ней. Чаще всего — оперативная память.

ROM (Read-Only Memory) — энергонезависимая память, используемая для хранения неизменяемых данных.

SDK (Software Development Kit) — комплект инструментов и средств разработки.

SMS (Short Message Service) — технология, позволяющая осуществлять прием и передачу коротких текстовых сообщений в сетях сотовой связи.

SQL (Structured Query Language) — язык структурированных запросов, применяемый для создания, модификации и управления данными в реляционных базах данных.

SVG (Scalable Vector Graphics) — язык разметки масштабируемой векторной графики, созданный консорциумом W3C и входящий в подмножество расширяемого языка разметки XML. Предназначен для описания двухмерной векторной и смешанной векторно/растровой графики в формате XML.

TTF (TrueType Font) — шрифт стандарта TrueType.

Twip (Twentieth of a Point) — типографская единица измерения, равная одной двадцатой пункта (1/567 сантиметра, или 1/1440 дюйма). Соотношение размеров твипа и пикселя изменяется в зависимости от разрешения экрана устройства.

W3C (World Wide Web Consortium) — консорциум, разрабатывающий и внедряющий технологические стандарты для WWW.

WSD (Writable Static Data) — область памяти, содержащая изменяемые глобальные данные DLL.

XIP (eXecute In Place) — метод выполнения программы напрямую с накопителя, без загрузки в оперативную память.

ПРИЛОЖЕНИЕ Б

Справочные материалы

Документация

Основы операционной системы Symbian

Официальный подготовительный курс к экзамену ASD

Stichbury J., Jacobs M. *The Accredited Symbian Developer Primer: Fundamentals of Symbian OS*. 2006. ISBN 9780470058275.

История Symbian OS

Litchfield S. *The History of Psion*. — *Palmtop Magazine*, 1998.
<http://3lib.ukonline.co.uk/historyofpsion.htm>

Архитектура Symbian OS

Morris B. *The Symbian OS Architecture Sourcebook. Design and Evolution of a Mobile Phone OS*. 2007. ISBN 9780470018460.

Sales J. *Symbian OS Internals. Real-time Kernel Programming*. — Symbian Press. 2005. ISBN 0470025247.

http://developer.symbian.org/wiki/index.php/Symbian_OS_Internals

Morris B. *What's New in the System Model for Symbian OS v9.4?*. Version 1.0. 2008.
http://developer.symbian.com/main/downloads/papers/What%27s_new_in_v9.4_system_model.pdf

Symbian Developer Network *Symbian OS v9.1 System Model*. Issued 2.1.
<http://developer.symbian.com/main/downloads/files/public%20v9.1%20ISSUED-2.1.pdf>

Sales J. *Demand Paging on Symbian*. 1st edition. 2009. ISBN 9781907253003.
http://developer.symbian.org/wiki/index.php/File:Demand_Paging_on_Symbian_online.pdf

Платформа безопасности

Heath C. *Symbian OS Platform Security: software developing using the Symbian OS security architecture*. — Symbian Press. 2006. ISBN 0470018828.

Symbian Software Limited *Platform security for all*. 2008.

http://developer.symbian.com/main/documentation/books/books_files/pdf/Plat+Sec+FINAL+-+WEB.pdf

Shackman M. *Platform Security — a Technical Overview*. Version 1.2.

http://developer.symbian.com/main/downloads/papers/plat_sec_tech_overview/platform_security_a_technical_overview.pdf

Работа с SDK

EMCC Software Ltd. *An Introduction to the Symbian OS Tool Chain*. 2008.

http://developer.symbian.com/main/downloads/papers/Symbian_OS_Tool_Chain_Overview.pdf

S60 Platform: Identification Codes Version 2.1.

http://www.forum.nokia.com/info/sw.nokia.com/id/61ae01cb-3c34-47f6-843e-485d4f56409b/S60_Platform_Identification_Codes.html?language=english

IDE Carbide.c++

Carbide.c++ v1.3. 2008. На русском языке, 1-я редакция.

http://developer.symbian.com/main/documentation/books/books_files/pdf/Carbide_Russian_FINAL.pdf

Carbide Online Help Center v2.0.

<http://carbidehelp.nokia.com>

On-Device Debugging with Carbide.c++. Version 1.1. 2008.

http://www.forum.nokia.com/info/sw.nokia.com/id/1bdbfff2-c190-4558-9c17-facc9e247be4/Carbide_cpp_On_Device_Debugging_v1_1_en.pdf.html

Основы Symbian C++

Stichbury J. *Symbian OS Explained: Effective C++ Programming for Smartphones*. 2004. ISBN 0470021306.

Harrison R., Shackman M. *Symbian OS C++ for Mobile Phones*. Vol. 3. 2007. ISBN 0470066415.

Обработка исключений, стек очистки

Morris B. *Exception Handling in Symbian OS*. Version 1.2. 2008.

http://developer.symbian.com/main/downloads/papers/Exception_Handling_in_Symbian_OS-v1.02.pdf

Morley J. *Leaves and Exceptions*. Version 1.3.1. 2007.

<http://developer.symbian.com/main/downloads/papers/Leaves%20and%20Exceptions.pdf>

Stichbuty J. *Don't Panic! Application Crashes Explained*. Symbian Developer Network C++ Code Clinic, November 2008.

<http://developer.symbian.com/wiki/x/GABKBQ>

Stichbuty J. *Can I Leave in a Constructor?* Symbian Developer Network C++ Code Clinic, June 2008.

http://developer.symbian.com/main/support/code_clinic/clinic_june2008/

Stichbuty J. *Can I Leave in a Destructor?* Symbian Developer Network C++ Code Clinic, July 2008.

http://developer.symbian.com/main/support/code_clinic/clinic_july2008/

T-классы

Basca S. *An Introduction to T Classes*. Version 1.0. 2009.

<http://developer.symbian.com/main/downloads/papers/TClasses.pdf>

C-классы

Penrillian *C Classes — an Overview*. Version 1.0. 2009.

http://developer.symbian.com/main/downloads/papers/C_Classes.pdf

R-классы

Basca S. *An Introduction to R Classes*. Version 1.09. 2009.

http://developer.symbian.com/main/downloads/papers/R_Classes_v1.09.pdf

M-классы

Penrillian *An Introduction to M Classes*. Version 1.0. 2009.

http://developer.symbian.com/main/downloads/papers/M_Classes.pdf

Stichbuty J. *Mixin Inheritance and the Cleanup Stack*. Symbian Developer Network C++ Code Clinic, May 2008.

http://developer.symbian.com/main/support/code_clinic/clinic_may2008/index.jsp

Дескрипторы

Stichbury J. *FAQ по дескрипторам*.

http://wiki.forum.nokia.com/index.php/FAQ_по_дескрипторам

Stichbury J. *Использование дескрипторов: советы*.

http://wiki.forum.nokia.com/index.php/Использование_дескрипторов:_советы

Gusev A. *Symbian OS Descriptors Cookbook*. Version: 1.0. 2008.

http://developer.symbian.com/main/downloads/papers/descriptors_cookbook/Descriptors+Cookbook.pdf

Shackman M. *Introduction to RBuf*. Version 1.0.

http://developer.symbian.com/main/downloads/papers/RBuf/Introduction_to_RBuf_v1.0.pdf

L-классы

Shackman M. *An Introduction to L Classes*. Version 1.2. 2009.

<http://developer.symbian.com/main/downloads/papers/LClasses.pdf>

EUserHL для Symbian 9.x v1.2 (июнь 2009).

http://developer.symbian.org/wiki/index.php/EUserHL_Core_Idiom_Library

Динамические массивы

Todd P. *Advanced RArray*. Version 1.0. 2008.

http://developer.symbian.com/main/downloads/papers/Advanced%20RArray/advanced_RArray.pdf

Активные объекты

Моррис Б. *CActive и Все-Все-Все*. Версия 1.9. 2007.

http://www.devmobile.ru/docs/cactive_and_friends_v1.9_russian.pdf

McDowall I. *Long Running Active Object: a Design Pattern*. Version 1.0. 2008.

<http://developer.symbian.com/main/downloads/papers/Long+Running+Active+Object.pdf>

Stichbuty J. *Active Objects Example Code on the Couch*. Symbian Developer Network C++ Code Clinic, August 2008.

http://developer.symbian.com/main/support/code_clinic/clinic_aug2008/

Разработка приложений

Stichbury J. *Symbian OS Explained: Effective C++ Programming for Smartphones*. 2004. ISBN 0470021306.

Harrison R., Shackman M. *Symbian OS C++ for Mobile Phones*. Vol. 3. 2007. ISBN 0470066415.

Создание библиотек

Willee H. *Symbian Platform Support for Writeable Static Data in DLLs*. 2008.

http://developer.symbian.org/wiki/index.php/Symbian_Platform_Support_for_Writeable_Static_Data_in_DLLs

Stichbuty J. *WSD and the Singleton Pattern*. 2008.

http://developer.symbian.org/wiki/index.php/WSD_and_the_Singleton_Pattern

Механизм ECom

S60 Platform: ECom Plug-in Architecture. Version 2.0. 2007.

http://www.forum.nokia.com/info/sw.nokia.com/id/53a369e8-14c7-4f52-9731-577db4e0d303/S60_Platform_ECom_Plug-in_Architecture_v2_0_en.pdf.html

S60 Platform: ECom Plug-In Examples v2.0.

http://www.forum.nokia.com/info/sw.nokia.com/id/75ae7bde-7401-490f-87ec-920e44f518c2/S60_Platform_ECom_Plug-in_Examples_v2_0_en.zip.html

Распознаватели

Демонстрационный проект *S60 Platform: Document Handler Example*.

http://www.forum.nokia.com/info/sw.nokia.com/id/3e6c5f1f-1fb3-4c04-96ad-40979c7f2665/S60_Platform_Document_Handler_Example_v1_0_en.zip.html

Межпроцессное взаимодействие

Data Sharing and Persistence. 2009. 1st edition.

http://developer.symbian.org/wiki/index.php/Data_Sharing_and_Persistence_with_Symbian_C++

Клиент-серверная архитектура приложений

Roe J. *Separated Client-Server: A Design Pattern*. Version 1.0. 2008.

http://developer.symbian.org/wiki/index.php/Separated_Client-Server:_A_Design_Pattern

Willee H., Thoelke A., Taylor A. *Transient Server Template*. Version 1.3. 2006.

http://developer.symbian.org/wiki/index.php/Transient_Server_Template

Демонстрационное клиент-серверное приложение Hercules и другие примеры из книги Jo Stichbuty *Symbian OS Explained: Effective C++ Programming for Smartphones*.

<http://www.meme-education.com/ASDBookSOSEcode.html>

Базы данных

Data Sharing and Persistence. 2009. 1st edition.

http://developer.symbian.org/wiki/index.php/Data_Sharing_and_Persistence_with_Symbian_C++

S60 Platform: Using DBMS APIs. Version 2.0. 2006.

http://www.forum.nokia.com/info/sw.nokia.com/id/e0a66f34-092a-4a52-8003-6bbc3aa83c8f/S60_Platform_Using_DBMS_APIs_v2_0_en.pdf.html

S60 Platform: DBMS Example v2.1.

http://www.forum.nokia.com/info/sw.nokia.com/id/9fb467ab-61a9-4672-92f0-8f713d17466f/S60_Platform_DBMS_Example_v2_1_en.zip.html

Сокеты

Campbell I. *Symbian OS Communications Programming*. 2nd edition. 2007. ISBN: 9780470512289.

Сертификация приложений

Complete Guide To Symbian Signed.

http://developer.symbian.org/wiki/index.php/Complete_Guide_To_Symbian_Signed

Symbian Signed Test Criteria.

http://developer.symbian.org/wiki/index.php/Symbian_Signed_Test_Criteria_%28Symbian_Signed%29

Ссылки

Symbian Developer Community.

<http://developer.symbian.org/>

Forum Nokia.

<http://www.forum.nokia.com/>

Forum Nokia Russia.

<http://www.forum.nokia.ru>

Сообщество разработчиков DevMobile.

<http://devmobile.ru/>

Active Perl v6.5.1.x

<http://downloads.activestate.com/ActivePerl/Windows/5.6/>

IDE Carbide.c++ v2.x.

http://www.forum.nokia.com/Resources_and_Information/Tools/IDEs/Carbide.c++/

IVT BlueSoleil 6.4.x.

<http://www.bluesoleil.com/download/>

Inkscape v0.47.

<http://www.inkscape.org/download/?lang=ru>

TC-ConvertP12.

http://developer.symbian.org/wiki/index.php/Symbian_Signed_Tools#Publisher_ID_export_tool

DevCertRequest v2.3.

http://developer.symbian.org/wiki/index.php/Symbian_Signed_Tools#DevCertRequest

Предметный указатель

A

ActivePerl 13
ASCII-кодировка 173
ASCII-строка 173
Assertions 136

C

C-класс 149
 двухфазное
 конструирование 150
Carbide.c++ 13
 анализ исходного кода 109
 запуск 96
 приложения 111
 импорт проекта 101
 интерфейс 97
 конфигурация 112
 навигационные панели 103
 обновление 122
 отладка
 на устройстве 116
 приложения 112
 очистка и заморозка
 проекта 110
 перспектива 97
 сборка
 SIS-пакета 115
 проекта 110
 создание проекта 98
 установка 96
Certified Signed 448
CSR-файл 444

D

DDL 450
DEF-файл 73, 240
DevMobile 16
DML 450

E

EPOC 19
Express Signed 447

F

Feature Pack 64
Forum Nokia 16, 63
 Википедия 16

I

IDE 450
 Carbide.c++ 211
IMEI 39

L

L-класс 181

M

M-класс 155
 методы обратного
 вызова 156
MAKE-файл 70
MIF-файл 227
MIME 450
MIME-тип 300
MMP-файл 43, 44, 70, 80, 138,
 198, 212, 223, 287, 367

N

Nokia PC Suite 13

O

Open Signed Offline 443
Open Signed Online 441
Oracle Database Lite 367

P

PKG-файл 83, 115
 опции 88
Product UID 83
PubId 440

R

R-класс 152
 ресурсы 153
 хендл 152
RAM-память 29
Resolver 287, 298
ROM-память 29

S

Samsung Mobile Innovator 16
SDK 451
 API 66
 выбор 63
 для текущей работы 67
 для S60 63
 компиляция
 приложений 68
 работа с эмулятором 73
 сборка проекта 70
 состав 65
 установка 64
SID 30, 34
SIS-дистрибутив 71
SIS-пакет 81, 219, 438

встроенные пакеты 90
добавление файлов 85
заголовок 81
локализации 81
сборка 115
создание 91
установка 94

SIS-файл 80
 подписывание 93
 создание 91
Software Development Kit 63
Sony Ericsson Developers
 World 16

Symbian Developer Commu-
 nity 16

Symbian Developer Net-
 work 16

Symbian Foundation 16, 20

Symbian Ltd. 21

Symbian OS 11, 12

 Carbide.c++ 95

 SDK 63

 работа с

 изображениями 432

Symbian Press 15

Symbian Signed

 тестирование 445

SymbianC++

 C-классы 149

 L-классы 181

 M-классы 155

 R-классы 152

 T-классы 147

 активные объекты 192

 базы данных 367

 библиотеки импорта 240

 время и дата 300

 дескрипторы-буферы 162

 дескрипторы-пакеты 179

 дескрипторы-указатели 167

 дескрипторы 158, 177

 динамические

 массивы 183

 библиотеки 235

 использование консоли 219

 классы 127

 кодировки 173

 константы 126

 куча 138

 ловушки 130, 131

- макросы 126
 - массивы
 - фиксированные 191
 - окна 405
 - паника 134
 - перечисления 128
 - потоки данных 350
 - сбросы 130
 - серверы 267
 - соглашение об
 - именовании 126
 - создание
 - библиотек 234
 - приложения 211
 - статические библиотеки 234
 - стек 138, 139
 - очистки 140
 - строковые константы 160
 - структура проекта 41
 - структуры 128
 - типы данных 124
 - управление
 - ресурсами 152
 - памятью 137
 - утверждения 136
 - функции 129
 - хранение строк в куче 169
 - хранилище данных 359
 - чтение и запись файлов 338
 - SymbianOS
 - СУБД 367
 - архитектура 23
 - базовые сервисы 25
 - библиотека
 - BAFL 26
 - CONE 28
 - User 25
 - версии 22
 - вывод текста 428
 - защищенные
 - возможности 437
 - имя файла 313
 - история 19
 - коммуникационные
 - сервисы 26
 - механизм ECom 286
 - паника 134
 - плагины 287
 - платформа безопасности 30
 - поток 246
 - процесс 246
 - ресурсы для
 - разработчика 16
 - серверы 265
 - сервер
 - C32 26
 - ESock 26
 - телефонии ETel 26
 - окон 404
 - сертификаты 38
 - система ECom 26
 - служба
 - FEP 28
 - СУБД 26
 - сокеты 390
 - таймер 303
 - уникальный
 - идентификатор 30
 - установка приложений 38
 - файловый сервер 25, 312
 - центральный
 - репозиторий 26
 - экранирование данных 36
 - SymbianResources 16
 - Symbian C++ 11, 14, 124
 - Symbian Ltd 20
- T**
- T-класс 147
 - методы 148
- U**
- UID 30, 35, 360
 - UID3 230
 - замена 231
 - Unicode 28, 173
 - Unicode-строка 346
 - Unicode-текст 173
- V**
- VID 30, 35
- W**
- WSD 245
- A**
- Автозапуск 232
 - Автостарт 232
 - Активный объект
 - планировщик 193
 - принцип работы 193
 - Асинхронная функция 193, 195
 - Асинхронный вызов 192
- Б**
- База данных 367
 - Oracle Database Lite 367
 - SQL-запрос 381, 383
 - Symbian SQL 367
 - встраиваемая 367
 - индексы 374
 - клиент-серверная 367
 - создание 368
 - таблицы 370
 - транзакции 388
 - чтение и запись данных 375
- Б**
- Библиотека
 - BAFL 26
 - CONE 28
 - SQLite 367
 - User 25
 - динамическая 235
 - загрузка 244
 - импорта 240
 - статическая 234
- В**
- Вывод графики 421
 - Вытесняющая
 - многопоточность 192
- Д**
- Двухфазное
 - конструирование 150
 - Дескриптор 158, 177
 - ASCII-кодировка 173
 - Unicode-текст 173
 - буфер 162
 - выбор типа 176
 - изменяемый 160
 - кодировки 173
 - неизменяемый 158
 - пакет 179
 - символьный 160
 - указатель 167
 - Динамическая
 - библиотека 235
- Е**
- Единица доверия 32
- З**
- Заморозка проекта 73, 110
 - Запуск сервера при установке
 - сессии 272
 - Защищенная возможность 31, 34, 437
 - возможности производителя
 - устройств 32
 - единица доверия 32
 - ограниченные
 - возможности 32
 - пользовательские

- возможности 31
системные возможности 31
- И**
- Идентификатор
безопасности 34
издателя 40, 440, 442
производителя 34
ресурса 56
- Изменяемые глобальные
данные 245
- Изображение 432
- К**
- Каталог
подкаталоги и файлы 325
- Класс
- CActive 196
 - CActiveScheduler 195, 196
 - CApaDataRecognizerType 299
 - CArrayX 184
 - CBase 149, 156
 - CBufStore 359
 - CCnvCharacterSetConverter 174
 - CCoeEnv 316
 - CConsoleBase 219
 - CDbColSet 372
 - CDbKey 373, 374
 - CDeltaTimer 312
 - CDictionaryFileStore 365
 - CDirectFileStore 360, 361
 - CDirectScreenAccess 435
 - CFbsBitmap 432
 - CFileMan 329
 - CFileStore 360
 - CFont 429, 430
 - CHeartbeat 310
 - CIdle 312
 - CImageDecoder 432
 - CleanupStack 141, 144
 - CnvUtfConverter 175
 - Console 219
 - CPeriodic 307
 - CPeriodic 308
 - CPermanentFileStore 360, 361
 - CPersistentStore 359
 - CSecureStore 360
 - CServer2 267, 274
 - CSession2 269
 - CStreamDictionary 361
 - CStreamStore 359
 - CTrapCleanup 141, 216
 - CWindowGc 428, 434
 - CWindowGC 431
 - CWsScreenDevice 425, 428, 429
 - HBufC 169
 - MStreamBuf 351
 - RArray 188
 - RBackedUpWindow 407
 - RBuf 169, 172
 - RChunk 261
 - RCondVar 255
 - RCriticalSection 259
 - RDbColWriteStream 380
 - RDbDatabase 368, 389
 - RDbNamedDatabase 369, 387
 - RDbRawSet 375
 - RDbStoreDatabase 368
 - RDbTable 375
 - RDbUpdate 390
 - RDbView 382, 383
 - RDrawableWindow 407, 425
 - REComSession 287
 - RFile 152, 338, 340, 342, 346, 349
 - RFileReadStream 353
 - RFileWriteStream 354
 - RFs 267, 316, 319, 323, 328
 - RHostResolver 396
 - RLibrary 244
 - RMemReadStream 356
 - RMemWriteStream 356
 - RMessage2 277, 278
 - RMsgQueue 262
 - RMutex 254
 - RPipe 265
 - RPointerArray 188
 - RProcess 247
 - RProperty 263
 - RReadStream 352
 - RSemaphore 258
 - RSessionBase 270
 - RSocket 398, 401, 403
 - RSocketServ 391
 - RStoreReadStream 360
 - RStoreWriteStream 365
 - RThread 195, 251
 - RTimer 200, 303
 - RWindow 407, 421, 424
 - RWindowBase 407
 - RWindowGroup 405, 407, 411
 - RWindowTreeNode 405
 - RWriteStream 352
 - RWsSession 407, 420
 - RWsSprite 436
 - TBuf 163
 - TBufC 163
 - TCleanupItem 142, 153
 - TDateTime 301
 - TDbCol 372
 - TDbKeyCol 374
 - TDbQuery 381
 - TDbWindow 383
 - TDes 160, 164
 - TDesC 159, 166, 168
 - TFileName 313
 - TFileText 346
 - TFindChunk 261
 - TFindMutex 254
 - TFindProcess 247
 - TFindServer 268
 - TFindThread 251
 - TFixedArray 191
 - TLex 180
 - TLitC 160
 - TParse 314
 - TParsePtr 316
 - TPckg 180
 - TPckgBuf 180
 - TPckgC 180
 - TProcessId 247
 - TProtocolDesc 392
 - TPtr 167
 - TPtrC 168
 - TRequestStatus 195
 - TSockAddr 396, 399
 - TSwizzle 366
 - TTime 300, 302
 - TWSEvent 409
 - User 195
 - CDir 325
- Кодировка 173
 конвертация 174
- Коммуникатор 11, 13
- Коммуникационные
 сервисы 26
- Компания
 Symbian Signed 439
- Компилятор 68
- Конвейер 265
- Конвертер svg2svgt 227
- Консоль 219
- Контроль версий файлов 105
- Конфликт имен 230
- Корневой поток 361
- Критическая секция 259
- Куча 138
 хранение строк 169

- Л**
 Лексический анализатор 180
 Ловушка 131
 Локализация 58
- М**
 Макрос
 __ASSERT_ALWAYS 136
 __ASSERT_DEBUG 136
 __UHEAP_MARK 139
 __UHEAP_MARK-END 139
 __UHEAP_MARK-ENDC 139
 _L() 161
 _LIT() 160
 TRAP 132
 TRAP_IGNORE 132
 TRAPD 132
 Массив 139
 динамический 183
 фиксированный 191
 Межпоточное
 взаимодействие 260
 Межпроцессное
 взаимодействие 260
 механизм Publish & Subscribe 263
 очереди сообщений 262
 разделяемые области
 памяти 260
 Меню приложений 221
 Метод обратного вызова 156
 Механизм
 ECom 286
 Publish & Subscribe 263
 Множественное
 наследование 155
 Модуль TRK 116
 отладка приложения 118
 подключение к
 компьютеру 117
 установка 116
 Мьютекс 254
- Н**
 Номер IMEI 39
- О**
 Окно 404
 группы окон 406
 корневое 405
 перерисовка 424
- Оператор
 << 357
 >> 357
 Отладка 112
 информация о сбое 135
 на устройстве 116, 121
 точки наблюдения 114
 точки останова 113
 Очередь сообщений 262
 Очистка проекта 80, 110
- П**
 Память
 участки 261
 Панель
 Breakpoints 114
 C/C++ Editor 106, 108
 Call Hierarchy 107
 Console 105, 110
 Debug 113
 Modules 114
 Outline 106
 Platform Security 110
 Project Explorer 103, 110,
 151, 211, 212, 229
 Registers 114
 Symbian Project Navigator 104
 SymbianOS Data 114
 Type Hierarchy 107
 Variables 113
 Паника 134
 Пиктограмма
 приложения 226
 Плагин 287
 Планировщик 193, 195
 Платформа 21
 MOAP 21, 22
 S60 16, 21, 22
 S80 20
 S90 20
 UIQ 16, 20, 22
 UIQ3 20
 безопасности 30
 уровни доверия 33
 пакет обновлений 22
 редакции 22
 Портал
 Forum Nokia 16
 NewLC 16
 Symbian Developer Community 16
 SymbianResources 16
- Поток 137, 246, 251
 главный 246
 данных 350
 хранилище 359
 чтение 351
 межпоточное
 взаимодействие 260
 свойства 263
 синхронизация 254
 Приватный каталог
 процесса 319
 Приложение
 автостарт 232
 конфликт имен 230
 сертификация 437, 439
 Примесь 155
 Проект
 MMP-файл 44
 заморозка 73
 очистка 80
 работа с файлами 105
 создание дистрибутива 80
 структура подкаталогов 41
 файлы ресурсов 50
 файл bld.inf 41
 Протокол 392
 HSDPA 20
 WiFi802.11 20
 характеристики 395
 Процесс 137, 246
 App Installer 232
 приватный каталог 319
 Пул слотов сообщений 276
- Р**
 Регистрация программы 221
 Резервирование
 идентификаторов 445
 Ресурсы проекта 52
 Ресурс 50
 идентификатор 56
 локализация 58
 объявление 54
 структура 52
- С**
 Сборка
 SIS-пакета 115
 компонента
 MMP-файл 44
 проекта 70, 110
 Сборочный скрипт 42
 Сброс 130, 133
 стек очистки 141

- Свойство 263
 - издатель 263
 - подключение 264
 - подписчики 263
 - хендл 264
 - Семафор 258
 - Сервер 267
 - AppArcServer 298
 - Ecom
 - распознаватели 298
 - Resolver 298
 - ECom 287
 - Esock
 - подключения 404
 - ESock 26, 391
 - WSERV 404
 - СУБД 368
 - сессия 387
 - временный 266
 - запуск при установке
 - сессии 272
 - окон 404, 435
 - стандартные события 409
 - сессия 407
 - спрайты 436
 - остановка 274
 - передача сообщений 276
 - последовательный C32 26
 - постоянный 266
 - сессия 266
 - сокетов 391
 - протоколы 392
 - телефонии ETel 26
 - файловый 25, 312
 - шрифтов и
 - изображений 428
 - Сертификат-пустышка 39, 92, 438
 - Сертификат разработчика 39, 438
 - Сертификация на ASD
 - SymbianC++ 126, 130, 136, 140, 146, 148, 151, 155, 157, 159, 160, 162, 167, 169, 172, 175, 179, 181, 191, 192, 210
 - SymbianOS 30, 33, 36, 37, 40
 - работа с SDK 70, 73, 79, 91
 - разработка
 - приложений 246, 260, 265, 285, 300, 337, 350, 359, 366, 404
 - структура проекта 50, 62
 - Сертификация 35, 437
 - Certified Signed 448
 - Express Signed 447
 - приложений 12
 - резервирование
 - идентификаторов 445
 - способы 437, 439
 - Сессия 266
 - запуск сервера 272
 - на стороне клиента 270
 - на стороне сервера 269
 - субсессии 267
 - файлового сервера 316
 - Символы маски 315
 - Синхронизация потоков 254
 - критические секции 259
 - мьютекс 254
 - семафоры 258
 - условные переменные 255
 - Система ECom 26
 - Служба FEP 28
 - Службы СУБД 26
 - Смартфон 11, 13, 15
 - Создание
 - дистрибутива 80
 - каталогов 321
 - приложения 211
 - Сокет 390
 - адреса 396
 - заккрытие 402
 - операции 403
 - передача данных 401
 - создание 398
 - Сообщения
 - пул слотов 276
 - Спрайт 436
 - Статическая библиотека 234
 - Стек 138, 139
 - очистки 140, 154, 216
 - Строковая константа 160
- Т**
- Таймер 303
 - Твиш 429, 430, 451
 - Текст
 - шрифты 428
 - Точка
 - наблюдения 114
 - останова 113
 - Транзакция 388
- У**
- Уведомления о нажатиях клавиш 407
 - Уникальный
 - идентификатор 30, 35
 - Управление памятью 137
 - Условная переменная 255
 - Установка приложений 38
 - Утверждение 136
 - Утечка памяти 139
 - Утилита 66
 - abld freeze 73
 - bmconv 71, 435
 - Capability Scanner 109
 - CodeScanner 109
 - DevCertRequest 443
 - devices 68
 - elf2e32 71
 - elftran 71
 - fntran 428
 - make 42
 - makekeys 92
 - makesis 80, 91, 93
 - makmake 71
 - mifconv 227
 - nmake 42
 - rcomp 58
 - signsis 80, 93
 - SIS-Builder 118
 - Sisar 64
 - svgtbinencode 227
 - TC-ConvertP12 442
- Ф**
- Файловое хранилище 360
 - создание 361
 - Файловый
 - менеджер 329
 - слушатель 332
 - сервер 25, 267
 - сессия 316
 - Файлы проекта 41
 - Файл
 - appinfo.rh 222
 - BDF 428
 - bld.inf 41, 80, 212
 - DEF 73
 - DLL 236
 - DSO 240
 - errrd 135
 - LIB 234
 - MAKE 70
 - MIF 226
 - MMP 70
 - PKG 81
 - RSS 52

- SIS 80
 SVG-T 226
 атрибуты 323
 открытие 338
 переименование и
 удаление 322
 полное имя 313
 режим доступа 340
 ресурсов 50
 чтение и запись
 данных 338, 342
 Формат
 MIF 226
 SVG 226
 SVG-T 226
 Форум Discussion Boards 16
- Х**
 Хендл 152
 открытие 264
- Хранилище данных 359
 открытие 364
 сжатие 365
- Ц**
 Целевая платформа 42
 Центральный репозиторий 26
 Центр сертификации 442
 Цифровой сертификат 38, 91
 пустышка 39
- Ш**
 Шрифт 428
 размер 429
 установка 429
- Э**
 Экзамен Accredited Symbian
 Developer 15
 Экранирование данных 36
- Экранируемые каталоги 36
 Экран
 доступ 425
 рисование 421
 Эмулятор 73, 77
 запуск приложения 111
 меню 75
 настройка параметров 76
 отладка приложения 112
 отличия от устройства 79
- Я**
 Ядро ЕКА2 24
 Язык
 C++ 14
 SQL 367, 451
 запросы 381
 SymbianC++ 124

ЯЗЫК ПРОГРАММИРОВАНИЯ C# 2008 И ПЛАТФОРМА .NET 3.5

4-е издание

Эндрю Троелсен



www.williamspublishing.com

Версия .NET 3.5 привнесла с собой как десятки новых языковых средств C#, так и множество новых API-интерфейсов .NET. В этой книге вы найдете полное описание всех нововведений в характерной для автора дружественной к читателю манере. Помимо прочего, подробно рассматривается язык LINQ, изменения, появившиеся в новой версии языка C# 2008 (автоматические свойства, методы расширений, анонимные типы и т.д.), а также множество функциональных средств среды Visual Studio 2008. Книга рассчитана на программистов разной квалификации, а также будет полезна для студентов и преподавателей дисциплин, связанных с программированием и разработкой для .NET.

ISBN 978-5-8459-1589-4 **в продаже**