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 **WINSCW**, **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 –WINSCW
```

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 MMP-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 MK-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.

MMP-файлы

MMP-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 MMP-файле декларируется использование тех или иных защищенных возможностей. Проект может содержать несколько MMP-файлов. Во время сборки на основе MMP-файлов и выбранной целевой платформы генерируются скрипты для компиляции компонентов.

MMP-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 MMP-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 MMP-файла.

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

USERINCLUDE    ..\inc
SYSTEMINCLUDE   \epoc32\include

SOURCEPATH     ..\src
SOURCE          helloworld.cpp

LIBRARY         euser.lib
```

Формат MMP-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 MMP-файлов. Тем не менее именно в терминах ключевых выражений обсуждаются вопросы, связанные с MMP-файлами, как в документации, так и на многочисленных форумах. В табл. 2.1 приводится описание основных ключевых выражений, используемых в MMP-файлах.

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

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

Продолжение табл. 2.1

Ключевое выражение	Описание
	возможностей с символом "–" в качестве префикса для исключения их из списка. Пример декларации всех защищенных возможностей, кроме группы возможностей производителя. `CAPABILITY ALL -TCB –AllFiles –DRM` В случае, если ключевое выражение `CAPABILITY` в MMP-файле не встречается, при компиляции используется значение `CAPABILITY NONE`
`LANG` *`language-list`*	Позволяет задать список поддерживаемых проектом локализаций. Язык локализации задается двузначным числом. Пример: `LANG 01 16` 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"
`LIBRARY` *`filename-list`* `STATICLIBRARY` *`filename-list`* `DEBUGLIBRARY` *`filename-list`*	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 *`filename-list`* может содержать как одну, так и несколько разделенных пробелами имен файлов библиотек (порядок не важен). Пример: `LIBRARY euser.lib efsrv.lib` Все три ключевых выражения могут использоваться многократно и в любом порядке
`SYSTEMINCLUDE` *`directory-list`* `USERINCLUDE` *`directory-list`*	Эти ключевые выражения задают каталоги, в которых выполняется поиск подключенных к исходному коду заголовочных файлов. 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.

Продолжение табл. 2.1

Ключевое выражение	Описание

	`USERINCLUDE ..\inc\`
	Примечательно, что пользовательские заголовочные файлы ищутся сначала в том же каталоге, в котором находится исходный код, и лишь затем в каталогах, заданных ключевым выражением `USERINCLUDE`. Если и там нужных файлов не оказалось, то поиск проводится в каталогах, указанных в ключевом выражении `SYSTEMINCLUDE`.
	Подключение заголовочных файлов в коде осуществляется с помощью директивы `#include`. При этом имя системных файлов заключается в угловые скобки, а пользовательских — в кавычки. Пример.
	Системный файл:
	`#include <e32def.h>`
	Пользовательский файл:
	`#include "myclass.h"`

`SOURCEPATH` *`directory`*	Задает 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 `SOURCEPATH`. Единственный параметр *`directory`* может быть как относительным путем (определяется от каталога, в котором находится MMP-файл), так и полным (определяется от каталога, заданного в переменной окружения `EPOCROOT`)
`SOURCE` *`source-file-list`*	Задает файлы с исходным кодом, использующиеся данным компонентом проекта. Список *`source-file-list`* представляет собой перечисление имен файлов с расширениями "`.cpp`", разделенных пробелами. Файлы ищутся в каталоге, заданном предшествующим ключевым выражением `SOURCEPATH`. 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

`START RESOURCE` *`source-file`*
`[target` *`target-file-name`*`]`
`[targetpath` *`targetpath`*`]`
`[header]`
`[lang` *`languages`*`]`
`[uid` *`uid-value-1`* `[`*`uid-value-2`*`] ]`
`END`
	Структура, описывающая правила компиляции файла ресурса (RSS). Параметр *`source-file`* задает имя файла и может включать относительный путь к нему. Файл ищется в каталоге, указанном предшествующим ключевым выражением `SOURCEPATH`.
	Значение параметра *`target-file-name`* задает имя файла после компиляции, не содержит расширение (оно добавляется автоматически) и по умолчанию совпадает с именем, задаваемым параметром *`source-file`*.

Продолжение табл. 2.1

Ключевое выражение	Описание
	Директива `targetpath` задает каталог, в который помещается скомпилированный ресурс, причем путь указывается относительно диска `z` эмулятора SDK (каталог `\epoc32\data\z\`). В случае если он опущен, используется значение ключевого выражения `TARGETPATH` для всего MMP-файла. Присутствие директивы `header` сигнализирует о необходимости создания так называемого "заголовочного файла ресурса" (resource header file). Это файл с расширением "`.rsg`", содержащий объявление идентификаторов-индексов, для находящихся в файле ресурса структур. Директива `UID` позволяет задать значения UID2 и UID3 скомпилированного ресурса. По умолчанию UID2 нулевой, а значением UID3 является 20-битовый хеш имени файла. Значения UID2 и UID3 могут быть также переопределены в самом файле ресурса. UID1 ресурса всегда равен `0x101F4A6B`. Директива `LANG` позволяет переопределить настройки локализации MMP-файла для данного файла ресурсов. Расширение скомпилированного файла ресурса имеет вид "`.r`*XX*", где *XX* — двузначный код языка локализации либо `sc`. Файл ресурсов компилируется столько раз, сколько локализаций он имеет

`START BITMAP` *target-file*
`[targetpath` *targetpath*`]`
`[header]`
`[sourcepath` *sourcepath*`]`
`source` *color-depth*
source-bitmap-list
`END`
	Структура `BITMAP` 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 "`.mbm`". Компилируемые в него изображения задаются с помощью директив `sourcepath` и `source`, подобно файлам исходного кода. При этом значение директивы `sourcepath` имеет влияние лишь в пределах этой структуры. Список *bitmap-list* 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 *color-depth* в формате `[c]digit`, `digit`. Присутствие префикса `c` 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 `1` (занимает меньше памяти). Вот небольшой пример объявления

Окончание табл. 2.1

Ключевое выражение	Описание
	MBM-файла, содержащего три изображения, одно из которых является маской первого: START BITMAP icons.mbm HEADER TARGETPATH \resource\apps SOURCEPATH ..\gfx SOURCE c12,1 icon.bmp icon_mask.bmp SOURCE c12 just_image.bmp END
EPOCHEAPSIZE *minimum maximum* EPOCSTACKSIZE *stacksize*	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: EPOCSTACKSIZE 0x5000 Оба ключевых выражения игнорируются при сборке для целевых платформ WINS/WINSCW
EPOCPROCESSPRIORITY *priority*	Приоритет процесса приложения. Может принимать значения low, background, foreground, high, windowserver, fileserver, realtimeserver или supervisor. По умолчанию — foreground. Используется только в исполняемых файлах. Игнорируется при сборке для целевых платформ WINS/WINSCW
COMPRESSTARGET INFLATECOMPRESSTARGET BYTEPAIRCOMPRESSTARGET NOCOMPRESSTARGET	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)

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 местоположения[1].

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>* [= *<initialiser>*]. 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 rcomp 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 ".r*XX*", где *XX* — двузначный код использующегося 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 ".r*XX*"-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.

Заменим все используемые в нашем файле users.rss строки идентификаторами и подключим 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.l`*XX*. 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 "`.lXX`" — 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 "`.lXX`" 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++.