БЛОГ GUNSMOKER-A

...WHEN ALTERING ONE'S MIND BECOMES AS EASY AS PROGRAMMING A COMPUTER, WHAT DOES IT MEAN TO BE HUMAN?..

ОБО МНЕ

АЛЕКСАНДР АЛЕКСЕЕВ
ТВЕРЬ, ТВЕРСКАЯ ОБЛАСТЬ, RUSSIA

Tech geek, have social skills of a thermonuclear device.

ПРОСМОТРЕТЬ ВЕСЬ ПРОФИЛЬ
Написать письмо (e-mail)

СЛУЧАЙНЫЕ ПОСТЫ

Task Dialogs от Vista/Windows 7 в Windows XP: читаем спецификацию с другой стороны
Задачка №17
Создаём систему плагинов, часть 6
Актуальность Delphi

11 ФЕВРАЛЯ 2011 Г.

DLL, DLL Hell, перенаправление DLL, Side-by-Side сборки и манифесты...

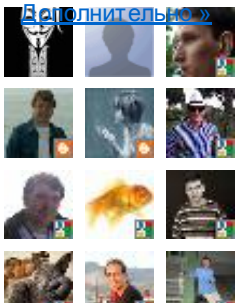
DLL (Dynamic Link Library — библиотека динамической компоновки) — понятие операционных систем Microsoft Windows и IBM OS/2; это библиотека, позволяющая многократное применение несколькими приложениями. К DLL относятся также элементы управления ActiveX и драйверы. В мире UNIX аналогичные функции выполняют т.н. shared objects («разделяемые объекты»). Формат файлов DLL придерживается тех же соглашений, что и формат исполняемых файлов, сочетая код и данные.

Наверняка почти все из читателей когда-нибудь создавали в Delphi DLL. И если вы делали это достаточно много и регулярно, то наверняка знакомы с понятием *DLL Hell*. Первоначально предполагалось, что введение DLL позволит эффективно организовать память и дисковое пространство, используя только один экземпляр библиотечного модуля для различных приложений. Это было особенно важно для ранних версий Microsoft Windows с жёсткими ограничениями по памяти. Далее, предполагалось улучшить эффективность разработок и использования системных средств за счёт модульности. Замена DLL-программ с одной версии на другую должна была позволить независимо наращивать систему, не затрагивая приложений. Кроме того, библиотеки DLL могли использоваться разнотипными приложениями — например, Microsoft Office, Microsoft Visual Studio и т.п. В дальнейшем идея модульности выросла в концепцию COM. Фактически, полных преимуществ от внедрения DLL получить не удалось именно по причине DLL Hell.

Содержание:

Введение

DLL Hell (DLL-кошмар, буквально: DLL-ад) — тупиковая ситуация, связанная с управлением динамическими библиотеками DLL в операционной системе Microsoft Windows. Аналогичная проблема в средах UNIX носит название Dependency Hell. Сущность проблемы заключается в конфликте версий DLL, призванных поддерживать определённые функции. По исходному замыслу, DLL должны быть совместимыми от версии к версии и взаимозаменяемыми в обе стороны. На практике однако оказалось, что достичь этого невозможно. Вы устанавливаете одну программу, и неожиданно перестает работать другая программа, казалось бы, совершенно не связанная с первой (это



может быть сообщение вроде "не найдена библиотека", "не найдена процедура в библиотеке" или что-то более тонкое вроде неправильной работы программы, вылета или зависания). Это происходит потому, что, незаметно для вас, две программы объединены использованием одного файла библиотеки DLL. Для работы этих двух программ могут требоваться совершенно разные версии файла `MSVCRT.DLL` в системной папке. Или же одна программа может обновить элемент управления ActiveX, который используется другой программой, при этом вторая программа может быть не полностью совместима с этим обновлением.

Разве это применимо ко мне?

Вы можете подумать: ну, со мной такого никогда не случится, ведь я очень грамотно проектирую свои DLL, так что новая версия библиотеки всегда обратно совместима с предыдущей, и все существующие программы смогут безболезненно использовать новую версию DLL вместо старой. А мой установщик тоже весьма грамотен и никогда не перезапишет более новую версию DLL старым вариантом, так что в системе всегда будет только последний вариант библиотеки. И установщик никогда не удаляет мою DLL - так что я в порядке, да?

Вообще-то нет. Проблема в том, что даже если **вы** *всё сделали правильно*, обязательно найдутся люди, которые используют вас неправильно. Например, пусть ваша DLL экспортирует такую функцию:

```
1 function Funcenstein(Buffer: Pointer; Size: Integer):
```

Соглашение по вызову просто: функция вернула `True` - значит, она завершилась успешно и вы можете использовать данные в `Buffer`. Если же функция вернула `False`, то она завершилась ошибкой, а значение `Buffer` не определено. Казалось бы, простое и логичное правило: что тут может пойти не так?

Дело в том, что проверять результат работы функции - это же куча работы. Люди, как известно, ленивы. Поэтому они могут заметить, что ваша функция не трогает буфер `Buffer`, если она завершается неудачей. И они могут написать код вроде такого:

```
1 Buf := AllocMem(Sz);
2 Funcenstein(Buf, Sz);
3 DoSomething(Buf, Sz);
4 FreeMem(Buf);
```

В этом куске кода подразумевается, что если функция `Funcenstein` завершится неудачно, то в `Buf` будут нули (ведь он не

ТЭГИ

7 (4)

тронут). Нули - это допустимое значение (скажем, нуль-терминированная строка нулевой длины), поэтому мы просто продолжаем выполнение (функция `DoSomething`).

Но нигде в контракте функции `Funcenstein` об этом не сказано, поэтому это - деталь реализации. Конечно же, вы и подумать не можете, что кому-то придёт в голову использовать вашу DLL таким образом. Поэтому в новой версии DLL (и функции) вы используете буфер `Buffer` для промежуточных вычислений. Вам нужен кусок памяти, а тут как раз он без дела болтается - так что вы используете его. *Вы имеете полное право на это*, вы **всё** делаете правильно. Вот только, когда вы установите на машину клиента новую версию своей DLL, как перестанет работать установленная программа другого человека, который использовал вашу DLL. Вина ваша? Нет. Но что скажет пользователь? "Не ставьте новую версию <*ваш продукт*> - из-за него ломается <*программа другого человека*>!", или: "Проклятый DLL Hell, проклятый Microsoft и проклятый Билл Гейтс!", или даже просто: "Компьютеры такие глупые! Их так тяжело использовать!".

Windows 2000: перенаправление DLL

Раньше приходилось принимать решение о том, какая программа «победила», а какая — «проиграла». Начиная с Windows 2000 появились средства, помогающие разрешить такие конфликты. Тем не менее эти средства представляют собой лишь временные решения, позволяя вернуть систему в рабочее состояние, пока вы будете заниматься поиском более постоянного решения конфликта.

В Windows 2000 реализована минимальная версия технологии, поставляемой сейчас под модным названием **Dynamic-Link Library Redirection** (перенаправление библиотек динамической компоновки). Чтобы включить перенаправление библиотеки DLL, создайте файл с тем же именем, что и файл программы, для библиотек DLL которой требуется перенаправление, но добавив `.local` к имени файла. Например, чтобы использовать перенаправление для программы `C:\Program Files\Litware Inc\Invoice.exe` следует создать файл `C:\Program Files\Litware Inc\Invoice.exe.local`. Содержимое файла не имеет значения; важен сам факт существования этого файла.

После включения перенаправления для программы все попытки этой программы загрузить библиотеку DLL сначала приведут к

поиску в той папке, где расположен файл программы, и лишь затем — в обычных местах. Поэтому в случае конфликта между различными файлами `MSVCRT.DLL` можно включить перенаправление для каждой из программ и поместить копии файлов `MSVCRT.DLL` для частного использования в папку установки каждой программы. После этого каждая программа будет использовать свою собственную версию библиотеки `MSVCRT.DLL`, а именно ту версию, с которой выполнялось тестирование программы.

Волшебство этого метода заключается ещё и в том, что он работает даже тогда, когда программа использует полный путь для загрузки библиотеки DLL. Например, представим себе, что программа пытается загрузить библиотеку `C:\Program Files\Common Files\Proseware Inc\taxplugin.dll`. После обновления до версии Proseware 2.0 вы обнаруживаете, что подключаемые модули расчета налогов несовместимы с программой подготовки счетов. Вы можете скопировать старую версию подключаемого модуля расчета налогов в `C:\Program Files\Litware Inc\taxplugin.dll`. Даже несмотря на то, что при загрузке подключаемого модуля расчета налогов используется полный путь, перенаправление библиотек DLL сначала выполнит поиск в текущей папке и будет использовать локальную копию вместо копии в папке «Proseware Inc».

В операционных системах Windows XP и выше правила перенаправления библиотек DLL немного отличаются, но общий принцип остается тем же. Кроме создания файла с расширением «`local`» можно создать и папку с таким *расширением*. В этом случае поиск будет выполняться в папке с расширением «`local`», а не в папке установки программы. Это позволяет использовать перенаправление для нескольких программ в одной папке, не создавая конфликтов. К примеру, если у программы из примера выше существует папка `C:\Program Files\Litware Inc\Invoice.exe.local`, то будет загружена DLL `C:\Program Files\Litware Inc\Invoice.exe.local\taxplugin.dll`

Осталось только заметить что `.local` файлы и папки игнорируются, если приложение использует манифесты для управления зависимостями (см. ниже).

Перенаправление библиотек DLL не позволит полностью избежать кошмара библиотек DLL, но, по крайней мере, оно предоставляет средства первой помощи для контроля над ситуацией на время решения проблемы. По этой же причине, если раньше для установщиков были рекомендации: пишите свои DLL в системные папки, ведь тогда их смогут использовать и

другие программы. И вы сэкономите на ресурсах! То начиная с Windows 2000 рекомендации меняются на противоположные: ребята, диск сегодня уже не ограничитель, так что установщикам следует располагать DLL в папке с программой. Да, место не сэкономите, но зато программа будет работать надёжно, ведь её не коснётся DLL Hell.

Windows XP: изолированные приложения и side-by-side сборки

Начиная с Windows XP в вашем распоряжении появляется новый механизм, позволяющий избавиться от проблемы DLL Hell: **Isolated Applications** и **Side-by-side Assemblies**. Это решение Microsoft Windows для уменьшения конфликтов версий в Windows приложениях. С его помощью разработчик может описывать требования своих приложений и DLL, так что они будут защищены от влияния других версий библиотек в системе. Конечные клиенты же получают выгоду от использования этого механизма тем, что приложения теперь будут работать надёжнее и будут устойчивы к обновлениям.

Эта технология поддерживаются начиная с Windows Server 2003 и Windows XP. На Windows 2000 вам нужно использовать Dynamic-Link Library Redirection. Кстати говоря, все из вас наверняка уже использовали технологию Isolated Applications, даже не осознавая этого. Так называемый "XP Manifest" включает в вашу программу специальное описание, что вашей программе требуется *новая версия библиотеки Common Controls*, так что при запуске вашей программы она получает особую версию `comctl32.dll` из папки `WinSxS`, а не обычную `comctl32.dll` из `System32`. Это и есть Isolated Applications в действии. Впрочем, сама Windows XP использует эту технологию далеко не везде. А вот Vista и выше используют её на полную (это можно заметить по размеру папки `WinSxS` на обоих системах).

Side-by-side сборки и изолированные приложения

Side-by-side assemblies - это Win32 *сборки* (DLL или COM-сборки), описываемые *манифестом*, разработанные так, что несколько версий одной и той же DLL могут быть загружены в один процесс и работать в нём одновременно, без конфликтов друг с другом (вот откуда идёт название "side-by-side" - т.е. работающие одновременно, спина-к-спине). Когда программист пишет приложение с использованием side-by-side сборок, он создаёт для приложения манифест, указывающий какая версия сборки нужна для этого приложения, если в системе есть несколько версий одной сборки. Такое приложение с манифестом

называется isolated application - *изолированное приложение*. Название, опять же, очевидно: ведь приложение изолируется от влияние других версий библиотек.

Приложение называется *полностью изолированным*, если все его компоненты являются side-by-side сборками. Если же ваше приложение использует один или более компонент, не являющихся side-by-side сборками (например: приложение использует обычную DLL без манифеста), то приложение называется *частично изолированным*. Заметьте, что частично изолированное приложение всё ещё уязвимо для DLL Hell. Учитывая, что мы уже сказали про "XP Manifest", многие (если не все) из ваших программ уже являются частично изолированными приложениями, хотя и в весьма малой степени.

Разработчикам программ рекомендуется разрабатывать полностью изолированные приложения и обновить свои старые приложения до полностью изолированных по следующим причинам:

- Изолированные приложения более устойчивы и надёжны, потому что на них не влияют установки, обновления или удаления других программ в системе.
- Изолированные приложения могут быть спроектированы так, что они будут запускаться ровно с теми версиями библиотек, с которыми они тестировались на машине разработчика.
- Изолированные приложения могут использовать функциональность side-by-side сборок Microsoft.
- Изолированные приложения не зависят от сроков поставки их side-by-side сборок, потому что приложения и администраторы могут обновлять конфигурацию после развёртывания приложения, не требуя переустановки приложения. Понятно, что это не применимо в случае, если у вас есть только одна версия сборки.
- Полностью изолированное приложение может быть установлено с использованием команды `xcopy`. Windows Installer также может быть настроен для установки полностью изолированного приложения без влияния на реестр. (примечание: в этом пункте в основном имеется ввиду т.н. "registration free COM" и аналогичные вещи)

В некоторых случаях вы можете обновить существующее приложение до частично или полностью изолированного без изменения исходного кода и иногда даже без перекомпиляции. Вы можете создать для приложения манифест, который описывает зависимости приложения от side-by-side сборок. Если приложение использует компоненты, которые не являются side-by-side сборками, то их можно распространять как частные

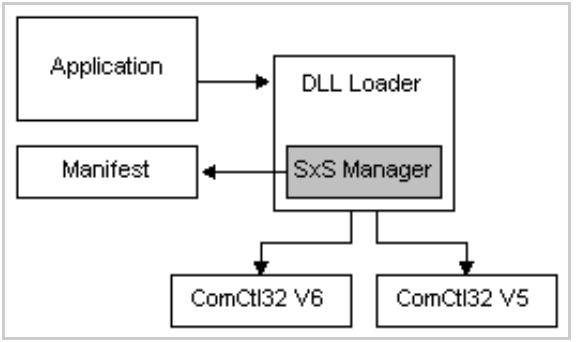
сборки (см. ниже).

Переделать приложение в полностью изолированное не всегда возможно. К примеру, некоторые компоненты, защищаемые Windows File Protection (WFP), не доступны в виде side-by-side сборок и не могут быть установлены вместе с приложением как частные сборки.

Управление сборками с помощью манифестов

Как уже было сказано, side-by-side сборки Windows описывается манифестами. Манифесты содержат мета-данные, которые описывают side-by-side сборки и зависимости этих сборок. Сборка может содержать набор DLL, классов Windows, COM серверов, библиотек типов или интерфейсов, которые представляются приложению как единое целое. Все они описываются в манифесте сборки. Обычно же, типичная side-by-side сборка - это одна DLL с одним манифестом в ней. К примеру, сборка `COMCTL32` от Microsoft является DLL библиотекой `comctl32.dll` с одним манифестом. А вот сборка run-time от Microsoft Visual C++ содержит несколько файлов. В некоторых случаях версии сборки, указанные в манифесте, могут быть изменены глобально или для одного приложения - авторами сборок, разработчиками приложений или системными администраторами. Разработчики также могут использовать side-by-side сборки, разработанные Microsoft, или side-by-side сборки других разработчиков. Мы уже не раз упоминали пример с подключением новой версии (6.0) библиотеки Common Controls с поддержкой тем в ваших приложениях.

Процесс приложения может использовать более одной версии side-by-side сборки. К примеру, приложение загружает две библиотеки, одна из которых использует версию 1 side-by-side сборки, а вторая загружаемая библиотека требует версии 2 той же сборки. Для различения сборок друг от друга используются манифесты и номера версий сборок. Используя эту информацию, загрузчик DLL может определить правильную версию DLL для загрузки. Это показано на следующем рисунке:



Представление типичной side-by-side сборки

В этом примере, обе версии `comctl32.dll` (6.0 и 5.0) находятся в кэше side-by-side сборок и доступны для приложений. Когда приложение запускается и инициирует загрузку DLL, то менеджер side-by-side сборок определяет, какая версия библиотеки нужна для приложения (что описано в манифесте этого приложения). Если такового требования нет, то менеджер загрузит вариант по-умолчанию. Для Windows XP это будет версия 5.0 `comctl32.dll`. Если менеджер находит информацию о зависимости от версии 6.0 (указано в манифесте), то именно эта версия и будет загружена для приложения.

Манифесты

Итак, с понятиями изолированного приложения и side-by-side сборок мы разобрались. А что такое манифест, о котором мы постоянно говорим? Манифест представляет собой XML файл, который привязывается к side-by-side сборке или изолированному приложению. Он может внедряться в исполняемый файл или быть отдельным файлом. Манифесты и файлы конфигураций не локализуются. Манифесты идентифицируют саму сборку (указывая её имя и версию) - через элемент `assemblyIdentity`. Они также могут содержать информацию о привязках и активации - вроде COM-классов, интерфейсов и библиотек типов. Ранее эта информации хранилась в реестре. В манифестах также указываются файлы, входящие в сборку. Side-by-side сборки не регистрируются в системе, но они доступны для приложений (и других сборок), которые указывают свою зависимость от них в своих манифестах.

Манифесты в виде отдельных файлов позволяют администраторам и приложениям управлять версиями side-by-side сборок после развёртывания приложения. Каждая side-by-side сборка должна иметь манифест, ассоциированный с ней. При установке Windows XP устанавливаются side-by-side сборки от Microsoft вместе с их манифестами. Если вы собираетесь разрабатывать ваши собственные side-by-side сборки, то вы также должны устанавливать их манифесты.

Как вы уже догадались, манифесты сборок и манифесты изолированных приложений - это немного разные вещи. Существуют такие типы манифестов:

- *Манифесты сборок* (assembly manifests) описывают side-by-side сборки. Они используются для управления именами, версиями, ресурсами и зависимостями side-by-side сборок. Манифесты общих сборок хранятся в папке `WinSxS` (из `System32`). Манифесты частных сборок хранятся либо в

ресурсах DLL, либо в папке приложения.

- *Манифесты приложений* (application manifests) описывают изолированные приложения. Они используются для управления именами и версиями сборок, с которыми должно связываться приложение при выполнении. Манифесты приложений хранятся в папке с исполняемым файлом приложения, либо же внедряются в ресурсы приложения.
- *Файлы конфигурации приложений* (Application Configuration Files) - это манифесты, используемые для перенаправления версий зависимых сборок.
- *Файлы конфигурации поставщика* (Publisher Configuration Files) - это манифесты, используемые для перенаправляения side-by-side сборки на совместимую с ней версию.

В этой статье не рассматриваются вопросы перенаправления сборок.

Чтобы создать манифест для приложения или сборки, вы создаёте пустой текстовый файл с расширением .manifest. К примеру, манифест приложения или сборки, который ссылается на приложение или сборку `myassembly` должен иметь следующий формат:

```
myassembly[.resource-ID].manifest
```

Вы можете пропустить часть `[.resource-ID]`, если `resource-ID` (идентификатор ресурса - см. чуть ниже) равен 1. Например, `Project1.exe.manifest`, эквивалентный ему `Project1.exe.1.manifest` или `Project1.exe.2.manifest`.

Внимание: если вы применяете манифест в виде отдельного файла и меняете сам манифест без изменения двоичного модуля, то вы должны **обновить дату изменения исполняемого модуля**, чтобы манифест вступил в действие. Происходит это по той простой причине, что Windows кэширует данные и не проверяет наличие/отсутствие/изменение манифеста, если файл программы не поменялся. Это типичная оптимизация частого случая (файл не меняется).

Если вам не нравится манифест в виде отдельного файла, а также для случаев, когда это неприменимо - вы можете внедрить манифест в исполняемый файл в виде ресурса. Для этого используются такие константы:

```
1  const
2    RT_MANIFEST                                          =
3    CREATEPROCESS_MANIFEST_RESOURCE_ID                   =
4    ISOLATIONAWARE_MANIFEST_RESOURCE_ID                  =
5    ISOLATIONAWARE_NOSTATICIMPORT_MANIFEST_RESOURCE_ID =
```

Значение идентификатора ресурса определяет, как зависимости сборок, описанные в манифесте, будут использоваться загрузчиком ОС.

К примеру, если вы установите идентификатор ресурса в 1, то загрузчик ОС будет использовать зависимости сборок, указанные в манифесте, как умалчиваемые для всего процесса. Все загружаемые плагины также будут использовать эти же зависимости.

Таблица ниже показывает, как загрузчик ОС использует манифест с различными значениями идентификатора ресурса. Заметьте, что разработчик может управлять этим процессом вручную, используя функции активации контекста (Activation Context API), которые не рассматриваются в этой статье.

Идентификатор	По умолчанию для процесса?	Используется при статическом импорте?	Используется для EXE?
1	Да	Да	Да
2	Нет	Да	Да
3	Нет	Нет	Да

`CREATEPROCESS_MANIFEST_RESOURCE_ID` должен использоваться приложениями, которые не работают с плагинами.

`ISOLATIONAWARE_MANIFEST_RESOURCE_ID` используется приложениями, которые загружают сторонние библиотеки (плагины).

Подробнее об этом процессе - см. ниже в разделе практики.

Частные и разделяемые сборки

Разделяемая (shared) сборка - это сборка, доступная для использования несколькими приложениями на машине. В Windows 7, Windows Vista и Windows XP side-by-side сборки могут быть установлены как разделяемые. Разделяемые side-by-side сборки не регистрируются глобально в системе, вместо этого они доступны приложениям, которые указывают зависимость от этой сборки в своих манифестах. Несколько разных версий side-by-side сборок могут разделяться одновременно несколькими приложениями, работающими одновременно.

Разделяемые side-by-side сборки устанавливаются в папку WinSxS. Разделяемые side-by-side сборки могут быть установлены при установке обновления операционной системы

или пакетом Windows Installer, который устанавливает или обновляет ваше приложение.

До Windows XP общие сборки (называемые просто DLL) регистрировались глобально и устанавливались в папку System. В этом случае приложению была доступна только последняя установленная версия сборки. Side-by-side сборка могла быть установлена как частная сборка для эксклюзивного использования приложением.

Как видите, здесь обычные и привычные вещи называются новыми именами.

Частная (private) сборка - это сборка, которая распространяется с конкретным приложением и используется только этим приложением. Иными словами, другие приложения не используют эту частную сборку (сборка не является разделяемой). Частные сборки являются одним из способом создавать изолированные приложения. Частные сборки должны проектироваться для работы side-by-side с другими версиями этой же сборки в системе.

Частные сборки должны иметь манифест сборки. Заметьте, что при распространении DLL как сборки к ней применяются ограничения на имя, чтобы соответствовать способу поиска частных сборок в Windows. При этом рекомендуется включать манифест сборки в саму DLL. В этом случае идентификатор ресурса должен быть равен 1 и имя частной сборки должно быть таким же, как и имя DLL. Например, если имя DLL - `MICROSOFT.WINDOWS.MYSAMPLE.DLL`, то значение атрибута `name`, используемого в элементе `assemblyIdentity` манифеста, также должно быть `Microsoft.Windows.mysample`. Альтернативный метод поиска частных сборок - предоставить манифест сборки в отдельном файле. В этом случае имя сборки и имя манифеста должны быть *отличны* от имени DLL. К примеру, `Microsoft.Windows.mysampleAsm`, `Microsoft.Windows.mysampleAsm.manifest` или `Microsoft.Windows.mysample.dll`. Как правило, это применяется для сборок, состоящих более чем из одной DLL.

Частные сборки устанавливаются в папку приложения. Обычно это сама папка приложения с исполняемым файлом. Частные сборки также могут располагаться в подпапке с именем сборки, или в подпапке с именем языка (тема локализации сборок не рассматривается в этой статье). Например, вы можете использовать следующую структуру папок для размещения частной сборки `Microsoft.tools.pop` без указания языка:

Структура	Описа
`AppDir\MICROSOFT.TOOLS.POP.DLL`	Маниф внедрё ресурс саму D
`AppDir\Microsoft.Tools.Pop.MANIFEST`	Маниф сделан отдель файле.
`AppDir\MICROSOFT.TOOLS.POP\MICROSOFT.TOOLS.POP.DLL`	Маниф внедрё ресурс котора распол в подпа именем сборки
`AppDir\Microsoft.Tools.Pop\Microsoft.Tools.Pop.MANIFEST`	Маниф распол в отдел файла подпап именем сборки

Частные сборки могут быть установлены любым способом установки, который может копировать файлы сборки в папку - к примеру, простая команда `xcopy`.

Частные сборки могут быть также установлены на операционную систему ниже Windows XP. В этом случае сборка должна быть зарегистрирована в системе обычным способом, а манифест сборки использоваться не будет. Копия частной сборки может быть скопирована в папку приложения для эксклюзивного использования этим приложением. Другая версия сборки может быть глобально зарегистрирована в системе и быть доступной любому приложению. Глобальная версия сборки может быть той версией, что установило ваше приложение, или любой другой - как старше, так и младше. Мы обсуждали эту тему в разделе выше, где мы говорили про перенаправление DLL в Windows 2000.

Заметьте, что шаги для создания частной сборки идентичны созданию разделяемой сборки (рассмотрено в разделе практики ниже), за исключением следующих:

- Частную сборку можно не подписывать, а элемент `publickeyToken` является необязательной частью блока

`assemblyIdentity` манифеста сборки.

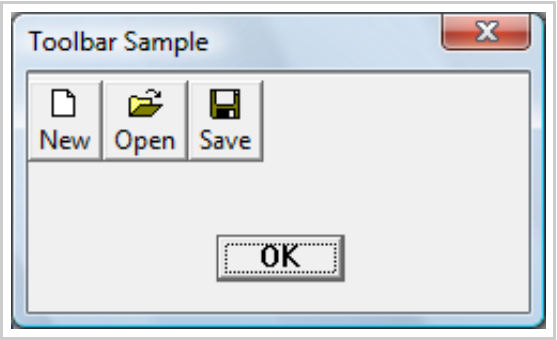
- Частные сборки могут быть установлены с использованием любой технологии установки - простым копированием. Частные сборки не обязаны устанавливаться с помощью Windows Installer.

Практический пример: включение визуальных стилей

Общие элементы управления (common controls), которые включены в состав Windows XP, Windows Vista и Windows 7, имеют новую функциональность, которая называется "визуальными стилями" (visual styles). Внешний вид элементов управления может быть изменён, основываясь на настройках внешнего вида системы, изменяемых пользователем.

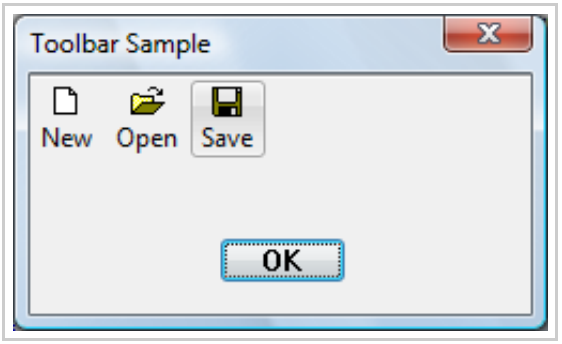
Примечание: визуальные стили не работают в режиме 256-цветов.

На рисунке ниже показано окно с панелью задач на Windows Vista. Приложение было скомпилировано со стандартной библиотекой общих элементов управления.



Снимок окна с кнопками без использования прозрачности

Для сравнения - на рисунке ниже показано то же окно, но в этот раз приложение было привязано к новой версии общих элементов управления и, поэтому, использует визуальные стили. Обратите внимание на разницу в отображении кнопок в клиентской области.



Снимок окна с кнопками с использованием прозрачности

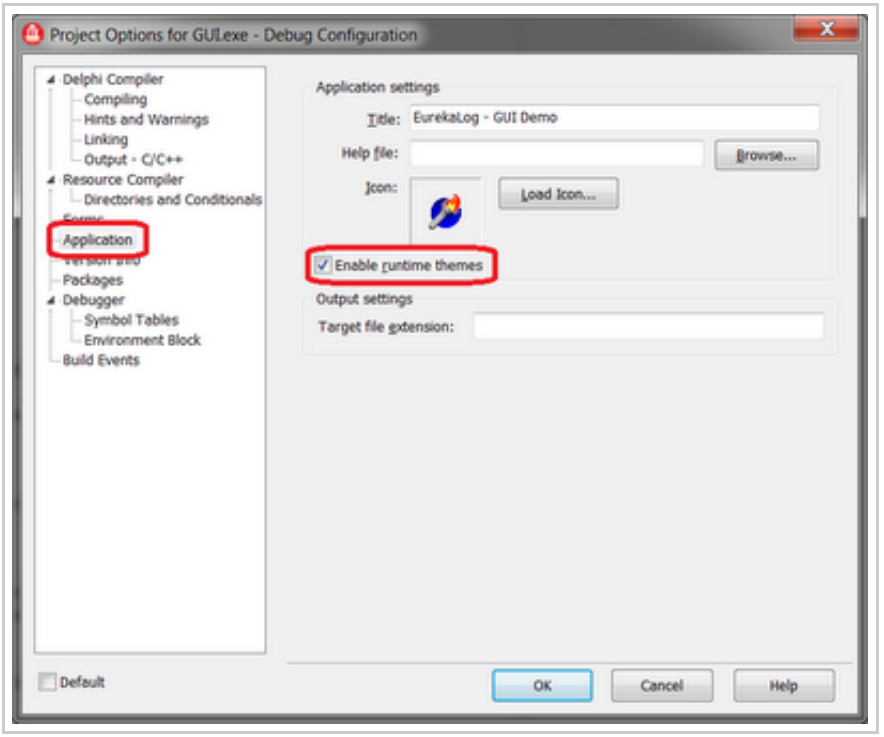
Новые библиотеки (ComCtl32.dll версии 6 и UxTheme.dll) доступны в по умолчанию в Windows XP и выше. Они позволяют задействовать в вашем приложении визуальные стили. UxTheme.dll используется ComCtl32.dll для реализации визуальных стилей. ComCtl32.dll запрашивает у UxTheme.dll размеры и другую информацию об элементах управления и вызывает UxTheme.dll для прорисовки различных частей элементов управления.

Чтобы использовать визуальные стили, приложение должно быть запущено на системе с наличием ComCtl32.dll версии 6. Windows XP и выше содержит ComCtl32.dll версии 5 и версии 6. Предыдущие версии операционной системы Windows содержат только ComCtl32.dll версии 5. По умолчанию, приложения везде используют ComCtl32.dll версии 5 по соображениям обратной совместимости. Если вы хотите, чтобы ваше приложение использовало ComCtl32.dll версии 6, вам нужно добавить манифест к своей программе. Тогда ваш код будет использовать ComCtl32.dll версии 5 на старых системах, а где возможно - ComCtl32.dll версии 6. Версия 6 также включает в себя несколько новых элементов управления и некоторые новые возможности старых элементов управления, но самое большое изменение - это, конечно же, поддержка визуальных стилей.

Если вы используете только стандартные элементы управления, то вам не нужно делать никаких других действий для включения визуальных стилей. Если же вы используете элементы управления с custom-прорисовкой, то вам нужно использовать предоставляемое API для получения информации о текущем визуальном стиле и рисовать свои элементы управления в этом стиле. См. Using Visual Styles with Owner-Drawn Controls.

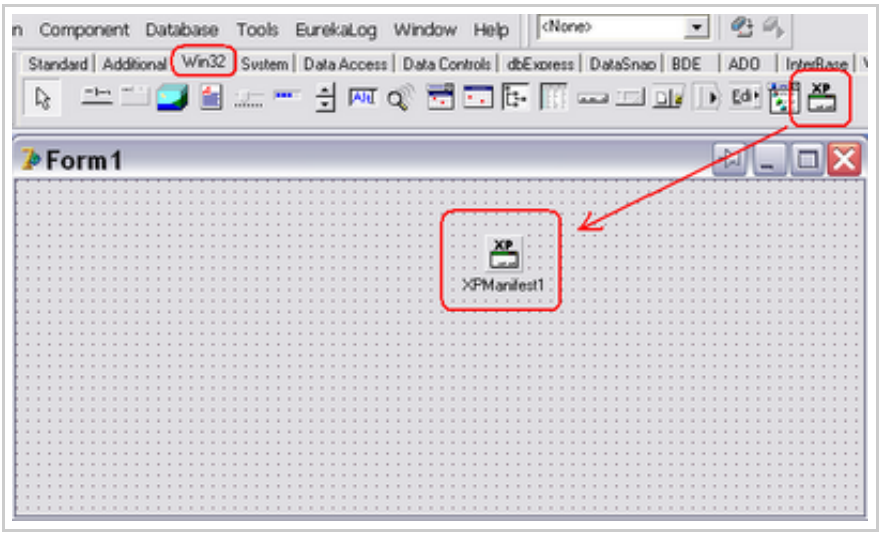
Автоматическое использование для чайников

В Delphi есть два способа для подключения манифеста - автоматический и вручную. Для автоматического способа вам нужно зайти в настройки проекта (`Project` / `Options`), перейти на вкладку `Application` и установить галочку `Enable runtime themes`:



Обратите внимание, что эта опция называется неверно: *темы* (themes) были ещё в Windows 95, а *визуальные стили* (visual styles) появились только в Windows XP. Поэтому эта опция должна называться как минимум `Enable visual styles`. Кроме того, эта же опция подключает в программу и "манифест Vista" - указание требуемого уровня привилегий для работы программы. О чём, вообще говоря, в названии опции нет даже и намёка. Поэтому, правильное название опции должно было бы быть `Add default manifests (enable visual styles and set execution level)`.

В старых версиях Delphi такой опции нет, а вместо неё используется компонент `TXPManifest`. Вам достаточно бросить его на форму (одну, любую) в вашем приложении:



Для отключения визуальных стилей вы снимаете галочку `Enable`

`themes` (а в случае использования старых версий Delphi - удаляете с формы компонент `TXPManifest`, **а также удаляете модуль `XPMan` из списка `uses`**).

Автоматический способ прост, но не лишён недостатков. Во-первых, при его использовании в программу включается манифест по умолчанию. Если вас не устраивают эти настройки, то вы не можете их изменить. Во-вторых, он работает не для всех сценариев использования. В-третьих, использование этого способа означает подключение в программу манифеста. Если вам нужно использовать манифест для указания дополнительной информации (см. примеры ниже), то вы не можете это сделать - ведь манифест может быть только один. В этом случае вам нужно отказаться от манифеста по умолчанию и добавить в программу свой манифест, включив в него как вашу дополнительную информацию, так и информацию, добавляемую автоматическим способом.

Поэтому, когда вы не можете использовать автоматический способ, то можно сделать всё руками.

Использование манифестов для включения визуальных стилей в приложениях

Манифест приложения позволяет последнему указать, какую версию библиотеки ComCtl32.dll оно требует. Манифесты представляют собой XML файлы. Имя файла манифеста приложения представляет собой имя файла приложения с добавленным к нему ".manifest". Например, `Project1.exe.manifest`. Следующий пример манифеста показывает, что первая секция манифеста относится к нему самому. В таблице ниже показаны атрибуты, которые необходимо установить для элемента `assemblyIdentity`:

Атрибут	Описание
version	Версия манифеста. Версия представляется в форме major.minor.revision.build (т.е., n.n.n.n, где n <=65535).
processorArchitecture	Процессор, для которого предназначено ваше приложение.
name	Включает в себя имя компании, имя продукта и имя приложения.
type	Тип вашего приложения, например: Win32.

Манифест-пример ниже также предоставляет описание вашего приложения и указывает его зависимости. Следующая таблица показывает атрибуты, устанавливаемые для элемента

`assemblyIdentity` в блоке зависимостей:

Атрибут	Описание
type	Тип компонента-зависимости. Например: Win32.
name	Имя компонента.
version	Версия компонента.
processorArchitecture	Процессор, для которого предназначен компонент.
publicKeyToken	Токен публичного ключа компонента.
language	Язык компонента.

Ну а теперь - и сам пример манифеста.

Важно: если вы пишете приложение для платформы Windows x86-64, то вы должны изменить запись `processorArchitecture` на `processorArchitecture="amd64"`. Вы также можете указать "*"для указания любой платформы.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" man
<assemblyIdentity
    version="1.0.0.0"
    processorArchitecture="X86"
    name="ИмяКомпании.ИмяПродукта.ИмяПриложения"
    type="win32"
/>
<description>Здесь - описание вашего приложения.</desc
<dependency>
    <dependentAssembly>
        <assemblyIdentity
            type="win32"
            name="Microsoft.Windows.Common-Controls"
            version="6.0.0.0"
            processorArchitecture="X86"
            publicKeyToken="6595b64144ccf1df"
            language="*"
        />
    </dependentAssembly>
</dependency>
</assembly>
```

Части до блока `dependency` (т.е. первый блок `assemblyIdentity` и тэг `description`) должны быть модифицированы для вашего приложения вписыванием в него ваших значений. Обратите внимание, как атрибут `name` в первом блоке `assemblyIdentity` похож на атрибут `name` во втором блоке `assemblyIdentity`.

Если вам не нравится манифест в виде отдельного файла, то вы можете внедрить манифест в исполняемый файл в виде ресурса. Для этого используются такие константы:

```
const
  RT_MANIFEST                                          =
```

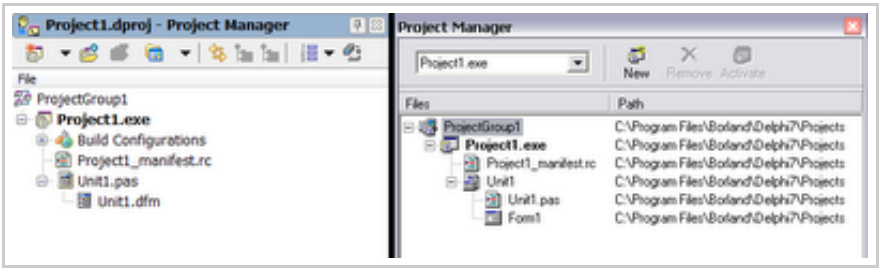
```
3   CREATEPROCESS_MANIFEST_RESOURCE_ID                =
4   ISOLATIONAWARE_MANIFEST_RESOURCE_ID               =
5   ISOLATIONAWARE_NOSTATICIMPORT_MANIFEST_RESOURCE_ID =
```

Вот как это делается. Вы создаёте пустой текстовый файл с расширением `.rc`, например `Project1_manifest.rc`. Затем вы добавляете в него строчку:

```
1 24 "Project1.exe.manifest"
```

Где `1` и `24` - константы выше, а `Project1.exe.manifest` - уже созданный и существующий файл, содержащий манифест (XML файл с текстом из примера выше).

После чего вы добавляете .rc файл в проект, используя команду `Project / Add to project`. Добавленный файл будет виден в менеджере проектов (слева - Delphi XE, справа - Delphi 7):



Вот и всё. Не забудьте только отключить опцию `Enable runtime themes` или удалить `TXPManifest` и `XPMan`.

> **Примечание 1:** в Delphi XE и выше намного проще воспользоваться менеджером ресурсов, вызываемым из меню `Project / Resources`. Вы просто щёлкаете по кнопке "Add", выбираете свой .manifest файл и указываете его имя (1) и тип ресурса (24). Всё, не нужен никакой .rc файл.
>
> **Примечание 2:** возможно, что в некоторых случаях вам также может потребоваться вручную добавить эту строчку в файл проекта (например, Project1.dpr):
>
> ```
> {$R 'Project1_manifest.res' 'Project1_manifest.rc'}
> ```
>
> Сделать это можно в любое место - к примеру, сразу после строки с `program` и до `uses`:
>
> ```
> 1  program Project1;
> 2
> 3  {$R 'Project1_manifest.res' 'Project1_manifest.r
> 4
> 5  uses
> 6    Forms,
> 7    Unit1 in 'Unit1.pas' {Form1};
> 8
> 9  {$R *.res}
> 10
> 11 begin
> 12   Application.Initialize;
> 13   Application.CreateForm(TForm1, Form1);
> 14   Application.Run;
> 15 end.
> ```

Каждый раз, когда вы будете делать компиляцию проекта, файл `Project1_manifest.rc` будет скомпилирован в `Project1_manifest.res` (вот почему мы назвали его `Project1_manifest.rc`, а не `Project1.rc` - чтобы не перезаписать стандартный `Project1.res`), а файл `Project1_manifest.res` будет подключен в готовый .exe файл.

Если по какой-то причине вам не удаётся скомпилировать .rc файл таким образом, то вам нужно будет сделать это вручную вызвав компилятор ресурсов `brcc32` из папки `Bin` Delphi, например (запускать надо из папки с .rc файлом):

```
"C:\Program Files\Embarcadero\RAD Studio\8.0\bin\brcc32.
exe" Project1_manifest.rc
```

Эта команда создаст файл `Project1_manifest.res` по `Project1_manifest.rc`, после чего файлы .rc и .manifest можно удалить, а в .dpr файл вставить:

```
1 {$R Project1_manifest.res}
```

Файл `Project1_manifest.res`, разумеется, удалять не нужно.

В следующих пунктах описаны шаги для применения визуальных стилей в приложениях разных типов. Обратите внимание на отличия манифестов в каждом случае.

ИСПОЛЬЗОВАНИЕ COMCTL32.DLL ВЕРСИИ 6 В ПРИЛОЖЕНИЯХ, КОТОРЫЕ ИСПОЛЬЗУЮТ ТОЛЬКО СТАНДАРТНЫЕ РАСШИРЕНИЯ

Примеры приложений, которые не используют расширения сторонних производителей:

- Калькулятор
- Косынка
- Сапёр
- Блокнот
- Солитер

Для приложений этого типа используется такой манифест:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?
2  <assembly xmlns="urn:schemas-microsoft-com:asm.v1" man
3  <assemblyIdentity
4      version="1.0.0.0"
5      processorArchitecture="X86"
6      name="ИмяКомпании.ИмяПродукта.ИмяПриложения"
7      type="win32"
8  />
9  <description>Здесь - описание вашего приложения.</desc
10 <dependency>
11     <dependentAssembly>
12         <assemblyIdentity
13             type="win32"
14             name="Microsoft.Windows.Common-Controls"
15             version="6.0.0.0"
16             processorArchitecture="X86"
```

```
            publicKeyToken="6595b64144ccf1df"
            language="*"
        />
    </dependentAssembly>
</dependency>
</assembly>
```

И в RC-файле подключается он так:

```
1 24 "Project1.exe.manifest"
```

Либо же вы можете просто оставить .manifest файл в папке с программой. ОС сначала загрузит манифест из файла, а лишь затем будет проверять секцию в exe-файле. Версия манифеста в отдельном файле имеет приоритет.

ИСПОЛЬЗОВАНИЕ COMCTL32 ВЕРСИИ 6 В ПРИЛОЖЕНИЯХ, КОТОРЫЕ ИСПОЛЬЗУЮТ РАСШИРЕНИЯ, ПЛАГИНЫ ИЛИ DLL ОТ ДРУГИХ ПРОИЗВОДИТЕЛЕЙ

Вот примеры приложений, использующих расширения от других людей:

- Microsoft Management Console (MMC)
- Оболочка Windows (Windows Shell)
- Delphi
- Total Commander

ComCtl32.dll версии 6 не является обратно совместимой с предыдущими версиями библиотеки. Использование ComCtl32.dll версии 6 требует изменения кода. Поэтому, если ваше приложение использует компоненты, разработанные другими людьми, вы не можете изменить их так, чтобы они работали с ComCtl32.dll версии 6. Иными словами, такое приложение применяет ComCtl32.dll версии 6 только для себя, но не для загружаемых компонентов. А каждый компонент отдельно декларирует, какая версия ComCtl32.dll ему необходима.

Для приложений этого типа используется такой манифест:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" man
<assemblyIdentity
    version="1.0.0.0"
    processorArchitecture="X86"
    name="ИмяКомпании.ИмяПродукта.ИмяПриложения"
    type="win32"
/>
<description>Здесь - описание вашего приложения.</desc
<dependency>
    <dependentAssembly>
        <assemblyIdentity
            type="win32"
            name="Microsoft.Windows.Common-Controls"
            version="6.0.0.0"
            processorArchitecture="X86"
            publicKeyToken="6595b64144ccf1df"
            language="*"
```

```
        />
    </dependentAssembly>
</dependency>
</assembly>
```

И в RC-файле подключается он так:

```
2 24 "Project1.exe.manifest"
```

ИСПОЛЬЗОВАНИЕ COMCTL32 ВЕРСИИ 6 В ПАНЕЛИ УПРАВЛЕНИЯ ИЛИ DLL, КОТОРАЯ ЗАПУСКАЕТСЯ RUNDLL32.EXE

Для приложений этого типа используется такой манифест:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" man
<assemblyIdentity
    version="1.0.0.0"
    processorArchitecture="X86"
    name="ИмяКомпании.ИмяПродукта.ИмяПриложения"
    type="win32"
/>
<description>Здесь - описание вашего приложения.</desc
<dependency>
    <dependentAssembly>
        <assemblyIdentity
            type="win32"
            name="Microsoft.Windows.Common-Controls"
            version="6.0.0.0"
            processorArchitecture="X86"
            publicKeyToken="6595b64144ccf1df"
            language="*"
        />
    </dependentAssembly>
</dependency>
</assembly>
```

И в RC-файле подключается он так:

```
123 24 "Project1.exe.manifest"
```

ИСПОЛЬЗОВАНИЕ COMCTL32 ВЕРСИИ 6 В ОСНАСТКЕ MMC

Поддержка визуальных стилей может быть добавлена в **большее число типов приложений, чем эти три, описанных выше**. К примеру, если вы разрабатываете оснастку MMC, вы можете добавить ей поддержку визуальных стилей использованием такого манифеста:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" man
<assemblyIdentity
    version="1.0.0.0"
    processorArchitecture="X86"
    name="ИмяКомпании.ИмяПродукта.ИмяПриложения"
    type="win32"
/>
<description>Здесь - описание вашего приложения.</desc
<dependency>
    <dependentAssembly>
```

```
        <assemblyIdentity
            type="win32"
            name="Microsoft.Windows.Common-Controls"
            version="6.0.0.0"
            processorArchitecture="X86"
            publicKeyToken="6595b64144ccf1df"
            language="*"
        />
    </dependentAssembly>
</dependency>
</assembly>
```

И в RC-файле подключается он так:

```
2 24 "Project1.exe.manifest"
```

Практический пример: использование частных сборок

Настало время поговорить и о создании своих собственных сборок. Быть может вам кажется, что это что-то страшное - ведь сколько всего было сказано в этой теме. Но на самом деле это очень просто.

Одна DLL - одна сборка

Для начала возьмите или создайте любое .exe приложение, которое загружает или статически ссылается на DLL. Вы можете создать пустое приложение и пустую DLL для этого эксперимента. Запустите приложение. Оно запустилось? Запустилось. И приложение загрузило DLL. Что же нам сделать, чтобы превратить DLL в сборку? А всего ничего: создать манифест сборки и подключить его в DLL.

Вот пример файла манифеста сборки:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
  <assembly xmlns="urn:schemas-microsoft-com:asm.v1" m
    <assemblyIdentity
        version="1.0.0.0"
        processorArchitecture="X86"
        name="Alex.Demo.Assembly"
        type="win32"
        language="*"
    />
    <description>Пример сборки.</description>
  </assembly>
```

Подключается он как:

```
1 24 "Manifest.manifest"
```

Здесь мы говорим, что наша DLL - это частная сборка с именем `Alex.Demo.Assembly` версии 1.0.0.0. Заметьте, что архитектура

процессора здесь указана явно. Кроме того, здесь отсутствуют атрибут `publickeyToken`, необходимый для разделяемых сборок.

Ну... вот, собственно и всё. Компилируйте DLL (разумеется, её имя должно быть `Alex.Demo.Assembly.dll` и приложение должно загружать или импортировать функцию именно из `Alex.Demo.Assembly.dll`) и теперь вы получите уже не просто DLL, а полноценную частную сборку.

Запустите приложение теперь. Ну, вроде бы ничего не изменилось, да? Да. Всё потому, что в самом приложении мы не указали зависимость от этой сборки. Поэтому приложение загружает эту сборку как обычную DLL.

Как у нас указываются зависимости от сборки? Правильно, подключением манифеста приложения. Вот пример такого манифеста:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
  <assembly xmlns="urn:schemas-microsoft-com:asm.v1" m
    <assemblyIdentity
      type="win32"
      name="Alex.Demo.Application"
      version="1.0.0.0"
      processorArchitecture="*"/>

    <dependency>
      <dependentAssembly>
        <assemblyIdentity
          type="win32"
          name="Microsoft.Windows.Common-Controls"
          version="6.0.0.0"
          publicKeyToken="6595b64144ccf1df"
          language="*"
          processorArchitecture="*"/>
      </dependentAssembly>
    </dependency>

    <dependency>
      <dependentAssembly>
        <assemblyIdentity
          type="win32"
          name="Alex.Demo.Assembly"
          version="1.0.0.0"
          language="*"
          processorArchitecture="*"/>
      </dependentAssembly>
    </dependency>

  </assembly>
```

Подключается манифест как обычно:

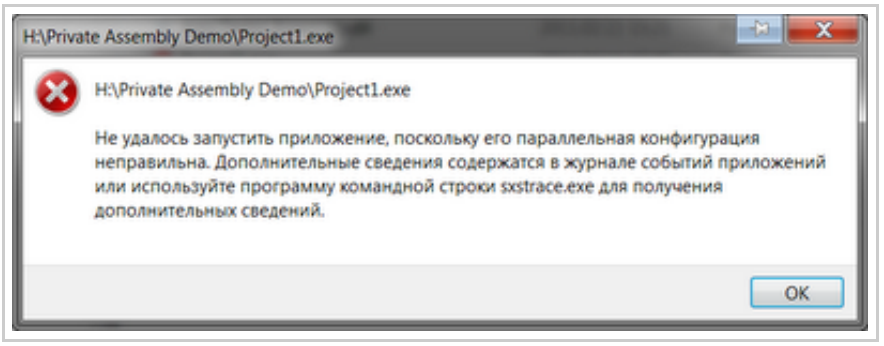
```
1 24 "Manifest.manifest"
```

Заметьте, что, по сути, это "стандартный манифест XP" (если так можно выразиться) с добавленным к нему одним блоком `dependency`, описывающий зависимость приложения от сборки `Alex.Demo.Assembly` версии 1.0.0.0.

Что будет, если запустить приложение теперь? Ну, вроде бы, снова ничего не изменится. Однако, под капотом произошло важное изменение: теперь DLL грузится как сборка. Что означает, что в силу вступают проверки версионной зависимости.

Как это можно увидеть? Попробуйте поменять версию (атрибут `version`) либо в манифесте приложения, либо в манифесте сборки (конечно же, речь идёт об атрибуте `version` для блока `assemblyIdentity` с `name` = `Alex.Demo.Assembly`). Скажем, пусть мы поменяем номер самой сборки с 1.0.0.0 до 2.0.0.0. Пересоберём сборку и запустим приложение.

Опа. Теперь мы получаем сообщение о том, что приложение не может найти нужную DLL:



Как же так, ведь DLL никуда не пропала и вообще мы её не двигали и ничего не меняли? А вот это и есть действие механизма сборок. Ведь приложению нужна сборка `Alex.Demo.Assembly` версии 1.0.0.0, но в папке лежит только версия 2.0.0.0. Это не та, что нам надо. А версии 1.0.0.0 нигде нет - вот поэтому приложение не может её загрузить и отказывается запускаться.

Это справедливо как для статического связывания, так и для загрузки через `LoadLibrary`.

Одна сборка - несколько DLL

Итак, мы реализовали сборку в виде единственной DLL и нам это практически ничего не стоило. Всего делов-то, добавить манифест (кстати, было бы неплохо в опции проекта Delphi добавить галочку "Enable assembly manifest" по аналогии с "Enable visual themes").

Что касается ситуации с несколькими файлами в одной сборке, то я не буду подробно её описывать - вы сможете разобраться с ней по аналогии. Замечу только, что для этого манифест приложения должен ссылаться на сборку, манифест сборки

должен быть отдельным файлом (`имя-сборки.manifest`), компоненты сборки должны называться отлично от имени самой сборки, а приложение при импорте функций из DLL должно ссылаться на компонент (конкретную DLL), а не имя сборки (потому что сборка - это описание зависимостей и файлов, а не библиотека функций). Иными словами, в коде не меняется ничего - меняются только манифесты. И вот простой пример манифеста сборки с двумя файлами:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
  <assembly xmlns="urn:schemas-microsoft-com:asm.v1" m
    <assemblyIdentity
        version="1.0.0.0"
        processorArchitecture="x86"
        name="Alex.Demo.Assembly"
        type="win32"
        language="ru-RU"
    />
    <description>Пример сборки.</description>
    <file name="Alex.Demo.Assembly.Component1.dll" />
    <file name="Alex.Demo.Assembly.Component2.dll" />
  </assembly>
```

Side-by-Side в частных сборках

Хорошо, а как же тогда нам разместить в папке две версии одной частной сборки? Чтобы приложение зависимое от сборки 2.0.0.0 грузило бы её, а плагин приложения, зависимый от 1.0.0.0 - грузил бы именно 1.0.0.0. Ну, в автоматическом режиме и только на частных сборках - никак. Ведь частная сборка должна лежать в папке с программой или подпапке с именем сборки. Что означает, что любые две частные сборки с одним именем, но разными версиями перезапишут друг друга.

Получается, что частные сборки очень похожи (в плане ограничений) на обычное перенаправление DLL. В чём же преимущества частных сборок по сравнению с обычными DLL? (кстати, если указан манифест, то файл .local или папка local игнорируются)

- К сборкам автоматически применяется механизм версионности. Мы только что видели, как приложение отказалось грузить частную сборку, только потому, что она была другой версии.
- Сборки более защищены. Ведь DLL ищутся во многих путях. Особенно в PATH. А сборки - только в хранилище WinSxS (общие) и папке приложения (частные).
- Сборки могут реализовать registration free COM. Это значит, что их можно установить простым копированием без регистрации и элевации.

Но настоящая сила частных сборок получается, когда вы

осознаете, что вы вообще-то можете реализовать side-by-side на частных сборках - использованием ручной их загрузки с применением Activation Context API. С его помощью вы можете указать альтернативные пути поиска сборок для загрузки, выделив, таким образом, отдельное место для хранения разных версий одной частной сборки. Впрочем, этот механизм в этой статье не рассматривается. Ведь намного проще в этом случае использовать разделяемые сборки...

Практический пример: использование разделяемых сборок

Хорошая новость - глобальные разделяемые сборки почти ничем не отличаются от частных сборок. Единственное отличие - их надо подписать и установить в хранилище.

Чтобы подписать сборку, вам потребуется сертификат, **пригодный для цифровой подписи кода**. Кроме того, длина его ключей должна быть **не менее 2048 бит**. Итак, чтобы подписать сборку, вам нужно:

1. Использовать какую-нибудь утилиту для извлечения токена публичного (открытого) ключа сертификата (public key token). В состав Visual Studio/PSDK входят утилиты `pktextract.exe` и `sn.exe` - они пригодны для этой цели. В Delphi ничего такого нет, но вы можете взять их в MSDN/PSDK.
2. Вписать полученное значение в атрибут `publicKeyToken` элемента `assemblyIdentity` манифеста сборки.
3. Использовать утилиту `MT.exe` для создания хэшей файлов сборки и создать каталог-описание (.cdf файл).
4. Использовать утилиту `Makecat.exe` над созданным .cdf-файлом для создания защищённого каталога сборки.
5. Наконец, подписать созданный каталог вашим сертификатом, используя утилиту `SignTool.exe`. Файл .cdf может быть удалён.

Заметьте, что изменение файлов сборки или содержимого манифеста после подписи приведёт к блокировке сборки. Если вы меняли эти файлы после простановки подписи, то вам необходимо подписать сборку заново.

Как это выглядит на практике.

Начните с подготовки файлов сборки, манифеста сборки и сертификата, которым вы будете подписывать сборку. Ключ сертификата должен иметь длину не менее 2048 бит. Вам не

обязательно использовать сертификат, подписанный доверенным центром сертификации (trusted certificate). Сертификат необходим только для проверки того, что сборка не была изменена. Поэтому для подписи сборок обычно используются само-подписанные сертификаты. Вероятно, будет хорошей идеей делать отдельный сертификат для каждого вашего продукта.

Вот как вы можете создать само-подписанный сертификат с использованием Windows SDK:

```
C:\MySampleAssembly>makecert -pe -ss MY -$ individual -n "CN=в
аше-имя" -len 2048 -r
```

Вы можете использовать любое имя в параметре `CN=`, так что лучше всего там написать что-то понятное. Команда в примере выше создаст само-подписанный сертификат и автоматически добавит его в ваше персональное хранилище сертификатов, которое вы можете посмотреть в соответствующем апплете Панели управления:

```
certmgr.msc
```

Сгенерированный сертификат будет показан Windows с красным значком, что означает, что доверия к сертификату нет, потому что он не был выпущен доверенным центром, а является само-подписанным. Это может раздражать, но эта ситуация штатная и её можно игнорировать.

Запустите утилиту `Pktextract.exe` из Windows SDK, чтобы извлечь токен открытого ключа сертификата. Чтобы `Pktextract.exe` работала правильно, надо чтобы сертификат был в той же папке, что и утилита. Используйте полученное значение токена в атрибуте `publicKeyToken`.

Вот пример файла манифеста, названного `MySampleAssembly.manifest`. Сборка `MySampleAssembly` содержит только один файл: `MYFILE.DLL`. Заметьте, что значение атрибута `publicKeyToken` элемента `assemblyIdentity` должно быть изменено на значение, полученное от работы `Pktextract.exe`:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?
2  <assembly xmlns="urn:schemas-microsoft-com:asm.v1" man
3      <assemblyIdentity
4          type="win32"
5          name="Microsoft.Windows.MySampleAssembly"
6          version="1.0.0.0"
7          processorArchitecture="x86"
8          publicKeyToken="0000000000000000"/>
9      <file name="myfile.dll"/>
10 </assembly>
```

Теперь надо запустить `Mt.exe` из Windows SDK. Файлы сборки

должны лежать в той же папке, что и манифест. В этом примере все файлы лежат в каталоге `MySampleAssembly`. Запустите `Mt.exe` так:

```
C:\MySampleAssembly>mt.exe -manifest MySampleAssembly.manifest
 -hashupdate -makecdfs
```

Вот как будет выглядеть наш манифест после работы `Mt.exe`:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" man
    <assemblyIdentity
        type="win32"
        name=" Microsoft.Windows.MySampleAssembly"
        version="1.0.0.0"
        processorArchitecture="x86"
        publicKeyToken="0000000000000000"/>
    <file
        name="myfile.dll"
        hash="a1d362d6278557bbe965a684ac7adb4e57427a29
        hashalg="SHA1"/>
</assembly>
```

Опция `-hashupdate` создала нам атрибуты `hash` и `hashalg`, а опция `-makecdfs` создала файл `MySampleAssembly.manifest.cdf`.

Далее, мы запускаем `Makecat.exe` для этого .cdf-файла:

```
C:\MySampleAssembly>makecat MySampleAssembly.manifest.cdf
```

И последний шаг - запуск `SignTool.exe` для простановки цифровой подписи. Сертификат для подписи должен быть тем же самым, из которого вы извлекали токен открытого ключа в самом начале:

```
C:\MySampleAssembly>signtool sign /f mycompany.pfx /du http://
www.mycompany.com/MySampleAssembly /t http://timestamp.verisig
n.com/scripts/timstamp.dll MySampleAssembly.cat
```

Это стандартная команда подписи файла `MySampleAssembly.cat` вашим сертификатом `mycompany.pfx`.

Готово. Теперь остаётся только установить её.

Windows требует, чтобы разделяемые сборки устанавливались с помощью Windows Installer (MSI). Это гарантирует, что Windows сможет починить сборку, если она будет повреждена.

Устанавливаемая сборка сохраняется в подкаталоге папки `%systemroot%\WinSxS`. Манифест и .CAT файлы переименовываются Windows и копируются в папку `%systemroot%\WinSxS\Manifests`, а другие файлы сборки идут в отдельные каталоги с именем, сгенерированным на основе манифеста. К примеру, имя сборки Microsoft Windows Common Controls версии 6.0.10 - это:

```
x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.10.
0_x-ww_f7fb5805
```

Для дальнейшей информации - см. установка сборки с помощью Windows Installer. Кроме того, начиная с Windows Vista, вы можете вручную установить сборку, используя API Side-by-Side сборок (см. метод `IAssemblyCache.InstallAssembly`).

Рекомендации по использованию Side-by-Side

Разработчики общих библиотек должны рассмотреть вариант предоставления их компонент в виде разделяемых публичных сборок, если выполняется хотя бы один пункт:

1. Компонент предоставляет приложениям интерфейс, который может использоваться многими приложениями. К примеру, компоненты вроде MSHTML, который даёт приложениям доступ к объектной модели Dynamic HTML (DHTML).
2. Компонент уже используется многими приложениями. Пример - COMCTL32.
3. Компонент только что создан и является новым.
4. Компонент является компонентом режима пользователя, а не драйвером устройства.

Не всякий компонент пригоден для реализации в виде side-by-side сборки, к примеру:

1. Компонент обрабатываем взаимодействие между приложениями. К примеру, OLE32 не является хорошим кандидатом для side-by-side сборки, потому что вы бы не хотели иметь в системе несколько вариантов взаимодействие между приложениями.
2. Компонент, управляющий устройством (физическим или виртуальным).

При разработке сборок нужно учитывать следующие рекомендации:

- Правило №1: **никогда** не изменяйте разделяемую сборку после того, как вы её выпустили. Всегда увеличивайте номер версии для изменений.
- Ваши DLL должны быть спроектированы так, чтобы несколько их версий могли бы работать одновременно в одном процессе без конфликтов друг с другом. К примеру, многие приложения могут загружать плагины, причём каждому плагину требуется своя версия вашей библиотеки. Как разработчик разделяемой side-by-side

сборки, вы должны протестировать ваш компонент для работы в этих условиях.

- Если вы планируете предоставлять доступ к вашему компоненту на системах Windows 2000 и младше - вам необходимо будет устанавливать компонент как обычную DLL на старых системах. В этом случае вам нужно убедиться, что ваш компонент имеет максимальную обратную совместимость.
- Оцените использование различных объектов в вашем коде при одновременной работе. Определите, какие структуры данных должны присутствовать в одном экземпляре (и разделяться несколькими версиями вашего компонента), а какие должны быть раздельны. Подберите подходящий механизм обмена данными - проецируемые в память файлы, регистрация оконных сообщений и классов Windows, общая память, семафоры, мьютексы, драйвера устройств и так далее. Любая структура данных, которая используется несколькими версиями вашего компонента одновременно должна иметь обратную совместимость. Определите, когда общие данные требуют объектов синхронизации.
- Некоторые объекты, вроде оконных классов и атомов, являются уникальными по имени в процессе. Поэтому такие объекты должны именоваться, согласно версии сборки - если только вы не планируете разделять их с другими версиями вашего компонента. Если вы используете версионные идентификаторы, то используйте стандартный четырёх-числовой формат.
- Не добавляйте в DLL никакого кода само-регистрации. DLL в side-by-side сборке не может быть саморегистрирующейся.
- Определяйте все зависимые от версии имена в вашем коде через константы. К примеру - имена ключей реестра для хранения состояния/настроек. Когда вы выпускаете новую версию вашей сборки, то вы можете просто изменить одну константу, вместо того, чтобы править весь код. Например:

```
1  const
2    Version       = '1.0.0.0';
3    MyRegistryKey = 'MyAssembly ' + Version;
4    MyFolder      = 'MyAssembly ' + Version + '\';
```

- Храните любые временные данные в папке `Temp`.
- Не сохраняйте данные пользователя в глобальные хранилища. Храните данные приложения отдельно от данных пользователя.
- Присваивайте всем общим файлам номер версии.
- Присваивайте номер версии всем разделяемым сообщениям и структурам данным.
- Если вы добавляете новую функциональность, которая не

совместима со старой (другой двоичный интерфейс, etc), то вы должны выбрать новый идентификатор (CLSID, ProgId и имя файла), а не модифицировать существующую сущность. Будущие версии вашей side-by-side сборки будут использовать эти новые CLSID, ProgId или имена файлов. Это предотвратит конфликт между разными версиями DLL.

- Если вы заново используете существующий идентификатор сущности (CLSID, ProgId, etc), то протестируйте свою сборку на обратную совместимость.
- Храните настройки по умолчанию в коде вашей сборки. Не используйте для этого реестр.
- Проектируйте свои хранилища данных так, чтобы они были совместимы как в обратную, как и в прямую сторону. Версии могут меняться в любом порядке, к примеру: v1, затем v3, затем v2.
- Настройки в реестре должны записываться на версионной основе, чтобы изолировать сборку от влияния других версий этой же сборки. Проектируйте вашу side-by-side сборку так, чтобы корректно хранить и обрабатывать состояние сборки при сценариях с многими версиями вашей сборки.
- Любое состояние сборки, сохраняемое в реестре, должно быть изолировано от других версий сборки. Настройки состояния, хранимые в реестре, должны сохраняться в индивидуальные для версии места в реестре. Это справедливо как для HKCU, так и для HKLM. К примеру, вы можете хранить HKCU-часть настроек сборки версии XXXX в ключе реестра:

  ```
  HKCU\MyCompany\MyComponent\VersionXXXX
  ```

- Любая информация, хранимая в реестре частными сборками, должна сохраняться в индивидуальное для приложения место. К примеру, если сборка версии XXYY является частной сборкой приложения "SomeApplication", то вы можете хранить её настройки как:

  ```
  HKCU\MyCompany\MyComponent\VersionXXYY\SomeApplication
  ```

- В идеальном варианте вы должны реализовать персистентную модель, при которой приложение сохраняет настройки, а сборка не трогает реестр. При этом сборка должна предоставлять API загрузки/сохранения настроек, которое может вызвать приложение.

Практический пример: другие типы стандартных манифестов

Кратко рассмотрим ещё три примера использования стандартных манифестов, а также суммирующий пример. Только, пожалуйста, не надо бездумно включать эти манифесты в свою программу. Прочитайте сначала статьи по теме, чтобы ознакомиться с темой и понимать, что делают эти манифесты, когда их нужно использовать, а когда - не нужно.

Примечание: обратите внимание, что примеры использования манифестов в этом разделе уже не имеют отношения к side-by-side сборкам и изолированным приложениям - это уже расширения возможностей манифестов, появившиеся в Windows Vista и выше.

High DPI awareness: объявление, что ваше приложение в курсе про режимы экрана с высоким DPI

Когда приложение объявляет себя как DPI-aware ("я в курсе, что бывают высокие DPI"), то это эквивалентно утверждению, что приложение будет хорошо работать на высоких DPI (включая DPI с 200%, равный 150). Эта настройка не имеет силы на Windows XP и ниже. Начиная с Windows Vista и выше когда включается DPI виртуализация, приложения, которые не помечены на DPI-aware, автоматически масштабируются системой, а также они получают подстановочные эмулирующие данные (с учётом масштабирования) от системных API функций вроде `GetSystemMetric`.

Примечание: по умолчанию виртуализация DPI включается только на DPI выше 120 (125%).

Хотя в Win32 API есть функция для объявления приложения как DPI-aware (`SetProcessDPIAware`), её использование не поощряется, кроме специальных сценариев. В общем случае рекомендуется использовать манифест для объявления приложения DPI-aware.

Для этого вы должны добавить в манифест элемент `<dpiAware>`. Например:

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" man
  <asmv3:application>
    <asmv3:windowsSettings xmlns="http://schemas.micro
      <dpiAware>true</dpiAware>
    </asmv3:windowsSettings>
  </asmv3:application>
</assembly>
```

Примечание: если элемент `<dpiAware>` появляется в манифесте сборки DLL компонента, то этот элемент игнорируется. Только манифест приложения может указывать на "осведомлённость о

DPI".

UAC: указание требуемого уровня привилегий для приложения

В выпуске Windows Vista есть положения, позволяющие коду без манифеста или неподписанному коду работать с полным административным маркером доступа.

Примечание: в будущих версиях системы единственным способом для приложения запуститься с полными административными привилегиями будет указание подписанного манифеста.

В Windows Vista и выше используется такой формат в манифесте для указания требуемого уровня привилегий:

```
<requestedExecutionLevel
level="asInvoker|highestAvailable|requireAdministrator
uiAccess="true|false" />
```

Возможные значения и их трактовка:

Значение	Описание	Комментарий
asInvoker	Приложение запускается с тем же токеном, что и вызывающий процесс.	Рекомендуется для обычных приложений пользователя. Примеры программ: Блокнот, Калькулятор, Microsoft Word, Delphi, Total Commander.
highestAvailable	Приложение запустится с максимальными привилегиями, которые могут быть доступны пользователю.	Рекомендуется для приложений смешанного режима. Примеры программ: Консоль MMC, Редактор реестра.
requireAdministrator	Приложение всегда запускается с полным токеном администратора.	Рекомендуется только для приложений администраторов. Примеры программ: установщики программ.

Смысл значений `uiAccess`:

Значение	Описание
	Приложению не требуется управлять вводом в

False	пользовательский интерфейс другого окна на рабочем столе. Приложения, которые не предоставляют возможности accessibility, должны устанавливать этот флаг в False. Приложения, которым требуется управлять вводом в другие окна на рабочем столе (к примеру, экранные клавиатуры) должны устанавливать это значение в True.
True	Приложению разрешается обходить уровни контроля интерфейса пользователя, чтобы получать доступ к окнам с высокими привилегиями на рабочем столе. Эта настройка должна применяться только приложениями, реализующими Assistive Technology.

Примечание: приложения с uiAccess равным True должны иметь корректную цифровую подпись. Кроме того, приложение должно располагаться в защищённом месте системы. Сегодня такими местами являются папки \Program Files\ и \windows\system32\.

Пример манифеста:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" man
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <assemblyIdentity
        version="1.0.0.0"
        processorArchitecture="X86"
        name="ИмяКомпании.ИмяПродукта.ИмяПриложения"
        type="win32"
    />
    <description>Здесь - описание вашего приложения.</
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel
          level="requireAdministrator"
          uiAccess="false"/>
        </requestedPrivileges>
       </security>
  </trustInfo>
</assembly>
```

Указание на совместимость с конкретной ОС

Поведение операционной системы может меняться в каждой новой версии системы. Поэтому приложение, написанное для старой версии системы, может работать не совсем верно (или вообще не работать) в новой версии системы. Мы только что разобрали пример с UAC. К примеру, в Windows 7 было немного изменено поведение системы. По умолчанию приложения получают поведение системы как в Windows XP (если в манифесте нет информации о требуемом уровне привилегий или совместимости с ОС) или как в Windows Vista (если в манифесте

указана информация о требуемом уровне привилегий или задекларирована совместимость с Windows Vista). Чтобы получить поведение как в Windows 7 (без эмуляции), приложению нужно указать это явно.

Общий сценарий при этом:

1. Указать, что приложение поддерживает Windows *N* (где N - это Vista, 7 или иная более поздняя версия системы).
2. Проверить, что приложение нормально работает в Windows *N*.
3. Устранить все найденные проблемы при работе приложения в Windows *N*:
    - Если это сделать удалось, то *оставить* отметку о совместимости приложения с Windows *N*.
    - Если это сделать **не** удалось, то *убрать* отметку о совместимости приложения с Windows *N*.

Приложениям, которые поддерживают функциональность и Windows Vista, и Windows 7, не требуются отдельные манифесты. В этом случае они должны задекларировать свою совместимость с операционной системой добавлением секции `supportedOS` в блок `application`. Например:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" man
  <compatibility xmlns="urn:schemas-microsoft-com:comp
      <application>
        <!--Это значение Id указывает, что приложение
          <supportedOS Id="{e2011457-1546-43c5-a5fe-00
        <!--Это значение Id указывает, что приложение
          <supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a2
      </application>
  </compatibility>
</assembly>
```

Допустимыми значениями `Id` на сегодня являются:

- `{e2011457-1546-43c5-a5fe-008deee3d3f0}` для Windows Vista: это значение по умолчанию, если иного не указано явно.
- `{35138b9a-5d96-4fbd-8e2d-a2440225f93a}` для Windows 7: необходимо указывать явно, если приложению требуется поведение Windows 7.

Microsoft будет создавать новые GUID по мере выхода новых версий Windows при необходимости.

Список изменений в поведении для Windows 7 по сравнению с Windows Vista можно увидеть в статье MSDN Windows 7 / Application Manifests.

Собираем всё вместе

Когда вам нужно использовать несколько указаний из манифестов в своём приложении (как стандартных, так и своих), вам не нужно создавать несколько манифестов - ведь манифест может быть только один. Вместо этого вам нужно просто выписать элементы из нескольких манифестов в один, например:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
  <assembly xmlns="urn:schemas-microsoft-com:asm.v1" m
    <assemblyIdentity
        version="1.0.0.0"
        processorArchitecture="X86"
        name="ИмяКомпании.ИмяПродукта.ИмяПриложения"
        type="win32"
    />
    <description>Здесь - описание вашего приложения.</

    <dependency>
        <dependentAssembly>
            <assemblyIdentity
                type="win32"
                name="Microsoft.Windows.Common-Control
                version="6.0.0.0"
                publicKeyToken="6595b64144ccf1df"
                language="*"
                processorArchitecture="*"
            />
        </dependentAssembly>
    </dependency>

    <compatibility xmlns="urn:schemas-microsoft-com:co
      <application>
          <supportedOS Id="{e2011457-1546-43c5-a5fe-00
          <supportedOS Id="{35138b9a-5d96-4fbd-8e2d-a2
      </application>
    </compatibility>

    <trustInfo xmlns="urn:schemas-microsoft-com:asm.v2
      <security>
        <requestedPrivileges>
          <requestedExecutionLevel
            level="asInvoker"/>
        </requestedPrivileges>
       </security>
    </trustInfo>

    <asmv3:application>
      <asmv3:windowsSettings xmlns="http://schemas.mic
        <dpiAware>true</dpiAware>
      </asmv3:windowsSettings>
    </asmv3:application>

  </assembly>
```

Заключение

Хотелось бы только отметить не упомянутое здесь расширение манифестов: технология ClickOnce, предназначенная для создания само-обновляемых приложений, работающих с минимальным взаимодействием с пользователем (название технологии происходит от "установка в один щелчок"). Ещё одним примером, с которым, правда, вы не встретитесь в Delphi,

является подключение сборки C++ run-time - аналогично тому, как мы подключали сборку ComCtl32 версии 6. Например:

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" man
  <dependency>
    <dependentAssembly>
      <assemblyIdentity
          type="win32"
          name="Microsoft.VC90.CRT"
          version="9.0.xxxxx.y"
          processorArchitecture="x86"
          publicKeyToken="1fc8b3b9a1e18e3b"
      />
    </dependentAssembly>
  </dependency>
</assembly>
```

Кроме того, за кадром статьи остались также многие темы работы с разделяемыми сборками, включая управление конфигурацией, registration free COM, перенаправление сборок, использование ресурсных сборок и т.п., а также ручное управление контекстами. Тем не менее, я надеюсь, что эта статья помогла вам ознакомится с темой зависимостей в DLL и манифестов. Пусть она служит вам отправной точкой в этой теме.

Кстати говоря, если вы столкнётесь с проблемами при поиске/загрузке сборок в своих программах, то утилита командной строки `sxstrace` поможет вам их решить.

См. также:

- MSDN: Dynamic-Link Library Search Order.
- MSDN: Dynamic-Link Library Redirection.
- MSDN: Dynamic-Link Library Security.
- MSDN: SetSearchPathMode function.
- MSDN: SetDllDirectory function.
- MSDN: LoadLibrary function.
- MSDN: Isolated Applications and Side-by-side Assemblies.
- MSDN: Assembly Searching Sequence.
- MSDN: Installing Side-by-side Assemblies as Shared Assemblies.
- MSDN: Installing Win32 Assemblies for Side-by-Side Sharing on Windows XP.
- MSDN: Installing Win32 Assemblies for the Private Use of an Application on Windows XP.
- MSDN: Side-by-Side Assembly API.
- MSDN: Supported Microsoft Side-by-side Assemblies - список Side-by-side сборок в Windows XP и Windows 2003.
- MSDN: Configuration - изменение зависимостей side-by-side сборок и изолированных приложений после их развёртывания.
- MSDN: Manifest Files Reference.
- MSDN: Activation Context Reference.

- MSDN: Visual Styles.
- MSDN: User Interface - High DPI Awareness.
- MSDN: High DPI.
- MSDN: User Account Control.
- MSDN: Developing secure applications.
- MSDN: Windows 7 / Application Manifests.
- MSDN: ClickOnce Security and Deployment.
- Использование `sxstrace` для диагностики проблем с загрузкой сборок.

-= АЛЕКСАНДР АЛЕКСЕЕВ =- 20:04 ТЭГИ 7, DELPHI, VISTA, WINDOWS, СТАТЬЯ

5 КОММЕНТАРИЙ(ЕВ):

Егор О. комментирует...

Отлично Алекс, спасибо за твой титанический труд. Впрочем, как всегда в твоих трудах ;)

11 ФЕВРАЛЯ 2011 Г., 22:54

Анонимный комментирует...

Спасибо, работа отличная!

12 ФЕВРАЛЯ 2011 Г., 16:01

Анонимный комментирует...

Прекрасная статья !

Практически вся основная информация собрана в одном месте. Кстати, я сталкивался с тем, что при двоичном изменении (патчинге) общих сборок, находящихся в WinSxS, они продолжали штатно загружаться и работать, никакой блокировки не было.

Автору - большое спасибо.

Олег.

6 АПРЕЛЯ 2012 Г., 19:48

Анонимный комментирует...

Примечание: приложения с uiAccess равным True должны иметь корректную цифровую подпись. Кроме того, приложение должно располагаться в защищённом месте системы. Сегодня такими местами являются папки \Program Files\ и \windows\system32\.

Пример манифеста: 1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

Здесь - описание вашего приложения.

Косяк в описаии

7 СЕНТЯБРЯ 2012 Г., 15:23

Александр Алексеев комментирует...

Не вижу, где косяк?

7 СЕНТЯБРЯ 2012 Г., 15:25

ОТПРАВИТЬ КОММЕНТАРИЙ

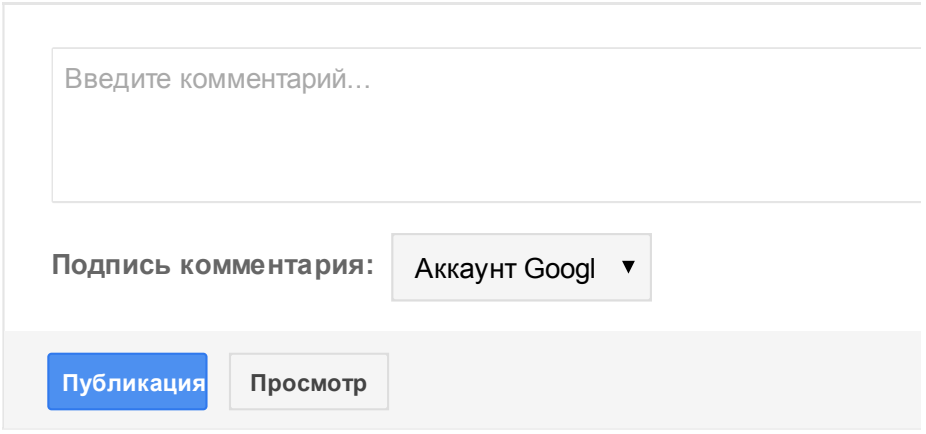
Можно использовать некоторые HTML-теги, например:

Жирный
Курсив
Ссылка

Вам необязательно регистрироваться для комментирования - для этого просто выберите из списка "Анонимный" (для анонимного комментария) или "Имя/URL" (для указания вашего имени и (опционально) ссылки на сайт). Все прочие варианты потребуют от вас входа в вашу учётку (поддерживается OpenID).

Пожалуйста, по возможности **используйте "Имя/URL" вместо "Анонимный"**. URL можно просто не указывать.

Ваше сообщение может быть помечено как спам спам-фильтром - не волнуйтесь, оно появится после проверки администратором.



ССЫЛКИ

Создать ссылку

Следующее Главная страница Предыдущее

NEVER SEND A HUMAN TO DO A MACHINE'S JOB