HIGH TECH

Delphi

Профессиональное программирование

Дмитрий Осипов

СИМВОЛ®

Санкт-Петербург — Москва
2006

Серия «High tech»

Дмитрий Осипов

Delphi. Профессиональное программирование

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Редактор	*А. Петухов*
Художник	*В. Гренда*
Корректор	*О. Макарова*
Верстка	*Н. Гриценко*

Осипов Д.
Delphi. Профессиональное программирование. – СПб.: Символ-Плюс, 2006. – 1056 с., ил.
ISBN 5-93286-074-X

Книга Д. Осипова «Delphi. Профессиональное программирование» принципиально отличается от стандартных изданий на эту тему. Это и не скороспелое «полное» руководство по очередной версии Borland® Delphi™, и не рядовой справочник, содержащий перевод файлов помощи к среде программирования. Идея книги в другом. Автор системно и последовательно излагает концепцию Delphi, предоставляя читателю не просто инструмент, а профессиональную методику, позволяющую разрабатывать эффективные приложения для Windows.

Книга рассчитана на подготовленного пользователя ПК, желающего самостоятельно научиться программировать и разрабатывать приложения и базы данных в среде Delphi. Опытные программисты смогут использовать издание как справочник. В тексте подробно описаны более 80 компонентов VCL, функции Object Pascal и Win32 API. В первой части книги излагаются основы языка программирования Delphi, подробно рассматриваются библиотека визуальных компонентов и процесс разработки собственных компонентов, изучаются динамически подключаемые библиотеки, процессы, многопоточные приложения, особенности межпрограммного взаимодействия, программирование на Win32 API, особенности построения сетевого программного обеспечения, технологии COM и OLE-automation. Вторая часть книги посвящена проектированию и созданию реляционных баз данных. Рассматриваются реляционная модель данных и язык SQL, изучаются компоненты доступа к данным и отображения данных, базирующиеся на механизмах BDE, ADO и InterBase.

ISBN 5-93286-074-X

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 30.03.2006. Формат 70x100 1/16 . Печать офсетная.
Объем 66 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Введение

Не многие области науки могут похвастаться таким бурным развитием, какое претерпели за свою сравнительно недолгую историю существования электронно-вычислительная техника и шагающие с ней рука об руку языки программирования. Не так давно самые первые программы писались на языке машинных команд. Это был поистине каторжный труд. Программист тех старозаветных времен не просто знал язык первых машин, он обладал глубокими инженерными знаниями архитектуры электронно-вычислительной машины (ЭВМ), системы команд процессора, организации памяти и многого другого. Такой высококлассный специалист ценился на вес золота, а производительность его работы была до смешного мала. Процесс создания элементарной программы отдаленно напоминал шаманские обряды (кто видел перфоратор, тот меня поймет), а про программистов слагались легенды.

Такое положение вещей мало кого устраивало, посему учеными предпринимались активные попытки хотя бы в какой-то степени «очеловечить» язык машин. Первым успехом в этом направлении было создание компиляторов с языков ассемблера. Язык низкого уровня ассемблер по-прежнему был очень близок к машинным командам, однако в нем уже отдаленно просматривались и человеческие черты. В ассемблере машинным командам соответствовали англоязычные мнемонические коды. Синтаксис языка не отличался особой изысканностью – каждая команда ассемблера могла включать три элемента: поле метки, код операции и поле операндов. Не так густо, но по сравнению с машинными командами это был настоящий прорыв.

Хотя появление ассемблера и соответствующих компиляторов несколько упростило работу программиста, но, по сути, язык мнемокодов все еще значительно отличался от языка общения людей. Однако, без всякого сомнения, можно утверждать, что ассемблер некоторым образом расширил круг программистов и (к сожалению) снизил требования к их инженерной подготовке, скажем, с уровня шамана до уровня вождя племени. Но взамен было получено ощутимое преимущество: разработка программы на ассемблере ускорилась если не на порядок, то, по крайней мере, в разы. Подчеркну еще один немаловажный факт: умение творить на ассемблере не стало анахронизмом и актуально до сих пор, в особенности в области системного программного обеспечения.

Эра превосходства умеющих общаться с ЭВМ шаманов и вождей над обычным людом длилась совсем недолго. Конец неравенству положили новые языки высокого (третьего) уровня. Хотя новые системы программирования по-прежнему представляли собой компромисс между языком машин и людей, но они уже стали доброжелательными, наполнились существенным словарным запасом, плюс ко всему семантика конструкций существенно приблизилась к обычным человеческим фразам. Благодаря всем этим преимуществам, лишь прочитав введение в язык PL/1, ALGOL, ADA, Fortran или во что-нибудь еще, уверенные в своих силах студенты в два счета переводили в состояние ступора ЭВМ любой степени надежности.

Девяностые годы прошлого века ознаменовались рождением языков 4-го поколения – 4GL (fourth generation languages). В них впервые вместо скучных строк кода программист получил удивительную возможность оперировать графическими, интуитивно понятными образами, а для создания элементарного приложения стало достаточно лишь несколько раз щелкнуть кнопкой мыши. В одно время даже раздавались восторженные возгласы о том, что программирование стало доступным для домохозяек...

Не знаю, хорошо это или плохо, но создание профессионального программного продукта и в наши дни по-прежнему требует от человека глубоких и разносторонних знаний, терпения и внимательности, находчивости и сообразительности и, если не таланта, то, по крайней мере, творческой одаренности, потому что программирование уже давно перестало быть просто наукой – это уже и искусство.

В настоящей книге рассматривается один из безусловных лидеров среди современных систем программирования – среда программирования Delphi. Это глубоко продуманный, высокоэффективный и (что немаловажно) весьма удобный программный продукт, позволяющий создавать приложения практически любой сложности, предназначенные для работы под управлением операционных систем Microsoft® Windows® и Linux.

Изначально Delphi специализировалась только на создании программного обеспечения под Windows. Для этого среда снабжена глубоко проработанной и эффективной библиотекой визуальных компонентов (VCL, Visual Components Library), элементы которой не только инкапсулировали в себе функции прикладного программного интерфейса (API, Application Program Interface) Windows, но и внесли существенные усовершенствования. Благодаря этому библиотека VCL успешно конкурирует с библиотекой MFC (Microsoft Foundation Class), разработанной в корпорации Microsoft, и служит фундаментом альтернативным Microsoft Visual Studio средам программирования Borland Delphi и Borland C++.

В условиях жесткой конкуренции фирма Borland постоянно развивает и улучшает возможности среды разработки. Начиная с шестой версии Delphi в состав среды разработки вошел пакет кроссплатформенной разработки CLX (Borland Component Library for Cross-Platform), основанный на идеях, апробированных в VCL. Delphi 2005 впитала идеи создания распределенных программных продуктов, базирующихся на архитектуре Microsoft® .NET Framework.

Особенность пакета CLX в том, что он позволяет строить приложения не только для Windows, но и для набирающей обороты ОС Linux. Тем самым программисты Delphi получили еще одно существенное преимущество – переносимость приложений между разными операционными системами. Однако за универсальность платформы пришлось заплатить – приложения CLX вынуждены отказаться от вызова функций, специфичных для каждой из операционных систем. В связи с этим опора на CLX не столь рациональна в тех случаях, когда вы нацелены только на работу с Windows. Поскольку настоящая книга посвящена программированию для Windows, то мы больше не будем возвращаться к CLX и системе Kylix (дополнение к Delphi для работы с CLX).

Перечисляя заслуги Delphi, стоит упомянуть доступность и интуитивную понятность интерфейса среды, наглядность кодовых конструкций языка Object Pascal, надежную систему выявления ошибок, высокоэффективный компилятор, умение поддерживать самые распространенные форматы баз данных и многое другое.

Соглашения, принятые в книге

Впервые встречающиеся термины выделены полужирным шрифтом, а элементы интерфейса Delphi – шрифтом OfficinaSans.

Моноширинным шрифтом выделены имена файлов, переменных, констант, массивов, записей, методов, классов, свойств, процедур, функций, модулей и библиотек, а также код примеров и синтаксические конструкции. Зарезервированные слова выделены моноширинным полужирным шрифтом.

Для акцентирования внимания читателя на ключевых частях материала текст выделяется следующим образом:

На заметку.
Данный материал – советы, комментарии или замечания – следует принять к сведению.

Внимание!
Текст, отмеченный восклицательным знаком, однозначно описывает действия программиста в той или иной ситуации.

Стоп!
Однозначный запрет; внимание заостряется на характерных ошибках. Короче говоря, никогда так не делайте.

Такой картинкой отмечено описание методов из состава прикладного интерфейса пользователя Windows 32 API. Кроме того, они «переведены» с языка C на язык Pascal.

Подробный предметный указатель позволяет найти в тексте интересующий вас класс, компонент, свойство и метод по их названию.

21

Службы Microsoft Windows NT

В операционных системах Microsoft Windows NT 4.0/2000/XP/2003, построенных на основе технологии NT (New Technology), наряду с обычными процессами трудятся *службы*, или *сервисы* NT (*services*). Служба – это низкоуровневый системный процесс, выполняющий функции по поддержке других, протекающих в операционной системе процессов. Как правило, службы не предоставляют пользователю никакого графического интерфейса. Они запускаются автоматически в момент загрузки компьютера или вручную из консоли управления компьютером. Старт сервиса может не зависеть даже от того, вошел пользователь в систему или нет, поэтому службы NT просто незаменимы при построении различного рода автономно работающих серверов.

В самом общем случае говорят о существовании двух видов служб: *служб ядра* (kernel-mode services) и обычных *системных служб* Win32. Вопросам программирования драйверов можно посвятить отдельную книгу, поэтому в этой главе мы скромно умолчим о них и все силы сосредоточим на изучении служб процессов Win32 и на вопросах управления службами из внешних программ.

Излагаемый далее материал разделен на две части. Сначала мы познакомимся с внешней стороной сервисов, для чего рассмотрим ряд системных функций, предназначенных для установки, анализа, конфигурирования, управления и удаления уже готовых системных служб. Вторая половина главы посвящена общим вопросам разработки системных служб средствами Delphi.

Администрирование служб в Windows NT

Для того чтобы управлять службами, пользователь должен обладать правами администратора. В ОС Windows 2000/2003/XP доступ к консоли управления службами обеспечивается из папки Администрирование. В этой папке можно выбрать ярлык Службы либо обратиться к ярлыку Управление компьютером. Окно консоли управления службами в компьютере, работающем под ОС Windows XP Professional, представлено на рис. 21.1.

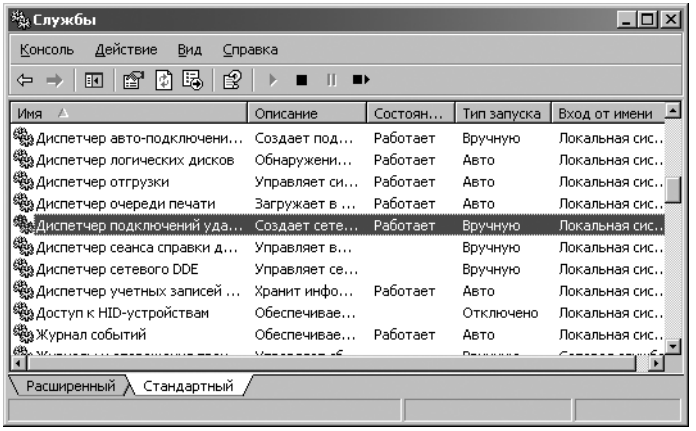

Рис. 21.1. *Консоль управления службами станции с ОС Windows XP Professional*

Кнопки панели управления консолью позволяют останавливать, временно приостанавливать и запускать вновь установленные на станции службы. Двойной щелчок по строке с названием службы вызовет диалоговое окно настройки выбранного сервиса. Здесь можно определить особенности запуска сервиса или вообще отказаться от его услуг.

Не имея достаточных знаний о роли той или иной службы на локальной или удаленной станции, ни в коем случае не останавливайте ее работы! Это может негативно отразиться на работе других служб или операционной системы в целом.

Управление службами из внешних приложений

В библиотеке визуальных компонентов Delphi системная служба инкапсулирована в отдельном классе `TService`, а процесс (владелец этой службы) создается из класса `TServiceApplication`. Пользуясь услугами перечисленных классов, программист без особых хлопот может разработать вполне жизнеспособный сервис. При этом ему даже нет необходимости глубоко вникать в суть проблемы (получается, что основное достоинство VCL Delphi мы превратили в недостаток). Именно поэтому предлагаю начать изучение системных служб не с наших горячо любимых классов VCL, а с вопроса управления уже готовыми службами из внешних приложений.

Соединение с менеджером системных служб Windows

В ОС Windows, построенных на фундаменте NT, управлением сервисами ведает *менеджер управления службами* (Service Control Manager, SCM). Именно на него возложены посреднические функции между программами, управляющими службами, и самими службами (рис. 21.2). Менеджер служб инициализируется на первоначальном этапе загрузки операционной системы, а заканчивает свою работу на самых последних секундах ее выполнения.

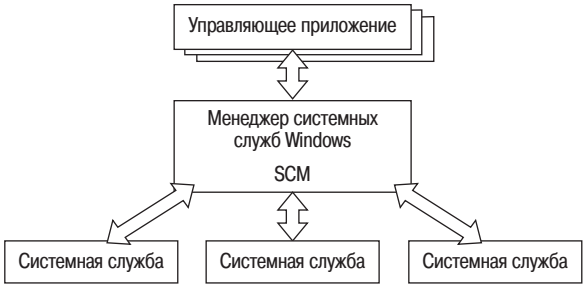

***Рис. 21.2.** Место менеджера системных служб в модели управления сервисами Windows NT*

Сразу после собственной инициализации SCM берет в свои руки бразды правления службами: управляет процессом загрузки всех автоматически подключаемых служб. Во время работы операционной системы менеджер передает команды от управляющих программ (например, консоли управления сервисами) к имеющимся в его подчинении службам. К перечню основных команд относятся:

- команды инсталляции и деинсталляции служб;
- команды запуска, приостановки, возобновления выполнения или полной остановки службы;
- команды анализа состояния и изменения конфигурационных характеристик службы.

Менеджер SCM отвечает за конфигурационную базу данных служб системы, а всю информацию о зарегистрированных в операционной системе сервисах он хранит в системном реестре Windows.

Для того чтобы организовать работу со службами компьютера, прежде всего необходимо получить идентификатор менеджера служб Windows. С этой задачей превосходно справляется метод Win32 API:

Win32 API
```
function OpenSCManager(lpMachineName, lpDatabaseName : pAnsiChar;
dwDesiredAccess : cardinal) : THandle;
```

Здесь `lpMachineName` – указатель на сетевое имя интересующего нас компьютера, которое должно начинаться с двух обратных слэшей: `\\NetComputer`. Если речь идет о нашей собственной станции, то в этот параметр передаем неопределенный указатель `nil`. Второй параметр `lpDatabaseName` представляет собой указатель на строку с именем базы данных менеджера управления службами. На данный момент для инициализации этого параметра предусмотрена одна-единственная константа `SERVICES_ACTIVE_DATABASE`, поэтому во второй параметр тоже можно передать `nil`. Последний параметр `dwDesiredAccess` определяет наши права доступа. Напомню, что самые большие права, права администратора компьютера, предоставит константа `SC_MANAGER_ALL_ACCESS`. В случае успешного выполнения функция возвращает указатель на менеджер служб.

```
uses WinSvc;
…
```

```
var hSCM : THandle;
begin
    hSCM:=OpenSCManager(nil, nil, SC_MANAGER_ALL_ACCESS);
    try
    //операции с менеджером
    finally
    CloseServiceHandle(hSCM);
    end;
end;
```

После того как необходимость в услугах менеджера служб Windows (или службы) отпадет, обязательно освободите его дескриптор, воспользовавшись методом:


```
function CloseServiceHandle(hSCObject : THandle) : LongBool;
```

Общие вопросы регистрации и конфигурирования службы

Вне зависимости от того, какую задачу станет решать проектируемая нами служба (индексировать файлы, обслуживать сетевой ресурс и т. д.), можно выделить самые общие вопросы, на которые должен ответить программист на первом этапе описания будущего сервиса. Все 12 вопросов, ответы на которые должны быть переданы в функцию `CreateService()` во время установки сервиса, представлены в табл. 21.1. Вопросы следуют не по порядку объявления параметров в функции, а сгруппированы по категориям. Это сделано потому, что к этим параметрам мы еще не раз вернемся при обсуждении свойств и методов класса `TService`.

Таблица 21.1. Общие вопросы конфигурирования службы и параметры метода CreateService()

№	Категория	Параметр	Описание
1.	**Тип службы**	`ServiceType`	Параметр определяет **тип службы**: служба ядра или служба Win32. Кроме того, от значения параметра зависит возможность взаимодействия с сервисом любого зарегистрированного в системе пользователя, т. е. интерактивность службы.
2.	**Идентификация службы в системе**	`ServiceName`	**Имя службы**, идентифицирующее сервис в системе и используемое в ряде функций Win32 API для обращения к службе.
3.		`DisplayName`	**Название службы**, которое будет использоваться для отображения в программах управления службами. Например, в консоли управления службами (рис. 21.1) название сервиса выводится в колонке Имя.
4.		`BinaryPathName`	**Путь к исполняемому файлу** сервиса.
5.	**Определение прав**	`ServiceStartName`	Определяет **учетную запись** Windows, от имени которой запускается служба.

Таблица 21.1 (продолжение)

№	Категория	Параметр	Описание
6.	**Определение прав**	Password	Задает **пароль** для учетной записи lpServiceStartName.
7.		DesiredAccess	**Ограничение на доступ к сервису.** Прежде чем менеджер служб разрешит внешнему процессу обратиться к сервису, он проверит наличие у него соответствующих прав.
8.	**Особенности загрузки службы**	Dependencies	**Зависимости.** Перечень служб, от которых зависит наш сервис.
9.		LoadOrderGroup	**Группа загрузки.** Указывает группу, вместе с которой будет произведена загрузка службы.
10.		TagId	**Порядковый номер** в группе загрузки.
11.		StartType	**Способ запуска** службы. Определяет, будет ли служба автоматически запускаться при старте системы или она стартует в ручном режиме (после вызова из внешней программы).
12.	**Реакция на ошибки**	ErrorControl	**Реакция на ошибки** во время запуска. Определяет поведение системы при сбоях во время автоматического старта службы, происходящего в процессе загрузки системы.

Наконец мы добрались до функции Win32 API, отвечающей за установку сервиса в системе:

Win32 API
```
function CreateService(SCManager : THandle; ServiceName, DisplayName :
pAnsiChar; DesiredAccess, ServiceType, StartType, ErrorControl :
Cardinal; BinaryPathName, LoadOrderGroup : pAnsiChar; TagId : Cardinal;
Dependencies, ServiceStartName, Password : pAnsiChar) : THandle;
```

Единственный параметр, который мы с вами не успели обсудить, – это указатель на менеджер служб Windows SCManager.

Если Вы планируете обращаться к методам Win32 API, связанным с обслуживанием системных служб, то позаботьтесь, чтобы в строке uses *проекта был указан модуль* WinSvc. *В свою очередь исходный код классов VCL, инкапсулирующих службы NT, сосредоточен в модуле* SvcMgr. *Этот модуль автоматически подключается к проекту при создании приложения-службы.*

Получение указателя службы

Если сервис уже установлен в операционной системе, то для работы с ним из внешнего приложения необходимо получить его указатель. Для этого предназначена функция:

Win32 API
```
function OpenService(SCManager : THandle; ServiceName : pAnsiChar;
DesiredAccess : Cardinal) : THandle;
```

Здесь SCManager – полученный усилиями функции OpenSCManager() указатель менеджера служб, ServiceName – указатель на текстовую строку с именем сервиса и DesiredAccess – желательные права доступа к сервису. При удачном стечении обстоятельств функция вернет указатель затребованной нами службы.

Таблица 21.2. Перечень прав доступа в Windows 2000

Учетная запись	Права доступа
Everyone (Все) или для Windows XP **Authenticated users** (пользователи, прошедшие аутентификацию)	SC_MANAGER_CONNECT SC_MANAGER_ENUMERATE_SERVICE SC_MANAGER_QUERY_LOCK_STATUS STANDARD_RIGHTS_READ
LocalSystem (локальная система)	SC_MANAGER_CONNECT SC_MANAGER_ENUMERATE_SERVICE SC_MANAGER_MODIFY_BOOT_CONFIG SC_MANAGER_QUERY_LOCK_STATUS STANDARD_RIGHTS_READ
Administrators (администраторы)	SC_MANAGER_ALL_ACCESS

Запуск службы

Для запуска службы из внешнего приложения используют функцию:

Win32 API
```
fucntion StartService(Service : THandle; NumServiceArgs : Cardinal;
var ServiceArgVectors : pAnsiChar) : LongBool;
```

Первый параметр функции определяет указатель интересующего нас сервиса, второй параметр NumServiceArgs – это количество строк в массиве аргументов, на который указывает третий по счету параметр ServiceArgVectors. В простейшем случае для запуска сервиса нам понадобится только его указатель.

```
var hSCM, hSrv : THandle;
    SStatus :_SERVICE_STATUS;
    args : pAnsiChar;
const SrvName='DemoService1';
begin
    hSCM:=OpenSCManager(nil, nil, SC_MANAGER_ALL_ACCESS);
    hSrv:=OpenService(hSCM,SrvName,SC_MANAGER_ALL_ACCESS);
    if hSrv>0 then StartService(hSrv, 0, args);
    CloseServiceHandle(hSrv);
    CloseServiceHandle(hSCM);
end;
```

Управление службой

Обладая указателем сервиса, приложение вправе выдать команду на его приостановку, возобновление и завершение работы. Для этого применяется метод Win32 API:

Win32 API
```
function ControlService(Service : THandle; Control : Cardinal; var
Service_Status : _SERVICE_STATUS) : LongBool;
```

Первый параметр метода – указатель сервиса. Параметр Control определяет, какие именно действия выполняются с сервисом (см. табл. 21.3). Последний параметр Service_Status – это указатель на объявленную в модуле WinSvc структуру _SERVICE_STATUS, хранящую основные характеристики службы. В случае корректного завершения функция ControlService() возвратит значение true.

Таблица 21.3. Константы кодов Control метода ControlService( ) для работы в Windows NT

Константа	Описание
SERVICE_CONTROL_STOP	Остановка сервиса.
SERVICE_CONTROL_PAUSE	Временная остановка сервиса.
SERVICE_CONTROL_CONTINUE	Продолжение работы сервиса после временной установки.
SERVICE_CONTROL_INTERROGATE	Запрос текущего состояния сервиса.

Например, для остановки сервиса hSCM потребуется такая строка кода:

```
ControlService(hSCM, SERVICE_CONTROL_STOP, SStatus);
```

Построение списка служб системы

Для разработчика системного ПО значительный интерес представляет функция, позволяющая построить список установленных в системе служб:


```
function EnumServicesStatus(SCManager : THandle; ServiceType, ServiceState
: Cardinal; var Services : _ENUM_SERVICE_STATUSA; BufSize : Cardinal;
var BytesNeeded, ServicesReturned, ResumeHandle : Cardinal) : LongBool;
```

Здесь SCManager – указатель менеджера служб. Параметр ServiceType задает тип служб. В него могут быть переданы флаги SERVICE_DRIVER и SERVICE_WIN32. В первом случае функция будет собирать информацию о службах ядра, а во втором – о системных службах. Допускается использовать оба флага одновременно.

Третий параметр ServiceState предназначен для фильтрации служб по их состоянию. В него допускается передавать одну из трех констант: SERVICE_INACTIVE – отбирать остановленные службы, SERVICE_ACTIVE – отбирать исполняющиеся службы, SERVICE_STATE_ALL – отбирать остановленные и активные службы.

Параметр Services – указатель на структуру, принимающую данные об имеющихся службах:

```
PEnumServiceStatus = PEnumServiceStatusA;
_ENUM_SERVICE_STATUSA = record
  lpServiceName: PAnsiChar;         //указатель на строку с именем сервиса
  lpDisplayName: PAnsiChar;         //указатель на строку с названием сервиса
  ServiceStatus: TServiceStatus;    //запись со статусом сервиса
end;
```

Параметр BufSize задает размерность приемной структуры; BytesNeeded – переменная, в которую будет занесена информация о требуемом размере при-

емного буфера; ServicesReturned – переменная, в которую метод возвратит информацию о количестве служб, выполняющихся в системе.

Прежде чем мы перейдем к последнему параметру ResumeHandle, стоит рассказать о специфике применения функции EnumServicesStatus(). Особенность заключается в том, что перед вызовом функции программист не знает, какой размер приемного буфера ему понадобится для сбора информации обо всех имеющихся в системе сервисах. Исходя из этого различают два варианта сбора информации о службах системы.

Первый вариант заключается в двукратном вызове метода EnumServicesStatus(). Первый вызов предназначен лишь для того, чтобы заставить функцию вернуть в параметр BytesNeeded требуемый размер приемного буфера. Поэтому при первом вызове функции в параметр BufSize, определяющий размер приемного буфера, направляем нулевое значение. Выяснив требуемую размерность буфера, вызываем функцию повторно и собираем все данные о сервисах.

Второй вариант работы с EnumServicesStatus() основан на «построчном» чтении данных о сервисах. Для этого требуется задать такой размер буфера, в котором поместятся данные об одной службе:

```
BufSize:=SizeOf(_ENUM_SERVICE_STATUS) + 2*(255 + 1);
```

Размер единичной записи складывается из размерности структуры _ENUM_SERVICE_STATUS и размера, необходимого для хранения имени и названия службы (по 255 символов на имя плюс 1 для завершающего нулевого символа).

Вот теперь настал черед поговорить о последнем параметре функции – ResumeHandle. После вызова метода EnumServicesStatus() в переменную, связанную с параметром ResumeHandle, будет помещена текущая позиция в перечислении. При первом обращении к EnumServicesStatus() эту переменную мы инициализируем нулем. При последующих вызовах по значению указанной переменной функция будет знать, о каких службах данные уже собраны, и возвратит следующую порцию данных.

В случае успешного выполнения метод возвратит ненулевое значение.

Сразу после вызова функции EnumServicesStatus() *необходимо собрать данные об ошибках с помощью* GetLastError(). *Если метод вернет константу* ERROR_MORE_DATA *(значение 234), это означает, что метод* EnumServicesStatus() *собрал не все данные.*

Ниже предложен исходный код, претворяющий в жизнь второй вариант сбора данных о доступных в системе сервисах.

```
uses WinSvc;
…
procedure GetServiceList(SCManager : THandle; var ServiceNamesList : TStringList);
var pServices : PEnumServiceStatus;
    BufSize, BytesNeeded, ServicesReturned, ResumeHandle : Cardinal;
    Err : Cardinal;
begin
BufSize:=SizeOf(_ENUM_SERVICE_STATUS) + 2*(255 + 1);   //узнаем размер записи
```

```
GetMem(pServices, BufSize);          //распределяем память для приемного буфера

ResumeHandle:=0;                     //начало работы с первого отсчета
repeat
EnumServicesStatus(SCManager, SERVICE_WIN32 OR SERVICE_DRIVER, SERVICE_STATE_ALL,
pServices^, BufSize, BytesNeeded, ServicesReturned, ResumeHandle);
Err:=GetLastError();
ServiceNamesList.Add(pServices^.lpServiceName)
until Err<>ERROR_MORE_DATA;

FreeMem (pServices, BufSize);
end;
```

С целью сокращения исходного кода мы ограничились получением данных лишь об именах сервисов. Эти данные возвращаются процедурой в параметре `ServiceNamesList` в виде набора строк.

Начиная с Windows 2000 активно эксплуатируется расширенный вариант функции построения списка служб – `EnumServicesStatusEx()`*. В дополнение к возможностям* `EnumServicesStatus()` *усовершенствованный вариант функции позволяет перечислить службы, принадлежащие определенной группе загрузки.*

Исследование службы

Для получения информации о конфигурации службы программист может воспользоваться функцией:

```
function QueryServiceConfig(Service : THandle; ServiceConfig :
PQueryServiceConfig; BufSize : Cardinal; var BytesNeeded : Cardinal) :
LongBool;
```

Параметр `Service` – это указатель сервиса. Далее в функцию передается указатель на приемный буфер (структуру `ServiceConfig`), в который будут помещены результаты работы функции. Размер буфера определяется параметром `BufSize`. Последний параметр `BytesNeeded` возвращает пожелания метода по поводу размера буфера.

Все поля информационной структуры `_QUERY_SERVICE_CONFIGA` коррелируются с рассмотренными в начале главы параметрами метода `CreateService()`:

```
PQueryServiceConfigA = ^TQueryServiceConfigA;
PQueryServiceConfig = PQueryServiceConfigA;

 _QUERY_SERVICE_CONFIGA = record
  dwServiceType: DWORD;              //тип сервиса
  dwStartType: DWORD;                //тип запуска
  dwErrorControl: DWORD;             //реакция на ошибки при загрузке системы
  lpBinaryPathName: PAnsiChar;       //путь к исполняемому файлу
  lpLoadOrderGroup: PAnsiChar;       //группа загрузки
  dwTagId: Cardinal;                 //номер запуска в группе
  lpDependencies: PAnsiChar;         //зависимые службы
  lpServiceStartName: PAnsiChar;     //учетная запись
  lpDisplayName: PAnsiChar;          //отображаемое на экране название
end;
```

Ниже предложен пример, демонстрирующий способ сбора данных о сервисе.

```
procedure GetServiceInfo(SCManager :THandle; SrvName : string);
var pConfig : PQueryServiceConfig;
    BytesNeeded : Cardinal;
    Srv : THandle;
begin
Srv:=OpenService(SCManager, SrvName, SC_MANAGER_ALL_ACCESS);   //доступ к сервису
{первый вызов функции для выяснения необходимого размера входного буфера}
QueryServiceConfig(Srv, nil, 0, BytesNeeded);
GetMem(pConfig,BytesNeeded);                          //инициализация входного буфера
{второй вызов функции с получением данных}
QueryServiceConfig(Srv, pConfig, BytesNeeded, BytesNeeded);
…
//АНАЛИЗ ПОЛУЧЕННЫХ ДАННЫХ
…
FreeMem(pConfig,BytesNeeded);                         //очистка памяти
CloseServiceHandle(Srv);                              //освобождение указателя сервиса
end;
```

Обратите внимание, что функция `QueryServiceConfig()` вызывается дважды. Первый вызов необходим для выяснения размера входного буфера. Реальные данные о сервисе с именем `SrvName` будут возвращены во время повторного вызова.

Начиная с Windows 2000 в состав Windows входит дополнительная функция сбора данных о сервисе `QueryServiceConfig2()`. *Она позволяет получить дополнительные параметры конфигурации службы, такие как описание службы и список действий при сбое службы.*

Конфигурирование службы

Для изменения основных параметров службы предназначена функция:

Win32 API

```
function ChangeServiceConfig(Service : THandle;  //указатель сервиса
  dwServiceType,                                 //тип сервиса
  dwStartType,                                   //особенности запуска
  dwErrorControl : Cardinal;                     //реакция на ошибки
  lpBinaryPathName,                              //путь к исполняемому файлу
  lpLoadOrderGroup : pAnsiChar;                  //группа загрузки
  lpdwTagId : cardinal;                          //номер в группе
  lpDependencies,                                //зависимые службы
  lpServiceStartName,                            //учетная запись
  lpPassword,                                    //пароль к учетной записи
  lpDisplayName : pAnsiChar                      //выводимое на экран название
) : LongBool;
```

На мой взгляд, параметры службы в дополнительных комментариях не нуждаются. Отмечу только, что если планируется изменить лишь некоторые из конфигурационных параметров функции, в неизменяемые параметры надо передавать константу `SERVICE_NO_CHANGE`.

Расширенная версия функции конфигурирования сервиса называется `ChangeServiceConfig2()`. *Кроме только что перечисленных возможностей новая функция позволяет задавать описание службы и список действий при сбое службы.*

Удаление службы из системы

Вся ирония программирования в том, что создаваемый бессонными ночами сервис-шедевр удаляется из системы одной строкой кода:

Win32 API
```
function DeleteService(Service : THandle) : LongBool;
```

Единственный параметр функции – указатель ненужной службы.

Инкапсуляция системной службы в VCL – класс TService

В библиотеке VCL служба NT представлена классом `TService`. При выборе его предка программисты Borland проявили остроумие и воспользовались услугами модуля данных `TDataModule` (рис. 21.3). Благодаря такому решению значительно упростился процесс программирования: службу можно «потрогать» руками и даже отдать в ее распоряжение ряд невизуальных элементов управления, например компоненты доступа к данным.

Тип службы

Тип службы определяется свойством `ServiceType`. Различают три типа: `stWin32` – сервис операционной системы, `stDevice` – сервис устройства и `stFileSystem` – сервис файловой системы.

```
property ServiceType: TServiceType;  //по умолчанию stWin32
```

Если вы хотите, чтобы после входа в систему со службой смог взаимодействовать любой пользователь, установите в `true` свойство:

```
property Interactive: Boolean;       //по умолчанию false
```

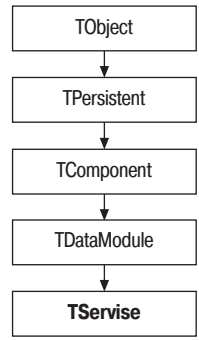

Рис. 21.3. *Иерархия наследования*

Интерактивная служба получает возможность создавать окна, которые будут видны пользователю, и устанавливать перехватчики сообщений. Включение флага поддержки интерактивности службы имеет смысл только тогда, когда мы создаем сервис операционной системы `stWin32`. Кроме того, интерактивный сервис должен выполняться в контексте системной учетной записи (`LocalSystem`).

Системные службы NT предназначены для фонового выполнения в отсутствие пользователя, поэтому, на мой взгляд, интерактивность – сомнительное достоинство сервиса и по возможности его следует избегать.

Идентификация службы в системе

Ключевое свойство, по которому будет идентифицироваться служба в системе, – ее имя:

```
property Name: TComponentName;
```

Название сервиса нам пригодится при его создании средствами Win32 API (функция CreateService) или при выяснении дескриптора (функция OpenService). Название сервиса, используемое менеджером служб с целью информирования пользователя, определяется в свойстве:

```
property DisplayName: String;
```

Это название будет отображаться в колонке Имя консоли управления службами (см. рис. 21.1).

Определение прав

Запуск сервиса всегда осуществляется от имени какой-либо зарегистрированной в системе учетной записи (см. колонку Вход от имени на рис. 21.1). Если все оставить на самотек, то служба стартует от имени системной учетной записи LocalSystem. Если же вы горите острым желанием дать бой политике безопасности операционных систем класса Windows NT, то можете поэкспериментировать со свойством:

```
property ServiceStartName: String;   //по умолчанию пусто
```

определяющим имя учетной записи. При задании учетной записи для службы требуется указать имя пользователя и пароль:

```
property Password: String;
```

Но будьте осторожны, потому что пароль сохраняется менеджером в реестре без проверки, а его корректность проверяется только в момент запуска сервиса. Специальные учетные записи (LocalSystem, LocalService и NetworkService) пароля не имеют.

Особенности загрузки и запуска службы

Корректное выполнение той или иной службы находится в прямой зависимости от того, запущены или нет сторонние сервисы, загружены ли необходимые драйверы. Если основная служба не загружена или остановлена, то это, безусловно, отразится и на выполнении зависимого от нее сервиса. В свою очередь с нашим сервисом также могут быть связаны вторичные службы. Менеджер служб Windows всегда отслеживает такие зависимости. Например, на рис. 21.4 представлено окно взаимосвязей службы диспетчера логических дисков компьютера с другими сервисами.

При определении сервиса необходимо указать список служб, от которых зависит сервис. Для этого предназначено свойство-контейнер:

```
property Dependencies: TDependencies;
```

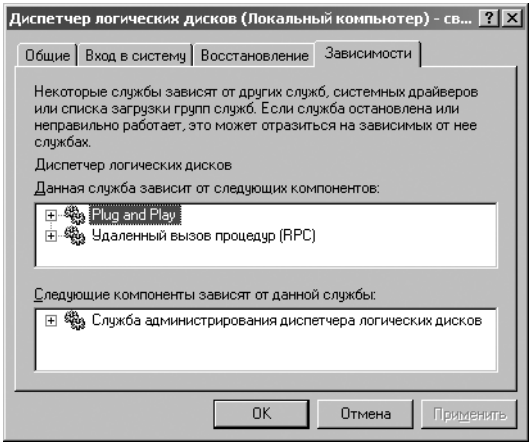

***Рис. 21.4.** Зависимости службы диспетчера логических дисков Windows*

Это массив, в роли элементов которого выступают объекты `TDependency` – ссылки на основные по отношению к нашему сервису службы.

```
property Items[Index: Integer]: TDependency; default;
```

При наличии явной зависимости между разрабатываемой нами службой и другими сервисами особую роль играет строгая очередность загрузки нашего и взаимосвязанных с ним сервисов. Для определения последовательности загрузки служб и системных драйверов в Windows введено понятие **групп загрузки**. При разработке службы программист указывает, с какой из групп следует загружать его сервис. Название группы определяется в свойстве:

```
property LoadGroup: String;
```

Службы из каждой группы стартуют только после того, как запустятся все автоматически стартующие службы из предыдущей группы. Сервисы, которые не принадлежат ни одной из групп загрузки, запускаются в последнюю очередь.

Список групп загрузки мы найдем в системном реестре:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\ServiceGroupOrder
```

Перечень включает несколько десятков наименований. При необходимости программист имеет право определить в реестре собственную группу загрузки.

Для того чтобы упорядочить очередность загрузки сервиса внутри группы загрузки, порядковый номер сервиса передается в свойство `TagID`. Это свойство используется только при загрузке служб ядра. Если загрузочный номер вам неизвестен, оставляйте в свойстве нулевое значение.

```
property TagID: Cardinal;          //по умолчанию 0
```

Особенности запуска службы определяются состоянием свойства:

```
property StartType: TStartType;    //по умолчанию stAuto
```

Предусмотрено пять вариантов старта сервиса, которые представлены в табл. 21.4.

Таблица 21.4. Возможные значения TStartType

Значение	Особенности старта службы	Примечание
stBoot	Служба запускается в момент загрузки ОС.	Только для служб ядра stDevice или stFileSystem.
stSystem	Служба стартует после загрузки и инициализации ОС.	
stAuto	Служба запустится автоматически во время загрузки ОС.	По умолчанию.
stManual	Служба стартует в ручном режиме (после вызова метода StartService).	
stDisabled	Служба отключена и может быть запущена только администратором.	

Определение поведения системы при ошибке запуска службы

Реакцию операционной системы на сбой в момент старта службы определяет свойство:

```
property ErrorSeverity: TErrorSeverity; //по умолчанию esNormal
```

Четыре возможных варианта поведения Windows представлены в табл. 21.5.

Таблица 21.5. Возможные значения TErrorSeverity

Значение	Поведение
esIgnore	Игнорировать ошибку. Данные об ошибке заносятся в журнал, служба продолжает выполняться.
esNormal	Ошибка заносится в журнал. На экран выводится сообщение об ошибке, после чего служба продолжает свою работу.
esSevere	Ошибка заносится в журнал. Выполнение сервиса продолжится только в случае, если система сможет загрузиться в режиме «последняя удачная конфигурация».
esCritical	Критическая ошибка. Если происходит отказ этой службы, ошибка заносится в журнал и система перезагружается в режиме «последняя удачная конфигурация».

События, связанные с инсталляцией и деинсталляцией службы

В роли инициатора регистрации службы может выступить внешнее приложение или процесс, владеющий этим сервисом. Сразу отмечу, что это особый вид приложения, не использующий привычный нам класс TApplication.

Фундамент приложения-службы составляет специализированный класс TServiceApplication. С этим классом и процессом разработки приложений такого рода мы познакомимся несколько позже, а пока рассмотрим, каким образом класс TService реагирует на установку (и удаление) службы.

Весь перечень установленных на компьютере служб регистрируется в системном реестре Windows. Информацию об имеющихся службах мы обнаружим в ветви:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Servicies
```

С процессом регистрации службы в операционной системе связана пара событий. В момент времени, предшествующий регистрации, генерируется событие:

```
property BeforeInstall: TServiceEvent;
```

Наиболее распространенное для служб событие описывается типизированной процедурой:

```
type TServiceEvent = procedure(Sender: TService) of object;
```

Единственный параметр события Sender *– это ссылка на сервис.*

Сразу после успешной установки службы в операционной системе вызывается событие:

```
property AfterInstall: TServiceEvent;
```

Деинсталляция службы также сопровождается двумя событиями. Перед удалением службы вызывается событие:

```
property BeforeUninstall: TServiceEvent;
```

После успешного удаления возникает событие:

```
property AfterUninstall: TServiceEvent;
```

Старт и остановка службы

В роли индикатора того, что служба может быть приостановлена (значение true) с помощью метода Win32 API ControlService(), выступает свойство:

```
property AllowPause: Boolean;
```

Если свойство установлено в true, то перед приостановкой службы вызывается событие OnPause(), а после возобновления – OnContinue(). С помощью свойства AllowStop проверяется, сможем ли мы остановить сервис, воспользовавшись методом Win32 API ControlService():

```
property AllowStop: Boolean;
```

Если свойство установлено в true, перед остановкой службы вызывается событие OnStop(). Весь перечень событий, связанных с запуском и остановкой службы, представлен в табл. 21.6.

Таблица 21.6. Основные события службы

Событие	Описание
`property OnStart: TStartEvent;` `type TStartEvent = procedure(Sender: TService; var Started: Boolean) of object;`	Событием `OnStart()` сопровождается первый старт службы. В этот момент осуществляется инициализация сервиса.
`property OnPause: TPauseEvent;` `type TPauseEvent = procedure(Sender: TService; var Paused: Boolean) of object;`	Генерируется в момент приостановки службы. Управляя параметром `Paused`, программист может запретить (`false`) или разрешить (`true`) приостановку службы.
`property OnContinue: TContinueEvent;` `type TContinueEvent = procedure(Sender: TService; var Continued: Boolean) of object;`	Возобновление работы сервиса. Если мы не имеем ничего против возобновления выполнения службы, то передаем параметру `Continued` значение `true`.
`property OnStop: TStopEvent;` `type TStopEvent = procedure(Sender: TService; var Stopped: Boolean) of object;`	Событие вызывается в тот момент, когда менеджер служб завершает работу сервиса.
`property OnShutdown: TServiceEvent;`	Событие генерируется в момент завершения работы ОС. Здесь мы можем описать действия, предшествующие принудительному разрушению сервиса.

Параметры запуска службы

Два свойства `TService` обслуживают параметры запуска службы. Общее количество параметров хранится в свойстве:

```
property ParamCount: Integer;
```

Для обращения к отдельному параметру запуска передайте его индекс в строковый массив:

```
property Param[Index: Integer]: String;    //только для чтения
```

Текущее состояние службы, изменение состояния

О текущем состоянии службы нас проинформирует свойство:

```
property Status: TCurrentStatus;
type TCurrentStatus = (csStopped,          // сервис остановлен
                       csStartPending,     // сервис стартует
                       csStopPending,      // выполнение сервиса прекращается
                       csRunning,          // сервис выполняется
                       csContinuePending,  // сервис в ожидании продолжения
                       csPausePending,     // сервис в ожидании приостановки
                       csPaused);          // сервис приостановлен
```

Сразу после загрузки (рис. 21.5) служба останавливается (`csStopped`). В этом состоянии она находится до тех пор, пока не будет запущена системой в автоматическом режиме либо вручную из внешнего приложения. При поступ-

лении указания на старт служба переводится во временное состояние подготовки к старту (csStartPending). Завершив инициализацию, служба приступает к работе (csRunning).

Выполняя поставленные перед ней задачи, служба не забывает отслеживать команды, поступающие к ней от менеджера SCM. Специально для этого в состав класса инкапсулирован поток – экземпляр класса TServiceThread. В состоянии csRunning службу в первую очередь интересуют команды на остановку или временную приостановку. При поступлении этих команд служба переходит в режим полной остановки (csStopped) или паузы (csPaused).

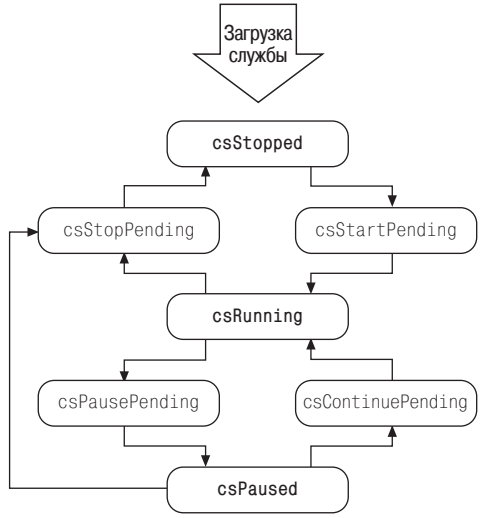

***Рис. 21.5.** Диаграмма изменений состояния службы*

Свойство Status не только информирует программиста, но и помогает ему управлять статусом службы из кода проекта. Любое изменение состояния службы согласуется с менеджером служб SCM. Для этого автоматически вызывается метод:

```
procedure ReportStatus;
```

В теле процедуры заполняется структура ServiceStatus : TServiceStatus, содержащая несколько информационных полей, таких как тип и текущее состояние службы, код ошибки и т. п. Затем вся эта информация и предложение по смене состояния службы направляются менеджеру при помощи функции Win32 API SetServiceStatus(). Если менеджер согласится с нашими предложениями, служба переводится в запрашиваемое состояние.

Не все службы поддерживают все промежуточные состояния csxxxPending. *Например, есть вероятность, что переход из режима* csRunning *в паузу* csPaused *будет осуществляться молниеносно, минуя* csPausePending. *Или в результате непредвиденного сбоя служба перейдет из любого состояния в режим полной остановки* csStopped.

Выполнение службы – поток TServiceThread

В операционной системе Windows NT каждая служба выполняется в отдельном потоке. Кроме того, для организации взаимодействия службы TService с менеджером служб для каждого экземпляра службы среда программирования Delphi создает новый поток (экземпляр класса TServiceThread). Ссылка на экземпляр потока хранится в свойстве службы:

```
property ServiceThread: TServiceThread;
```

Поток TServiceThread является прямым наследником классического потока TThread. Его основное отличие от предка состоит в наличии процедуры:

```
procedure ProcessRequests(WaitForMessage : Boolean);
```

Благодаря ProcessRequests() сервис взаимодействует с менеджером служб Windows. В рамках метода обслуживающий службу поток отслеживает управляющие сообщения, поступающие от менеджера SCM:

1. SERVICE_CONTROL_STOP – остановить сервис;
2. SERVICE_CONTROL_PAUSE – приостановить сервис;
3. SERVICE_CONTROL_CONTINUE – продолжить сервис;
4. SERVICE_CONTROL_SHUTDOWN – уничтожить сервис;
5. SERVICE_CONTROL_INTERROGATE – запросить состояние сервиса;

Таким образом, с помощью ProcessRequests() служба поддерживает постоянный контакт с менеджером служб Windows. Единственный параметр процедуры WaitForMessage переводит процедуру в синхронный (true) или асинхронный (false) режим работы. При выполнении процедуры в синхронном режиме поток ожидает управляющую команду от SCM и соответственно приостанавливает выполнение службы до поступления этой команды. В асинхронном режиме обращение к ProcessRequests() не останавливает службы.

А теперь посмотрим, каким образом служба TService сможет воспользоваться услугами этого метода. Для этого познакомимся с центральным событием, связанным с выполнением сервисом своих прямых обязанностей:

```
property OnExecute : TServiceEvent;
```

Это именно то событие, в рамках которого описывается основной исполняемый код службы. Если в функциональные обязанности службы входит постоянное обслуживание какого-то устройства, непрерывный сбор информации или что-то подобное, то внутри OnExecute() реализуют цикл, выход из которого производится по команде менеджера SCM. А эту команду проще всего получить благодаря ProcessRequests():

```
procedure TDemoService1.ServiceExecute(Sender: TService);
begin
 repeat
 //основной код службы
 ServiceThread.ProcessRequests(False);  //обращение к SCM за дальнейшими указаниями
 until (terminated=true);
end;
```

Цикл выполняется до тех пор, пока свойство Terminated не перейдет в состояние true или его не остановит менеджер служб Windows.

```
property Terminated: Boolean;
```

Свойство Terminated уведомляет программиста о том, что выполнение службы завершается и она должна быть уничтожена.

Ведение журнала событий

Любая воспитанная служба умеет общаться с журналом событий операционной системы. Благодаря записям журнала администратор компьютера получает возможность контролировать корректность работы системы и при необходимости вмешиваться в деятельность того или иного процесса или службы. Каждая строка журнала (рис. 21.6) включает информацию об имени службы, типе события, времени его возникновения и т. п.

Код ошибки, возвращаемый в момент некорректной остановки или при неудачном запуске сервиса, содержится в свойстве:

```
property ErrCode: DWord;
```

Если свойство `ErrCode` не определено, то код ошибки будет взят из свойства:

```
property Win32ErrCode: DWord;
```

И наконец, самый последний вариант развития событий, когда код ошибки не задан ни в одном из перечисленных свойств, – в свойство `Win32ErrCode` передается константа `ERROR_SERVICE_SPECIFIC_ERROR`.

О факте старта, остановки, приостановки, возобновления сервиса менеджер служб информируется автоматически без вмешательства программиста. Продолжительность отрезка времени между моментами запуска (приостановки, остановки) службы и отправки сообщения в адрес менеджера служб компьютера определяется свойством:

```
property WaitHint : Integer;       //по умолчанию 5000 миллисекунд
```

Как правило, для взаимодействия с функциями Win32 API установленного по умолчанию значения вполне достаточно. Однако если служба выполняет длительную операцию, то, вызвав `ReportStatus()`, целесообразно принудительно проинформировать менеджер о состоянии службы.

Процесс воспитания порядочной службы полностью лежит на совести создавшего ее программиста. Поэтому творец ПО обязан знать о существовании метода, отправляющего в журнал событий дополнительную информацию о жизнедеятельности сервиса:

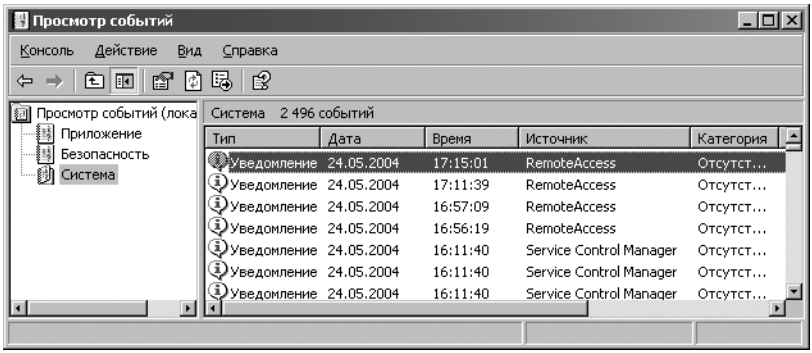

***Рис. 21.6.** Журнал событий приложений системы*

```
procedure LogMessage(Message : string; EventType : DWord = 1; Category :
                                       Integer = 0; ID : Integer = 0);
```

Здесь Message – текстовое содержание сообщения, EventType – тип события (табл. 21.7). Категория события и его идентификационный номер определяются значениями, переданными в параметры Category и ID.

Таблица 21.7. Константы типа события EventType

Константа	Описание типа события
EVENTLOG_ERROR_TYPE	Произошла ошибка.
EVENTLOG_WARNING_TYPE	Предупреждение о потенциальной опасности.
EVENTLOG_INFORMATION_TYPE	Обычное информационное уведомление.
EVENTLOG_AUDIT_SUCCESS	Аудит завершился успешно.
EVENTLOG_AUDIT_FAILURE	Ошибка аудита.

При желании скрупулезно разобраться с тонкостями ведения журнала событий стоит изучить следующую ветвь реестра:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\EventLog
```

Приложение-служба – класс TServiceApplication

Для инкапсуляции возможностей приложения-службы в визуальной библиотеке компонентов Delphi разработан класс TServiceAppliaction. Для создания нового сервисного приложения необходимо воспользоваться пунктом главного меню File – New – Other и на странице New диалогового окна New Items выбрать двойным щелчком пиктограмму Service Application.

В результате создается шаблон кода, в котором объявляется глобальный объект Application типа TServiceApplication. Кроме того, в проекте объявляется инкапсулирующий службу NT объект Service1:TService1. При необходимости можно добавить в приложение дополнительные сервисы. Для этого надо вновь открыть диалоговое окно New Items и выбрать пиктограмму Service.

Приложение TServiceAppliaction играет лишь роль оболочки для входящих в его состав сервисов и поэтому практически не обладает индивидуальными свойствами и методами. Из имеющихся в нашем распоряжении свойств интерес представляет:

```
property ServiceCount : Integer;   //только для чтения
```

информирующее программиста о количестве сервисов (экземпляров класса TService). Все объявленные в классе TServiceAppliaction виртуальные методы (CreateForm, Initialize и Run), а также конструктор и деструктор приложения предназначены для внутреннего использования и могут пригодиться лишь при разработке потомков класса.

Пример проекта службы

Попробуем создать демонстрационный проект простейшего сервиса для ОС Windows NT. Для этого откроем диалоговое окно New Items и на странице New дважды щелкнем по пиктограмме Service Application. В результате Delphi создаст шаблон проекта службы с одним сервисом.

Начнем работу с конфигурирования сервиса. Для этого выберем модуль Service1 и внесем ряд изменений в его свойства:

Тип сервиса:	`ServiceType:=stWin32;`
Интерактивность:	`Interactive:=true;`
Имя сервиса:	`Name:=DemoService;`
Название:	`DisplayName:=Демонстрация сервиса;`
Тип старта:	`StartType:=stManual;`
Реакция на ошибки при старте:	`ErrorSeverity:=esIgnory;`

Сохраните проект в отдельном каталоге, при этом модуль службы назовите DemoSrv.pas, а весь проект – dmsrv.dpr. В секции частных объявлений модуля DemoSrv.pas опишем две переменные:

```
type TDemoService = class(TService)
  private
    dc : hDC;                  //контекст устройства
    Counter : integer;         //счетчик
```

В момент старта сервиса получаем контекст дисплея и обнуляем счетчик:

```
procedure TDemoService.ServiceStart(Sender: TService; var Started: Boolean);
begin
dc:=CreateDC('DISPLAY', nil, nil, nil);
Counter:=0;
end;
```

Переходим к описанию события OnExecute() сервиса. О факте работы сервиса информируем пользователя текстовой строкой, в которой выводим текущее значение счетчика. Служба остановится в случае, когда счетчик превысит значение 100 или по команде от внешней управляющей программы. Для этого внутри цикла с помощью ProcessRequests() регулярно производим асинхронный опрос менеджера служб на предмет поступления команд от внешних приложений.

```
procedure TDemoService.ServiceExecute(Sender: TService);
var s : string;
begin
   repeat
     s:=Format('%s %d',[DemoService.DisplayName, Counter]);
     TextOut(DC, 10,10, pChar(S), Length(s));
     Sleep(180);
     INC(Counter);
```

```
      ServiceThread.ProcessRequests(False);
    until (terminated=true) or (Counter>100);
end;
```

Обращаю внимание, что вывод текстовой строки на экран возможен только в случае, когда сервис работает в интерактивном режиме. Событие остановки используем для освобождения дескриптора контекста устройства:

```
procedure TDemoService.ServiceStop(Sender: TService; var Stopped: Boolean);
begin
   MessageBeep(0);
   DeleteDC(DC);
end;
```

Проект завершен. Откомпилируйте его. Теперь необходимо зарегистрировать службу в SCM.

Регистрация службы средствами приложения

Для регистрации службы в операционной системе владеющее службой приложение должно быть запущено из командной строки с ключом /INSTALL. Например:

```
C:\DemoService\dmsrv.exe /install
```

Для снятия с регистрации применяют ключ /UNINSTALL. Процесс установки сервиса сопровождается выводом уведомляющего сообщения. Для отказа от показа окна уведомления используйте ключ /SILENT.

Откройте консоль управления службами компьютера, найдите в ней наш сервис «Демонстрация сервиса» и запустите его на выполнение...

Советы по отладке системной службы

Отладка системной службы – занятие непростое и не очень приятное по двум причинам. Во-первых, службой управляет менеджер SCM, который весьма капризен по отношению к внешним вмешательствам. Во-вторых, приложение-служба обладает как минимум двумя потоками. Если же ваше приложение-служба инкапсулирует несколько сервисов, то сложность отладки возрастает в геометрической прогрессии в соответствии с количеством интегрируемых служб.

Для организации отладки службы официальная документация Borland Delphi рекомендует сделать следующие шаги:

1. Откомпилируйте приложение-службу с включенной отладочной информацией.
2. Установите сервис в системе (для этого понадобится ключ /install).
3. Найдите службу в консоли управления и включите поддержку интерактивности (в русскоязычной версии Windows XP это называется «Разрешить взаимодействие с рабочим столом»).

4. Запустите редактор реестра Windows (рис. 21.7) и найдите ветвь *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion*. Создайте подключ Image File Execution Options, а внутри еще один ключ с названием исполняемого файла сервиса, например demo.exe. Затем для demo.exe создается новый строковый параметр с названием Debugger и в его значение передается полный путь к исполняемому файлу `delphi32.exe`.

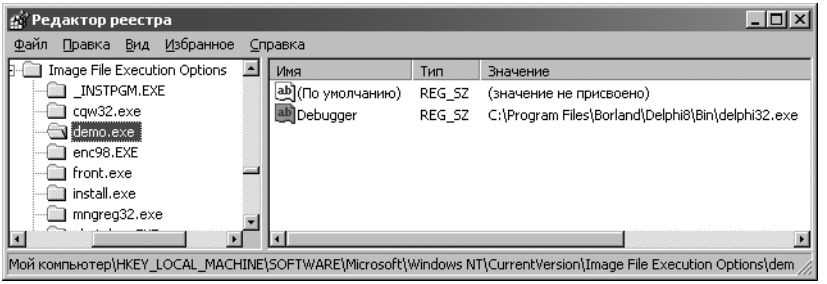

Рис. 21.7. *Подключение отладчика к сервису в редакторе реестра Windows*

5. Запустите сервис из консоли управления службами.

Какой бы странной ни показалась вам моя рекомендация по отладке служб, но реализуйте все функциональные возможности службы в обычном приложении. Выявите и устраните ошибки стандартным образом в отладчике Delphi, а затем уже перенесите апробированный код в приложение-службу.

Резюме

Системная служба – это своего рода визитная карточка современной ОС Windows 2000/XP/2003. Благодаря службам Windows решает десятки разноплановых задач, связанных с обслуживанием устройств и протекающих в системе процессов. В этой главе был рассмотрен материал, посвященный вопросам управления службами и создания собственных служб средствами Delphi.