**Сервис или служба** – это процесс, который выполняет служебные функции. Сервисы являются аналогами резидентных программ, которые использовались в операционных системах, предшествующих операционной системе Windows NT

**Резидентная программа – это специфический для старых ОС способ оставить программу в памяти для выполнения фоновых задач.**

То есть сервис — это такая программа, которая запускается при загрузке операционной системы или в процессе ее работы по специальной команде и заканчивает свою работу при завершении работы операционной системы или по специальной команде

НО! Не каждая программа, запускаемая со стартом операционной системы, является сервисом

Примеры сервисов:

* Фоновый процесс для доступа к БД (называются **серверами**).
* Программы для доступа к устройствам (**драйверами**).
* Процесс, отслеживающий работу приложения (**мониторами**).

Характеристики:

* Работают только в фоновом режиме.
* Не имеют собственного управляющего интерфейса (ни GUI, ни TUI).
* Управляются специальной программой ОС – менеджером служб (SCM).
* Запускаются/останавливаются: со стартом/выключением ОС, входом/выходом пользователя, по команде (от SCM).
* Предназначены для предоставления услуг другим программам или ОС, а не пользователям напрямую.

SCM (Service Control Manager) управляет работой сервисов. Функции SCM:

* SCM хранит информацию обо всех установленных сервисах (их имена, пути к исполняемым файлам, параметры запуска, зависимости и т.д.) в специальном разделе реестра. База данных находится под ключом **HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services**
* Запуск сервисов (при загрузке ОС или по запросу).
* Предоставляет API для получения списка установленных сервисов.
* SCM знает текущий статус каждого сервиса (остановлен, работает, запускается, останавливается и т.д.).
* Принимает команды (например, от утилиты services.msc или sc.exe) и передает их соответствующему сервису (например, "остановиться", "приостановить").
* Для обеспечения целостности при установке/удалении/конфигурировании сервисов SCM может блокировать свою базу данных.
* Внутри Services каждый установленный сервис (и драйвер) представлен отдельным подразделом (ключом реестра).
* Имя этого подраздела – это внутреннее, системное имя сервиса (Service Name), которое используется в API и командах типа sc.exe. Оно задается при создании сервиса (например, функцией CreateService). Оно может отличаться от "отображаемого имени" (Display Name), которое видят пользователи в оснастке "Службы".

# Информация о сервисе

Если что **Служебное имя (Service Name)** — это и есть **имя самого подраздела** в реестре по указанному выше пути.

**DependOnGroup/DependOnService:** Определяют зависимости. Сервис не запустится, пока не запустятся все сервисы и все сервисы из всех групп, от которых он зависит.

**Description:** Текстовое описание.

**DisplayName:** Имя, видимое пользователям. Именно DisplayName вы видите в столбце "Имя" в оснастке "Службы" (services.msc), в диспетчере задач на вкладке "Службы" и в других инструментах управления системой.

**ErrorControl**: Проблемы во время загрузки ОС. Влияет на сам процесс старта системы. Определяет реакцию. Реагирует загрузчик/ядро.

**FailureActions**: Проблемы во время нормальной работы ОС (уже после загрузки). Определяет реакцию. Влияет на доступность конкретной службы. Реагирует SCM.

**Group:** Имя группы загрузки, к которой принадлежит сервис. Используется для определения порядка запуска относительно других групп

**ImagePath:** Полный путь к исполняемому файлу сервиса (.exe или .sys), включая необходимые параметры командной строки. SCM использует это для запуска процесса сервиса. Что внутри??

**ObjectName:** Определяет учетную запись (контекст безопасности), от имени которой будет работать сервис. Влияет на доступ сервиса к ресурсам.

* LocalSystem (или не указан - часто используется по умолчанию):
  + **Описание:** Встроенная учетная запись "Система".
  + **Локальные права:** Максимальные привилегии на локальном компьютере. Имеет почти неограниченный доступ к локальным ресурсам.
  + **Сетевые права:** Представляется в сети как **учетная запись компьютера** (DOMAIN\ComputerName$). Имеет доступ к сетевым ресурсам, разрешенный для этой учетной записи компьютера в домене (если компьютер в домене).
* NT AUTHORITY\LocalService:
  + **Описание:** Встроенная учетная запись "Локальная служба".
  + **Локальные права:** Ограниченные права на локальном компьютере, схожие с правами обычного пользователя (член группы Users). Имеет минимальные привилегии, необходимые для выполнения стандартных задач служб.
  + **Сетевые права:** Представляется в сети как **анонимный пользователь**. Практически не имеет доступа к сетевым ресурсам.
* NT AUTHORITY\NetworkService:
  + **Описание:** Встроенная учетная запись "Сетевая служба".
  + **Локальные права:** Ограниченные права на локальном компьютере, схожие с LocalService (член группы Users).
  + **Сетевые права:** Представляется в сети как **учетная запись компьютера** (DOMAIN\ComputerName$), так же, как и LocalSystem.
* *Указанная учетная запись пользователя* (ИмяДомена\ИмяПользователя или .\ИмяПользователя для локальной учетной записи.

**Start:** Определяет, как сервис запускается.

* 0x0 (Boot):
  + **Описание:** Запуск загрузчиком ОС (OS Loader).
  + **Когда:** На самом раннем этапе загрузки системы, до инициализации ядра.
  + **Для чего:** Только для критически важных **драйверов** режима ядра, необходимых для доступа к загрузочному устройству (например, драйвер контроллера диска).
* 0x1 (System):
  + **Описание:** Запуск ядром на этапе инициализации ядра.
  + **Когда:** Во время инициализации ядра, после загрузки драйверов типа Boot, но до запуска SCM и большинства служб.
  + **Для чего:** Для основных **драйверов** режима ядра, не являющихся абсолютно необходимыми для загрузки, но нужных для работы базовых подсистем (например, драйверы файловой системы, Plug and Play).
* 0x2 (Automatic):
  + **Описание:** Автоматический запуск.
  + **Когда:** Запускается **Диспетчером управления службами (SCM)** после загрузки операционной системы и запуска самого SCM. Запускаются с учетом зависимостей.
  + **Для чего:** Для большинства стандартных служб, которые должны работать всегда, когда работает система (сетевые службы, планировщик, журнал событий и т.д.).
* 0x3 (Manual):
  + **Описание:** Запускается **Диспетчером Управления Службами (SCM - services.exe)**, но **только по команде**.
  + **Когда:** Служба **не запускается** автоматически при старте ОС. Она может быть запущена:
    - Вручную пользователем (через services.msc, net start, sc start).
    - Программой, которая вызывает функции API для запуска службы.
    - Другой службой, если эта служба указана как зависимость (DependOnService).
    - Системой по срабатыванию триггера (в современных версиях Windows).
  + **Для чего:** Для служб, которые нужны не постоянно, а только при определенных условиях или по требованию.
* 0x4 (Disabled):
  + **Описание:** Отключено.
  + **Когда:** Служба **не может быть запущена** ни автоматически, ни вручную (пока ее тип запуска не будет изменен на другой). SCM не позволит запустить такую службу. Запуск по триггеру также обычно не работает.
  + **Для чего:** Для отключения ненужных или проблемных служб.

**Tag:** Числовой идентификатор, уникальный в пределах группы (Group), используемый для определения порядка запуска сервисов *внутри* одной группы.

**Type:** Определяет тип сервиса.

* 0x1 (SERVICE\_KERNEL\_DRIVER) - драйвер устройства режима ядра, выполняется в пространстве ядра ОС, управляется подсистемами ядра (а не напрямую SCM). Что есть одинаковые значения?
* 0x2 (SERVICE\_FILE\_SYSTEM\_DRIVER) - драйвер файловой системы, выполняется в пространстве ядра ОС, управляется подсистемами ядра (а не напрямую SCM).
* 0x10 (SERVICE\_WIN32\_OWN\_PROCESS) – обычная служба Win32 (пользовательского режима), которая запускается **в собственном, отдельном процессе**.
* 0x20 (SERVICE\_WIN32\_SHARE\_PROCESS) – обычная служба Win32 (пользовательского режима), которая запускается **в разделяемом (общем) процессе** с другими службами.
* *Комбинированные типы:* Значения могут комбинироваться.

# Группы порядка загрузки

**Группы порядка загрузки –** логически объеденный набор сервисов, который определяет порядок загрузки сервисов входящих в него относительно остальных групп или сервисов

**Механизм определения порядка:**

* **Порядок групп (ServiceGroupOrder):** Этот ключ в **HKLM\SYSTEM\CurrentControlSet\Control** содержит список имен групп (тип REG\_MULTI\_SZ). SCM запускает сервисы группа за группой в соответствии с этим списком.
* **Порядок внутри группы (GroupOrderList):** Для каждой группы, упомянутой в ServiceGroupOrder, в этом ключе (HKLM\SYSTEM\CurrentControlSet\Control) может быть запись (тип REG\_BINARY), содержащая теги (Tag из параметров сервиса) в том порядке, в котором сервисы *внутри этой группы* должны быть запущены. Необязательно, чтобы для каждой группы была запись.
* **Явные зависимости (DependOnService, DependOnGroup):** После учета порядка групп и порядка внутри групп, SCM разрешает явные зависимости DependOnService и DependOnGroup для каждого конкретного сервиса.

**Насчёт типа файла службы**

Технически, исполняемый файл сервиса может быть скомпилирован как консольное (/SUBSYSTEM:CONSOLE) или оконное (/SUBSYSTEM:WINDOWS) приложение. SCM все равно сможет его запустить.

Сервисы могут взаимодействовать с пользователем только через рабочий стол или через окно сообщений. Это утверждение из лк Бернацкого, но:

* Исторический контекст (до Windows Vista):
  + Службы могли иметь тип SERVICE\_INTERACTIVE\_PROCESS.
  + В свойствах службы была галочка "Разрешить взаимодействие с рабочим столом" (Allow service to interact with desktop).
  + Если оба условия выполнялись, служба работала в той же сессии, что и первый вошедший пользователь (Session 0), и могла напрямую показывать окна и диалоги на его рабочем столе. Это было небезопасно (т.н. Shatter attacks).
* Современный контекст (Windows Vista и новее): Session 0 Isolation
  + Все службы принудительно запускаются в изолированной Session 0.
  + Пользователи входят в систему в других сессиях (Session 1, Session 2 и т.д.).
  + Прямое взаимодействие между службой в Session 0 и рабочим столом пользователя в другой сессии заблокировано системой по соображениям безопасности. Галочка "Взаимодействие с рабочим столом" практически не имеет эффекта.
  + Попытка службы показать обычное окно (CreateWindow) или стандартное окно сообщения (MessageBox) приведет к тому, что оно либо не появится вообще, либо появится невидимо в Session 0.
* Как службы взаимодействуют с пользователем СЕЙЧАС?
  + Косвенно, через IPC: Самый распространенный и правильный способ. Служба (работающая в фоне) общается с отдельным приложением пользовательского режима (например, значком в трее, панелью управления, агентом), которое запущено в сессии пользователя. Обмен данными идет через механизмы межпроцессного взаимодействия (IPC): именованные каналы (Named Pipes), RPC, сокеты, WCF и т.д. Именно пользовательское приложение показывает интерфейс (окна, уведомления) на основе данных от службы.
  + Специализированные функции: Существует функция WTSSendMessage, которая позволяет процессу (включая службу) отправить простое окно сообщения (message box) в конкретную сессию пользователя. Это исключение, предназначенное для важных уведомлений, но это не полноценное взаимодействие с рабочим столом.

**Структура проекта**

Функция обратного вызова – это функция, которая передаётся другой функции в качестве аргумента.

**Функции обратного вызова (Callbacks):** Это функции, написанные разработчиком сервиса, но вызываемые не им напрямую, а системой (в данном случае, SCM или его потоком-диспетчером) в определенные моменты времени. Разработчик передает указатели на эти функции системе через API.

Сервис должен содержать две функции обратного вызова:

* **Точка входа сервиса (Service Main Function):** Функция, которая будет вызвана SCM, когда сервис должен начать свою работу. Здесь выполняется инициализация и основная логика сервиса.
* **Обработчик управляющих команд (Handler Function):** Функция, которая будет вызвана SCM (через диспетчера), когда сервису приходит команда (например, остановиться, приостановить, запросить статус).

int main(int argc, char \*argv) { ... }

// точка входа сервиса

VOID WINAPI ServiceMain (DWORD dwArgc, LPTSTR \*lpszArgv) { ... }

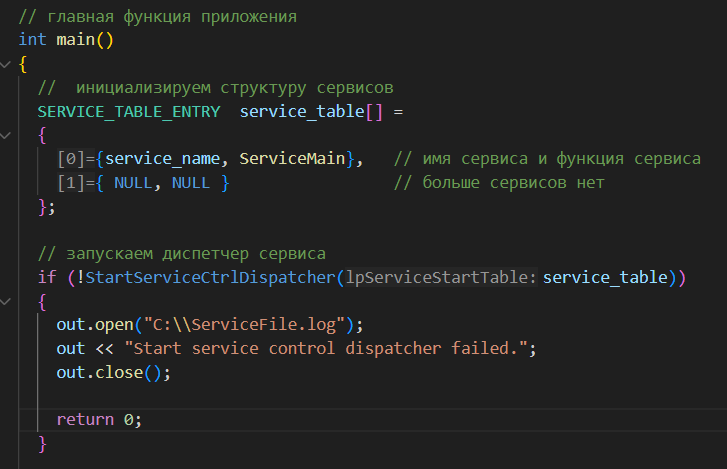
// обработчик запросов

VOID WINAPI ServiceCtrlHandler (DWORD dwControl) { ... }

* **ServiceMain:** Функция, которую вызовет SCM после успешной регистрации через main. Здесь начинается реальная работа сервиса. Она должна зарегистрировать обработчик команд (ServiceCtrlHandler).
* **ServiceCtrlHandler:** Функция, которую будет вызывать диспетчер SCM при получении команд для этого сервиса. Она обрабатывает такие запросы, как остановка, пауза и т.д.
* WINAPI: Соглашение о вызовах, стандартное для функций Windows API (определяет, как параметры передаются в функцию и кто очищает стек). VOID - функция не возвращает значения. DWORD, LPTSTR - стандартные типы данных Windows API.

**Роль main**

Главной задачей функции main является запуск диспетчера сервиса, который является потоком и управляет этим сервисом (с помощью **StartServiceCtrlDispatcher**)



SERVICE\_TABLE\_ENTRY: Это структура, которая связывает системное имя сервиса (строка) с его функцией точки входа (ServiceMain).

Зачем тут писать имя сервиса? Чтобы можно было exe использовать для разных сервисов?

Массив service\_table: Создается массив этих структур. Каждая строка описывает один сервис, который может быть запущен из этого EXE. Последняя запись в массиве должна быть нулевой ({NULL, NULL}), чтобы обозначить конец списка. В примере показан случай с одним сервисом (service\_name - это строка с системным именем, ServiceMain - указатель на функцию).

StartServiceCtrlDispatcher(service\_table): Эта функция вызывается из main. Она передает таблицу сервисов SCM'у. Функция блокирует поток main до тех пор, пока все сервисы, запущенные из этого процесса, не остановятся. SCM, получив таблицу, смотрит, какой сервис нужно запустить (по имени, переданному при старте сервиса), находит соответствующую ServiceMain в таблице и вызывает ее в новом потоке (том самом диспетчере).

1. У нас есть *определение* сервиса в реестре. В этом определении, в поле ImagePath, указан путь к вашему .exe файлу.
2. Когда SCM решает запустить сервис, он **запускает указанный .exe как процесс**. На этом этапе процесс существует, но SCM **НЕ считает, что сервис уже РАБОТАЕТ**. Он считает, что *процесс сервиса запущен* и находится в стадии инициализации.
3. Внутри запущенного процесса выполняется функция main. Ее задача — вызвать StartServiceCtrlDispatcher. Эта функция не определяет точку входа сама по себе, а передает SCM'у таблицу, в которой ваш код уже указал имя сервиса и соответствующую ему функцию ServiceMain (точку входа для логики сервиса). SCM, получив эту таблицу, узнает, какую функцию вызвать дальше.
4. Внутри запущенного процесса выполняется функция main. Ее задача — вызвать StartServiceCtrlDispatcher. Эта функция не определяет точку входа сама по себе, а передает SCM'у таблицу, в которой ваш код уже указал имя сервиса и соответствующую ему функцию ServiceMain (точку входа для логики сервиса). SCM, получив эту таблицу, узнает, какую функцию вызвать дальше.
5. SCM (или система по его указанию) создает новый поток: Этот новый поток создается внутри вашего процесса.
6. Новый поток выполняет ServiceMain: Именно этот новый поток начинает выполнение с вашей функции ServiceMain (той, что была указана в таблице для запущенного сервиса).
7. Роль диспетчера: Этот новый поток, выполняющий ServiceMain, также берет на себя роль диспетчера команд для этого сервиса. Внутри ServiceMain вы вызываете RegisterServiceCtrlHandler, которая сообщает системе, какую функцию (ServiceCtrlHandler) нужно вызывать в этом же потоке, когда SCM пришлет команду.

* Из-за этого таймаута, *длительную* инициализацию нельзя проводить в main *до* вызова StartServiceCtrlDispatcher. Всю основную инициализацию, специфичную для сервиса (открытие файлов, подключение к БД, создание рабочих потоков и т.д.), следует выполнять *внутри* ServiceMain, *после* того, как связь с SCM установлена (т.е., после успешного вызова RegisterServiceCtrlHandler).
* Если нужна какая-то быстрая инициализация глобальных данных до вызова StartServiceCtrlDispatcher, это можно сделать в main. Если же она долгая, но должна произойти до старта основного цикла сервиса, можно запустить ее в отдельном потоке из main, а main сразу вызовет StartServiceCtrlDispatcher. ServiceMain затем дождется завершения этого потока инициализации перед тем, как перейти к основной работе.
* Есть 30 секунд на вызов StartServiceCtrlDispatcher.

**Роль ServiceMain**

Функция, определяющая точку входа сервиса, должна иметь следующий прототип:

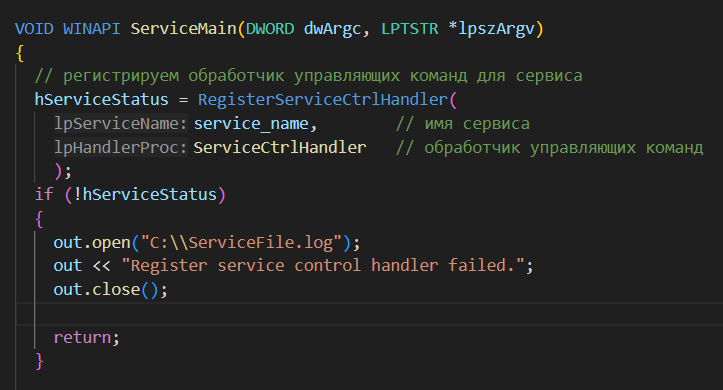
VOID WINAPI имя\_точки\_входа(DWORD dwArgc, LPTSTR \*lpszArgv);

Имя: Обычно ServiceMain, но может быть любым, если оно правильно указано в SERVICE\_TABLE\_ENTRY. Если в EXE несколько сервисов, у каждого должна быть своя (или одна общая, разбирающаяся по имени) функция ServiceMain.

dwArgc: Количество аргументов командной строки, переданных сервису при запуске (например, через sc start MyService param1 param2).

lpszArgv: Массив указателей на строки аргументов командной строки.

Аналогия с main: Параметры dwArgc и lpszArgv аналогичны argc и argv в стандартной функции main, позволяя передавать сервису параметры при запуске.



RegisterServiceCtrlHandler(lpServiceName, lpHandlerProc): Эта функция должна быть вызвана как можно раньше внутри ServiceMain

* lpServiceName: Системное имя сервиса (та же строка, что и в SERVICE\_TABLE\_ENTRY). Это поле сейчас игнорируется и может быть NULL, если тип сервиса SERVICE\_WIN32\_OWN\_PROCESS. Зачем указывать имя сервиса?
* lpHandlerProc: Указатель на функцию-обработчик команд (в примере ServiceCtrlHandler).
* **Возвращаемое значение (hServiceStatus):** Функция возвращает дескриптор состояния сервиса (SERVICE\_STATUS\_HANDLE). Этот дескриптор *необходимо* сохранить и использовать в последующих вызовах SetServiceStatus для обновления статуса сервиса.

Представляет обязательную последовательность шагов внутри ServiceMain для корректной работы сервиса и взаимодействия с SCM.

1. Регистрация обработчика: Сразу вызвать RegisterServiceCtrlHandler, чтобы сервис мог получать команды (особенно команду остановки, если инициализация затянется).
2. **Состояние START\_PENDING**: Немедленно после регистрации обработчика сообщить SCM, что сервис начал запускаться, вызвав SetServiceStatus с состоянием SERVICE\_START\_PENDING. Здесь же нужно установить dwWaitHint и dwCheckPoint (см. ниже), если инициализация долгая.
3. Инициализация: Выполнить все необходимые действия для подготовки сервиса к работе (загрузка конфига, открытие сокетов, запуск рабочих потоков и т.д.). Во время долгой инициализации нужно периодически вызывать SetServiceStatus с обновленным dwCheckPoint, чтобы SCM знал, что процесс не завис.
4. **Состояние RUNNING**: После успешной инициализации сообщить SCM, что сервис готов к работе, вызвав SetServiceStatus с состоянием SERVICE\_RUNNING. dwCheckPoint и dwWaitHint сбрасываются в 0.
5. Основной цикл работы: Выполнять основную задачу сервиса. Этот цикл должен периодически проверять, не пришла ли команда на остановку или паузу (обычно это делается через глобальный флаг, который устанавливается в ServiceCtrlHandler).
6. Завершение: Когда получена команда остановки (SERVICE\_CONTROL\_STOP или SERVICE\_CONTROL\_SHUTDOWN), ServiceCtrlHandler должен установить флаг завершения. Основной цикл ServiceMain должен увидеть этот флаг, выполнить необходимые действия по очистке (закрыть файлы, освободить ресурсы), сообщить SCM о состоянии **SERVICE\_STOP\_PENDING**, затем **SERVICE\_STOPPED**, и только после этого функция ServiceMain должна завершиться.

**Роль диспетчера сервиса**

Диспетчер сервиса – это поток, который управляет сервисом.

Диспетчер сервиса запускается при помощи StartServiceCtrlDispatcher.

Диспетчер сервиса получает управляющие сигналы от менеджера сервисов по именованному каналу и передает эти запросы функции **ServiceCtrlHandler**, которая обрабатывает эти управляющие запросы

Если в приложении несколько сервисов, то для каждого сервиса запускается свой диспетчер и для каждого диспетчера определяется своя функция обработки управляющих запросов, которая выполняется в контексте соответствующего диспетчера сервисов

**Роль ServiceCtrlHandler**

Выполняется в контексте диспетчера сервиса.

Функция, определяющая обработчик управляющих запросов, должна иметь следующий прототип:

VOID WINAPI имя\_обработчика\_запросов (DWORD dwControl);

Если определяется только один сервис, то эта функция обычно называется ServiceCtrlHandler. Если же в приложении определяется несколько сервисов, то естественно, что обработчик запросов для каждого сервиса должен иметь свое имя.

Эта функция содержит только один параметр, который содержит **код управляющего сигнала**. Так как обработчик запросов вызывается диспетчером сервиса, то и коды управляющих сигналов он получает от своего диспетчера

Возможны следующие основные управляющие коды:

**>SERVICE\_CONTROL\_STOP – остановить сервис**

**>SERVICE\_CONTROL\_PAUSE – приостановить сервис**

**>SERVICE\_CONTROL\_CONTINUE – возобновить сервис**

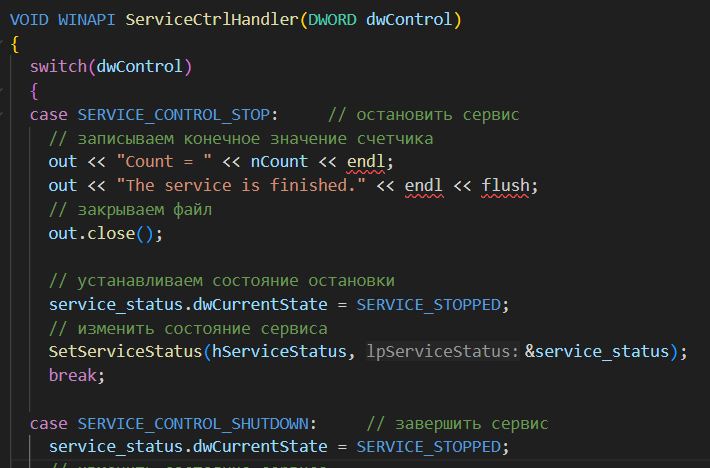
**>SERVICE\_CONTROL\_INTERROGATE – обновить состояние сервиса. Сервис должен немедленно вызвать SetServiceStatus со своим актуальным состоянием.**

**>SERVICE\_CONTROL\_SHUTDOWN – закончить работу сервиса**

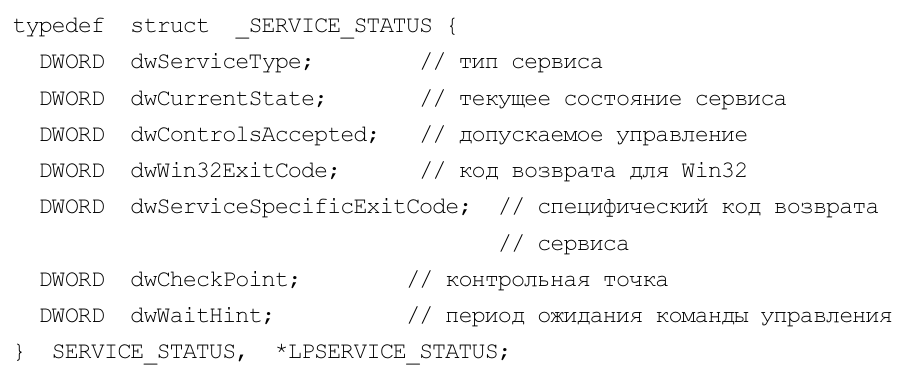
* Обработчик должен игнорировать (или возвращать ошибку) команды, для которых соответствующий флаг не был установлен в dwControlsAccepted при последнем вызове SetServiceStatus. Исключение - INTERROGATE.

**Пользовательские коды:** Сервис может обрабатывать кастомные команды (**128-255**), отправленные ему через ControlService функцией другого приложения. Это используется для специфичного для приложения управления.

* + В нормальной ситуации состояние сервиса должно изменится в течении 30 секунд.
  + Обработчик не должен блокироваться надолго. Если обработка команды (например, остановка) требует много времени (больше 30 секунд), ServiceCtrlHandler должен:
    1. Сообщить SCM о начале операции (SERVICE\_STOP\_PENDING с dwWaitHint и dwCheckPoint=1).
    2. Запустить рабочий поток для выполнения длительной операции.
    3. Быстро завершить работу ServiceCtrlHandler.
    4. Рабочий поток должен периодически обновлять dwCheckPoint через SetServiceStatus и в конце установить финальное состояние (SERVICE\_STOPPED).
  + **Таймаут SHUTDOWN (20 сек):** При получении SERVICE\_CONTROL\_SHUTDOWN у сервиса есть ограниченное время (по умолчанию около 20 секунд, настраивается в реестре WaitToKillServiceTimeout) на завершение. Если он не успевает, система может принудительно завершить его процесс. Поэтому обработка SHUTDOWN должна быть максимально быстрой.



Параметр lpServiceStatus функции **SetServiceStatus** должен указывать на структуру типа **SERVICE\_STATUS**, которая содержит информацию о состоянии сервиса. Эта структура имеет следующий формат:



Поле **dwServiceType** содержит тип сервиса и может принимать следующие значения:

1. Определяет, будет ли сервис работать в своем собственном процессе или делить один процесс с другими сервисами: **SERVICE\_WIN32\_OWN\_PROCESS** – сервис является самостоятельным процессом, **SERVICE\_WIN32\_SHARE\_PROCESS** – сервис разделяет процесс с другими сервисами (от имени **системной учетной записи LocalSystem, NetworkService, LocalService)**

2. Типы для драйверов режима ядра. **SERVICE\_KERNEL\_DRIVER** – сервис является драйвером устройства, **SERVICE\_FILE\_SYSTEM\_DRIVER** – сервис является драйвером файловой системы

3. Типы для сервисов, работающих в контексте вошедшего пользователя: **SERVICE\_USER\_OWN\_PROCESS** – сервис является самостоятельным процессом, запускаемым для определенного пользователя, **SERVICE\_USER\_SHARE\_PROCESS** – сервис разделяет процесс с другими сервисами запускаемыми для определенного пользователя

Кроме того, первые два флага могут быть установлены совместно с флагом **SERVICE\_INTERACTIVE\_PROCESS** – сервис может взаимодействовать с рабочим столом

Поле **dwCurrentState** содержит текущее состояние сервиса. Это поле может принимать одно из следующих значений:

>SERVICE\_STOPPED – сервис остановлен

>SERVICE\_START\_PENDING – сервис стартует

>SERVICE\_STOP\_PENDING – сервис останавливается

>SERVICE\_RUNNING – сервис работает

>SERVICE\_CONTINUE\_PENDING – сервис переходит в рабочее состояние

>SERVICE\_PAUSE\_PENDING – сервис переходит в состояние ожидания

>SERVICE\_PAUSED – сервис находится в состоянии ожидания

START\_PENDING, STOP\_PENDING, CONTINUE\_PENDING, PAUSE\_PENDING: Промежуточные (pending) состояния, указывающие, что сервис выполняет переход между стабильными состояниями. Во время этих состояний SCM ожидает периодического обновления **dwCheckPoint** или перехода в стабильное состояние в течение **dwWaitHint**.

* **dwWaitHint (Подсказка об ожидании):**
  + Используется, когда сервис в *промежуточном* состоянии (START\_PENDING, STOP\_PENDING и т.д.).
  + Сообщает SCM: "Я ожидаю, что следующий шаг в моей долгой операции (или её завершение) займет **примерно столько-то миллисекунд**".
  + SCM использует это как **таймаут**: если за это время ничего не изменилось (не увеличился dwCheckPoint или не сменилось состояние), SCM может посчитать, что сервис завис.
* **dwCheckPoint (Контрольная точка):**
  + Используется *вместе* с dwWaitHint во время долгих переходов между состояниями.
  + Это **счетчик прогресса**. Сервис должен его **увеличивать** (обычно на 1) и вызывать SetServiceStatus каждый раз, когда выполнен значимый шаг в долгой операции.
  + Показывает SCM: "Я не завис, я продвинулся в выполнении операции до точки № dwCheckPoint".
  + Если переход быстрый (быстрее dwWaitHint), dwCheckPoint можно не использовать (оставить 0 или 1).

Перечисляет флаги, которые сервис устанавливает в поле **dwControlsAccepted**, чтобы сообщить SCM, какие команды он способен обрабатывать в своей функции ServiceCtrlHandler

>SERVICE\_ACCEPT\_STOP – можно остановить сервис

>SERVICE\_ACCEPT\_PAUSE\_CONTINUE – можно приостановить и возобновить сервис

>SERVICE\_ACCEPT\_SHUTDOWN – Сервис хочет получать уведомление SERVICE\_CONTROL\_SHUTDOWN при выключении системы

>SERVICE\_ACCEPT\_PRESHUTDOWN – сервис может выполнить действия перед выключением системы ?

>SERVICE\_ACCEPT\_PARAMCHANGE - сервис может вновь прочитать свои стартовые параметры без перезагрузки

>SERVICE\_ACCEPT\_NETBINDCHANGE - сервис является сетевой компонентой

**dwServiceSpecificExitCode:** позволяет сервису вернуть свой собственный код ошибки, если стандартных кодов Win32 недостаточно. Для этого в **dwWin32ExitCode** нужно установить ERROR\_SERVICE\_SPECIFIC\_ERROR, а сюда записать специфический код.

**dwCheckPoint:** используется *только* когда сервис находится в одном из pending-состояний (START\_PENDING, STOP\_PENDING, PAUSE\_PENDING, CONTINUE\_PENDING). Сервис должен инкрементировать это значение (начиная с 1) и вызывать SetServiceStatus каждый раз, чтобы показать SCM, что он прогрессирует и не завис. Если переход быстрый (меньше значения dwWaitHint и меньше таймаута SCM), dwCheckPoint можно оставить равным 0.

**dwWaitHint:** используется *только* когда сервис находится в pending-состоянии. Это значение (в миллисекундах) сообщает SCM, сколько *примерно* времени сервису потребуется до следующего шага – либо увеличения dwCheckPoint, либо перехода в финальное стабильное состояние. SCM использует это значение как таймаут. Если сервис находится в pending-состоянии, и с момента последнего вызова SetServiceStatus прошло больше времени, чем dwWaitHint, *и при этом* dwCheckPoint не увеличился, SCM может посчитать, что сервис завис, и предпринять действия (например, записать ошибку в журнал или даже завершить процесс сервиса).

**Смысл переходных состояний**

Когда SCM получает команду на изменение состояния службы (например, на её запуск или остановку), оно ожидает, что служба перейдет в новое состояние. Однако процессы, такие как инициализация или освобождение ресурсов, могут занять некоторое время. В этом случае, чтобы избежать преждевременного завершения операции или неполного выполнения, служба может вернуться в состояние "Pending", которое говорит SCM: "Я все еще работаю над операцией, дайте мне немного времени".

* Прежде чем SCM сможет управлять сервисом, информация о нем должна быть добавлена в базу данных SCM (в реестр). Это делается либо специальным инсталлятором, либо вручную командой sc create, либо программно через CreateService.
* Запуск: Способы запуска уже обсуждались (авто, вручную, по требованию, программно через StartService).
* Удаление (Uninstallation): когда сервис больше не нужен, его следует удалить из базы данных SCM (командой sc delete или функцией DeleteService), чтобы система о нем "забыла".

**WinAPI**

Перечисляет **основные функции WinAPI для программной работы с сервисами из *другого* приложения** (например, инсталлятора или управляющей утилиты).

* + **OpenSCManager:** Первый шаг. Получает дескриптор SCM на локальной или удаленной машине, необходимый для вызова большинства других функций. Требует определенных прав доступа.
  + **CreateService:** Регистрирует новый сервис в базе данных SCM, указывая все его параметры (имя, путь, тип, старт, зависимости и т.д.).
  + **OpenService:** Получает дескриптор *конкретного*, уже установленного сервиса, необходимый для управления им (Start, Control, Delete, Query). Требует прав доступа к сервису.
  + **StartService:** Используя дескриптор, полученный от OpenService, **дает команду SCM запустить** этот сервис (SCM найдет его .exe и запустит процесс, который затем вызовет StartServiceCtrlDispatcher и т.д.).
  + **ControlService:** Используя дескриптор сервиса, **отправляет управляющий код** (STOP, PAUSE, пользовательский код и т.д.) работающему сервису. SCM передаст этот код потоку-диспетчеру сервиса, который вызовет ваш ServiceCtrlHandler.
  + **DeleteService:** Помечает сервис для удаления. Фактическое удаление из реестра произойдет после закрытия всех дескрипторов этого сервиса и его остановки.
  + **QueryServiceStatusEx:** Запрашивает текущий статус сервиса (структуру SERVICE\_STATUS\_PROCESS, которая включает PID и другую информацию).
  + **Библиотека:** Для использования этих функций при компиляции нужно линковаться с библиотекой **Advapi32.lib** (или указать -ladvapi32 для MinGW/GCC). Сами функции находятся в Advapi32.dll.

В linux: **inih — это простая, легковесная библиотека на языке C для парсинга (чтения и разбора) конфигурационных файлов в формате .ini.**

Представляет **sc.exe - мощную утилиту командной строки** для управления сервисами, являющуюся оберткой над функциями WinAPI Service.

* + **sc create <имя\_сервиса> [параметры...]:** Аналог CreateService. Позволяет задать binPath=, type=, start=, depend=, obj=, password= и другие параметры.
  + **sc start <имя\_сервиса> [аргументы...]:** Аналог StartService.
  + **sc control <имя\_сервиса> <код>:** Аналог ControlService. код - числовое значение команды (например, 1 для stop, 2 для pause, 3 для continue). Можно использовать и пользовательские коды.
  + **sc stop <имя\_сервиса>:** Удобная обертка над sc control <имя> 1.
  + **sc delete <имя\_сервиса>:** Аналог DeleteService.
  + **sc query [имя\_сервиса] [параметры...]:** Аналог QueryServiceStatusEx (и других функций для получения информации). Позволяет запросить статус конкретного сервиса или список сервисов по разным критериям (тип, состояние, группа).
  + **Другие команды:** sc config (изменение конфигурации), sc qc (запрос конфигурации), sc qdescription (запрос описания), sc failure (настройка действий при сбое) и т.д.

**Linux-демон**

Демон (англ. daemon) – это процесс, обладающий следующими свойствами:

> Часто демоны создаются во время загрузки системы и работают до момента ее выключения.

>Выполняется в фоновом режиме

>Не имеет контролирующего терминала.

>Названия демонов принято заканчивать буквой d (хотя это не является обязательным правилом)

**Для того чтобы стать демоном, программа должна выполнить следующие шаги:**

1. **Вызов fork**. Потомок (наш демон) продолжает работать, а родитель завершается.

* Оболочка запустила программу, она что-то там поделала (создала демона) и завершилась. **Для оболочки команда завершилась**, мы опять увидим приглашение на ввод следующей команды, а демон продолжает работать в фоновом режиме.
* Системный вызов setsid() (следующий шаг) может быть успешно вызван только процессом, который *не* является лидером группы процессов (Process Group Leader). После fork(), потомок наследует идентификатор группы процессов (PGID) от родителя, но получает свой уникальный идентификатор процесса (PID). Так как PID != PGID, **потомок гарантированно не является лидером группы**, что позволяет ему успешно вызвать setsid().
* Когда родитель завершается, осиротевший **потомок "усыновляется" процессом init (systemd)**. Это важно для корректной обработки статуса завершения демона системой.

2. Дочерний процесс (демон) вызывает **setsid**, чтобы начать новую сессию и разорвать любые связи с терминалом. **Эффекты:**

* + - Процесс становится **лидером новой сессии** (Session Leader).
    - Процесс становится **лидером новой группы процессов** (Process Group Leader).
    - Процесс **отсоединяется от терминала** (если он был).

3. **Проблема:** Даже после setsid есть маааленький шанс, что если демон случайно откроет файл терминала (например, /dev/tty1), то этот терминал снова станет для него "контролирующим". Демону это не нужно.

* Делаем еще один fork! Тот процесс, который вызвал setsid (назовем его "средний"), создает **еще одну копию ("внука")** и тут же завершается. Работать продолжает "внук".
* "Средний" процесс был лидером сессии (после setsid). А "внук" уже *не* лидер сессии. Только лидер сессии может случайно "схватить" терминал. Поскольку наш "внук" не лидер, он теперь в полной безопасности от случайного подключения к терминалу.

4. Вызываем **umask(0).**

* Обычно система немного "обрезает" права доступа, когда программа создает файлы (это настройка umask, унаследованная от родителя). Демону это не нужно, он хочет сам точно указывать, какие права будут у его лог-файлов, конфигов и т.д. umask(0) говорит: "Не обрезай ничего, я сам разберусь".

5. Меняем **текущий рабочий каталог на корневой (/).**

* Представь, ты запустил демона из папки на флешке. Пока демон работает, система не даст тебе отмонтировать эту флешку ("устройство занято"). **Чтобы демон не "держал" никакой каталог** (особенно на съемных носителях), его переводят в корень (/) или другую папку, которая точно всегда будет доступна и не будет отмонтирована.

6. Закрыть все открытые файловые дескрипторы, которые демон унаследовал от своего родителя.

7. **Стандартный ввод** (stdin, дескриптор 0), **стандартный вывод** (stdout, дескриптор 1) и **стандартный вывод ошибок** (stderr, дескриптор 2) перенаправляем в специальный файл **/dev/null**. Это как "черная дыра": всё, что туда пишешь – исчезает, при попытке чтения – сразу конец файла.

**Рекомендации**

Обычно демон останавливается при выключении системы. Получает сигнал **SIGTERM** от init, дающий ему шанс корректно завершиться (освободить ресурсы, сохранить данные). Делать это нужно **быстро (5 секунд)**, так как вскоре после SIGTERM может прийти **SIGKILL**, который убьет процесс принудительно.

Так как работает долго, нужно следить за утечками памяти и незакрытыми файлами (файловыми дескрипторами). Иначе ресурсы системы могут исчерпаться, и демон придется перезапускать.

Часто нужно, чтобы работал только один экземпляр демона. Это достигается с помощью "**файла блокировки**": первый запущенный демон создает и блокирует специальный файл. Если попытаться запустить второй экземпляр, он увидит, что файл заблокирован, поймет, что демон уже работает, и сам завершится.

**Менеджеры сервисов**

Для работы с демонами в Linux также как и в Windows существует менеджер сервисов:

* **init (SysVinit):** Традиционная система инициализации в Linux/Unix. Считается устаревшей во многих дистрибутивах.
* **systemd:** Современная система инициализации и управления сервисами. Является стандартом де-факто во многих популярных дистрибутивах (Ubuntu, Fedora, CentOS, Debian new).

При использовании init или systemd для запуска вашего приложения как сервиса, вам *не нужно* вручную выполнять шаги демонизации (fork, setsid, chdir, close FDs, redirect stdio) в коде вашего приложения. **Система инициализации сделает это за вас!** Ваш код должен просто выполнять основную работу, как обычное приложение (возможно, с чтением конфига и логированием). Менеджер сервисов сам запустит его в фоне, отсоединит от терминала и т.д.

**Менеджер сервисов init**

При использовании менеджера init:

1. **/etc/<имя демона>**: Здесь должны лежать **конфигурационные файлы** твоего демона.
2. **/sbin/<имя демона>**: Сюда предлагается помещать **исполняемый файл** самого демона.
3. **/etc/init.d/<имя демона>**: Это **ключевое место** для init системы. Сюда кладется **скрипт управления** твоим демоном. Именно этот скрипт будет вызываться системой (или тобой через команду service) с разными аргументами (start, stop и т.д.), чтобы запустить, остановить или проверить статус демона.
4. **/var/log**: Стандартное место для **файлов журналов (логов)**.

Первый и последний пункты не являются требованиями, а скорее общепринятыми практиками при создании системных сервисов

Стандартные команды, которые должен понимать твой управляющий скрипт в /etc/init.d/<имя демона>:

* **start**: Запустить демон.
* **stop**: Остановить демон.
* **restart**: Перезапустить демон.
* **reload**: Сказать *уже работающему* демону перечитать свои конфигурационные файлы *без остановки*.
* **force-reload**: Попытаться выполнить reload. Если демон не поддерживает reload (или скрипт не знает, как это сделать), то просто выполнить restart.
* **status**: Показать текущее состояние демона.

**service**: Это удобная команда для пользователя/администратора. Вместо того чтобы писать полный путь /etc/init.d/<имя демона> start, ты можешь просто написать service <имя демона> start. Эта утилита сама найдет нужный скрипт в /etc/init.d и выполнит его с указанной командой (start, stop и т.д.).

**systemctl**: Это аналог команды service, но для более современной системы инициализации **systemd**, которая сейчас используется в большинстве дистрибутивов Linux вместо старой init.

Пошаговый пример добавления сервиса с использованием SysVinit.

**Шаг 1:** Скомпилировать исходный код в исполняемый файл.

**Шаг 2:** Скопировать файл в системный каталог для исполняемых файлов демонов /sbin/.

**Шаг 3:** Создать init-скрипт (shell-скрипт, обрабатывающий start, stop и т.д.) и поместить его в /etc/init.d/ под именем демона.

**Шаг 4 (Опционально):** Если у демона есть конфигурационные файлы, создать для них каталог /etc/.

**Шаг 5 (Опционально):** Логи обычно пишутся в /var/log/.

**Шаг 6:** Выполнить команду **service --status-all**. Эта команда перебирает все скрипты в /etc/init.d/ и пытается выполнить для каждого команду status.В списке должен появиться новый сервис.

**Шаг 7:** Использовать команду service с разными действиями.

**Шаг 8:** Для автозапуска вместе с системой надо перейти в каталог **/etc/init.d** и выполнить следующую команду: **sudo update-rc.d имя defaults**

**Менеджер сервисов systemd**

**Шаг 1:** Скомпилировать исходный код в исполняемый файл.

**Шаг 2:** Скопировать файл в системный каталог для исполняемых файлов демонов /sbin/.

**Шаг 3:** Вместо init-скрипта создается **unit-файл** с расширением **.service**. Этот файл имеет текстовый формат INI-стиля и описывает, как systemd должен управлять сервисом (путь к исполняемому файлу, пользователь, зависимости, параметры запуска и т.д.). Файл помещается в каталог **/etc/systemd/system/.**

**Шаг 4 (Опционально):** Если у демона есть конфигурационные файлы, создать для них каталог /etc/.

**Шаг 5 (Опционально):** Логи обычно пишутся в /var/log/.

**Шаг 6:** Использовать команду systemctl list-unit-files. Команда показывает список всех unit-файлов, известных systemd, и их статус (enabled, disabled, static, masked).

**Шаг 7:** Использовать команду systemctl с разными действиями.

**Шаг 8:** Использовать команду **systemctl enable <имя\_сервиса>.service** для автозапуска.

[Unit]  
Description=DemoService Daemon  
After=local-fs.target

[Service]  
Type=simple  
ExecStart=/usr/local/bin/demoserviced  
ExecReload=/bin/kill -HUP $MAINPID  
KillSignal=SIGTERM

[Install]  
WantedBy=multi-user.target

**[Unit]**: Эта секция содержит общие метаданные о нашем сервисе (юните) и определяет его зависимости и порядок запуска относительно других юнитов в системе.

**Description**: Человекочитаемое описание сервиса.

**After=local-fs.target**: Директива упорядочивания запуска. Она говорит systemd: "Запускай этот сервис только *после* того, как будет достигнуто состояние (таргет) local-fs.target". **Что такое local-fs.target?** Это стандартный таргет systemd, который активируется, когда все *локальные* файловые системы успешно смонтированы и готовы к использованию.

**[Service]:** Эта секция содержит конфигурацию, специфичную для юнитов типа "сервис". Она определяет, как systemd должен запускать и управлять процессом этого сервиса.

**Type=simple**: Указывает тип запуска сервиса. Это **очень важная** директива в твоем случае. simple означает, что основной процесс сервиса — это тот самый процесс, который запускается командой ExecStart. systemd считает сервис успешно запущенным сразу после того, как ExecStart команда будет выполнена (без ошибок). systemd **не ожидает**, что этот процесс сделает fork().

**ExecStart=/usr/local/bin/demoserviced**: Полный путь и команда для запуска исполняемого файла твоего сервиса.

**ExecReload=/bin/kill -HUP $MAINPID**: Команда, которую systemd выполнит, когда ты попросишь перезагрузить конфигурацию сервиса командой sudo systemctl reload demoserviced.

* **/bin/kill**: Это стандартная Linux-утилита для отправки сигналов процессам.
* HUP — это сокращение от SIGHUP

**KillSignal=SIGTERM**: Сигнал, который systemd будет использовать по умолчанию, чтобы попросить твой сервис завершить работу, когда ты выполняешь sudo systemctl stop demoserviced.

**[Install]**: Эта секция не влияет на то, как сервис работает, но определяет, как он интегрируется в систему при использовании команд systemctl enable (включить автозапуск) и systemctl disable (выключить автозапуск).

* **Что такое multi-user.target?** Это стандартный таргет systemd, представляющий собой обычный многопользовательский режим работы системы без графического интерфейса (типичный режим для серверов).

**sigaction**— это системный вызов в Linux (и других POSIX-совместимых системах), который используется для изучения и изменения *действия*, выполняемого процессом при получении определенного *сигнала*.

Функция sigaction имеет следующий прототип:

int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact);

* signum: Номер сигнала, для которого мы устанавливаем или получаем действие (например, SIGINT, SIGTERM, SIGUSR1). Нельзя перехватить или игнорировать SIGKILL и SIGSTOP.
* act: Указатель на структуру struct sigaction, которая описывает *новое* действие для сигнала signum. Если передать NULL, действие не меняется (можно использовать для получения текущих настроек).
* oldact: Указатель на структуру struct sigaction, куда будет записано *предыдущее* (старое) действие для сигнала signum. Если передать NULL, старое действие не сохраняется.

struct sigaction {

void (\*sa\_handler)(int); // Простой обработчик (как в signal())

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*); // Расширенный обработчик

sigset\_t sa\_mask; // Сигналы, блокируемые во время выполнения обработчика

int sa\_flags; // Флаги, управляющие поведением

void (\*sa\_restorer)(void); // Не используется приложениями (устарело/внутреннее)

};

В нашем коде:

Цель использования sigaction — настроить реакцию демона на два конкретных сигнала:

1. SIGTERM: Стандартный сигнал для запроса на завершение работы. Демон должен корректно остановиться.
2. SIGHUP: Традиционный сигнал для запроса на перечитку конфигурации без полной остановки и перезапуска.