

ⓘ Этот общий диалог создан искусственным интеллектом, предназначен только для ознакомления.



ntos_36.txt
TXT 1.06MB

Привет, продолжим.

> Windows управляет питанием с помощью строгой иерархии: пользовательский API обращается к системным сервисам в ядре (`NT`), которые затем делегируют управление оборудованию через драйвер `acpi.sys`.

📖 Основные сценарии управления питанием

1. Выключение и Перезагрузка

Завершение работы происходит через специальный системный сервис:

* **Ключевой системный сервис**: `NtShutdownSystem`. Он является последним этапом, предшествующим фактическому изменению состояния питания.

* **Стек вызовов**: Обычно он вызывается из `winlogon.exe` или `csrss.exe` (в старых версиях). Функция имеет параметр `SHUTDOWN_ACTION`, позволяющий выбрать действие: `ShutdownNoReboot`, `ShutdownReboot` или `ShutdownPowerOff`.

* **Подготовительный этап**: Перед вызовом важно уведомить приложения и драйверы. Сама система после вызова `NtShutdownSystem` перейдет к вызову `NtSetSystemPowerState`.

2. Сон и Гибернация

Переход в энергосберегающие режимы (`S3` — сон, `S4` — гибернация) также инициируется через системные вызовы.

* **Конечная точка в ядре**: Все пользовательские запросы на сон в конечном итоге вызывают ядерный сервис **`NtInitiatePowerAction`**. Стек вызовов выглядит так: `SetSuspendState` или `SetSystemPowerState` → `NtInitiatePowerAction`.

* **Роль `PopSetPowerAction`**: Внутри ядра, в функции `PopSetPowerAction`, устанавливается глобальный флаг (`PopAction`), который управляет предстоящим переходом.

* **Синхронизация привода**: API для синхронизации привода.

Синхронная природа API для сна являются синхронными. Вызов `SetSuspendState` не возвращает управление до тех пор, пока система не проснется и не войдет в рабочее состояние (`S0`).

🧠 Как это устроено: API, Сервисы и IOCTL

Теперь, чтобы ответить на ваш вопрос напрямую: **в основе процесса лежат системные вызовы (syscalls), а не IOCTL**. Используются стандартные механизмы ядра Windows.

Уровень	Что используется	Примечания
:---	:---	:---
	Пользовательский режим (API)	`SetSuspendState` (из `powrprof.dll`) и `SetSystemPowerState` (из `kernel32.dll`).
		`SetSuspendState` предпочтительнее для Windows Vista и новее и не требует явного включения привилегии `SE_SHUTDOWN_NAME`.
	Системный сервис (NT)	`NtInitiatePowerAction` — это системный сервис в ядре (`ntoskrnl.exe`), точка входа для всех операций сна. Как и все системные сервисы, доступен через `syscall` из `ntdll.dll`. Для выключения сервис `NtShutdownSystem` .
	Подготовка аппаратуры	`acpi.sys` взаимодействует с BIOS через порты управления для реального переключения состояний питания. Этот процесс не инициируется через `IOCTL`, а является частью выполнения системного вызова в ядре.

⚙️ Особенности и Нюансы

* **Привилегии**: Для вызова **`SetSystemPowerState`** процесс должен иметь включенную привилегию **`SE_SHUTDOWN_NAME`**. **`SetSuspendState`** управляет этим автоматически.

* **Блокировка перехода**: Драйверы могут временно предотвратить переход в сон, заявляя о занятости системы. Для этого используется, например, функция **`PoRegisterSystemState`**. Это не `IOCTL`, а специальный вызов ядра (`ntoskrnl.exe`).

* **Modern Standby (S0)**: Современные системы используют режим **`S0` (Modern Standby)**, который меняет поведение некоторых событий. Например, системные сервисы могут не получать уведомление **`PBT_APMSUSPEND`** (сигнализирующее о сне), так как ОС "засыпает", не выходя из рабочего состояния. Для таких случаев рекомендуется использовать API **`PowerRegisterSuspendResumeNotification`**.

использовать API `PowerRegisterSuspendResumeNotification` .

Пройди по графу `ntshutdownsystem` и посмотри последовательность вызовов.

Отлично, продолжим. Я проанализировал предоставленный код `ntos_36.txt` и сопоставил его с описанной иерархией управления питанием. В коде мы видим не только реализацию системных сервисов, но и огромное количество внутренней инфраструктуры для управления состоянием системы, процессорами и памятью.

Последовательность вызовов `NtShutdownSystem` → `PopTransitionSystemPowerStateEx`

Давайте пройдем по графу вызовов, используя код из предоставленного файла.

1. Точка входа: `NtSetSystemPowerState`

- Начинается с функции, которая принимает три параметра (`arg_0` , `arg_4` , `arg_8`) и, по сути, является оберткой для `PopTransitionSystemPowerStateEx` .

2. Основная логика: `PopTransitionSystemPowerStateEx`

- Эта функция (начинается с `loc_71EA5B`) выполняет начальную проверку параметров и привилегий, а затем вызывает сервисную функцию `ZwSetSystemPowerState` .
- Если же процесс не имеет нужной привилегии, он пытается получить её через `SeSinglePrivilegeCheck` .
- Если все проверки пройдены, вызывается `PopTransitionSystemPowerStateEx` для выполнения самого перехода.

3. Подготовка к переходу: `PopTransitionSystemPowerStateEx` (продолжение)

- Функция начинает подготовку структуры данных, которая будет описывать переход (`esi`), инициализируя множество полей.
- Вызывается `PopReadShutdownPolicy` для получения политик завершения работы (если состояние `S4`).
- Далее следует блокировка перехода, захват системных ресурсов через `PopAcquireTransitionLock` и `MmLockPagableSectionByHandle` .
- Последний шаг — вызов `PopAcquireRelaxedLock` для монопольного доступа к

- **важный этап** — вызов `PopAcquirePolicyLock` для монопольного доступа к политикам питания. Внутри этой функции (см. `_PopAcquirePolicyLock@0`) мы видим вызов `PsBoostThreadIo` и `ExAcquireResourceExclusiveLite` — классическая блокировка на основе ресурса.

4. Выбор и применение политики:

- Код проверяет глобальные флаги и политики через вызовы `PopFilterCapabilities` и `PopResetActionDefaults`.
- Устанавливается состояние действия питания через `PopSetPowerActionState`.
- **Здесь важно:** процесс не инициируется через `IOCTL`. Вместо этого, он выполняется путем вызова серии внутренних функций ядра, которые напрямую манипулируют структурами данных и драйверами.
- Вызывается `PopExecutePowerAction` для выполнения действия. Эта функция является диспетчером, который определяет, нужно ли переходить в сон, гибернацию или выключаться.

5. Обработка конкретного состояния (Сон/Гибернация):

- В зависимости от требуемого состояния (`Sleep`, `Hibernate`, `Shutdown`) выбирается обработчик из таблицы `_PopPowerStateHandlers`.
- Для гибернации (`S4`) вызывается `PopSaveHiberContext`, который мы видели в коде. Этот процесс включает сохранение состояния системы в файл `hiberfil.sys`.
- `PopSaveHiberContext`: Функция, которая управляет сохранением дампа памяти.

```
asm
```

```
loc_71D61C:          ; Начало основного цикла сохранения
```

```
...
```

```
call _KeDisableInterrupts@0 ; Отключение прерываний на время сохранения
```

```
call _IoInitializeDumpStack@8 ; Инициализация стека для дампа
```

```
call PopMarkComponentsBootPhase ; Подготовка компонентов к дампу
```

```
call PopWriteHeaderPages     ; Запись заголовка файла гибернации
```

```
call _RtlNumberOfClearBits@4 ; Подсчет страниц, которые нужно сохранить
```

```
...
```

```
call PopRequestWrite         ; Непосредственная запись данных на диск
```

6. Выполнение системного состояния (Оборудование):

- Когда все данные сохранены, вызывается `PopInvokeSystemStateHandler`. Эта

функция выполняет системные вызовы для переключения состояний питания процессора и других устройств.

- Внутри `PopInvokeSystemStateHandler` мы видим `call dword ptr [ecx+8]`, который вызывает функцию-обработчик из структуры `_PopPowerStateHandlers`. Этот обработчик может быть, например, из `acpi.sys` и переключит аппаратуру в нужное состояние (например, S3 или S4).

7. Выход из гибернации / Сна:

- После пробуждения управление возвращается в точку после вызова `KeWaitForSingleObject` (см. `loc_71F8B0`), где поток ждал завершения операции.
- Вызываются функции обратного восстановления, такие как `PopRestoreHiberContext` для восстановления состояния из файла гибернации.
- Затем выполняется `PopReleasePolicyLock` для освобождения захваченных ресурсов, и, наконец, управление возвращается в пользовательский режим с кодом ошибки `eax`.

Диаграмма

Код

IOCTL : Почему не он?

Использование системных вызовов (`syscalls`) вместо `IOCTL` объясняется

архитектурой:

- **Централизованное управление:** Питание — это глобальная системная функция. Она контролируется центральным компонентом (`ntoskrnl.exe`), который предоставляет единый интерфейс для всех режимов работы.
- **Безопасность:** Системный вызов `NtInitiatePowerAction` проверяет привилегии и права вызывающего процесса. Использование `IOCTL` отправил бы запрос в конкретный драйвер (например, `acpi.sys`), что позволило бы любому процессу с доступом к этому драйверу влиять на питание, что создает риск безопасности.
- **Атомарность:** Состояния питания затрагивают всю систему. Процесс перехода должен быть атомарным с точки зрения ядра. Драйвер не может сам решать, когда и как переключать питание; он должен подчиняться командам от диспетчера питания ядра.

Таким образом, `IOCTL` используется для управления конкретным устройством через его драйвер (например, отключение питания USB-порта), но общее состояние всей системы (сон, гибернация, выключение) — это исключительная прерогатива менеджера питания ядра Windows.

что в этом `ivt` ?

вот есть инит:

```
loc_7650ED:                ; CODE XREF: NtPowerInformation+4AAj
                        ; DATA XREF: PAGE:007657BEo
```

```
test ebx, ebx
jz  loc_765531
cmp  [ebp+var_360], 0
jnz  loc_765531
cmp  [ebp+var_36C], ecx
jb  loc_76579B
mov  eax, [ebx]
cmp  eax, 7
jnb  loc_765531
mov  ecx, eax
shl  ecx, 4
mov  [ebp+var_38C], ecx
cmp  dword_6C2DA8[ecx], 0
jnz  loc_765513
```

```

loc_765131:                ; CODE XREF: NtPowerInformation+11D4j
                        ; NtPowerInformation+11E9j
                        lea edi, _PopPowerStateHandlers[ecx]
                        mov     esi, ebx
                        movsd
                        movsd
                        movsd
                        movsd

```

Отличный вопрос! Вы нашли очень интересный фрагмент кода — это часть системного сервиса `NtPowerInformation`, который является одним из главных "входов" для получения и установки информации о питании из пользовательского режима.

Давайте разберем этот код, чтобы понять, как он связан с иерархией управления питанием и что такое этот IVT.

Что такое NtPowerInformation ?

`NtPowerInformation` — это системный вызов (`syscall`) в ядре Windows, который предоставляет единый интерфейс для работы с информацией о питании. Он обрабатывает множество различных запросов, идентифицируемых по коду (`eax` в вашем коде).

Анализ вашего фрагмента

1. Проверка параметров:

```

assembly

test     ebx, ebx
jz      loc_765531
cmp     [ebp+var_360], 0
jnz     loc_765531

```

- `ebx` — это, скорее всего, указатель на структуру `POWER_INFORMATION_LEVEL`. Код проверяет, что указатель не `NULL` и что другие параметры корректны.

2. Проверка уровня информации:

```
assembly
```

```
mov    eax, [ebx]
cmp    eax, 7
jnb   loc_765531
```

- Здесь считывается `POWER_INFORMATION_LEVEL` (код запроса) и проверяется, что он меньше 7. Если код ≥ 7 , запрос считается неверным, и функция завершается с ошибкой.

3. Извлечение обработчика:

```
assembly
```

```
mov    ecx, eax
shl   ecx, 4
mov    [ebp+var_38C], ecx
cmp    dword_6C2DA8[ecx], 0
jnz   loc_765513
```

- Код запроса (`eax`) умножается на 16 (сдвиг влево на 4 бита), чтобы получить смещение в таблице функций-обработчиков.
- `dword_6C2DA8` — это, по сути, таблица указателей на функции-обработчики для разных `POWER_INFORMATION_LEVEL`. Скорее всего, каждая запись в таблице имеет размер 16 байт (отсюда сдвиг на 4), и первое поле в этой записи — это указатель на функцию. В вашем случае проверяется, что по смещению `ecx` (которое = код_запроса * 16) лежит не `NULL` (т.е. обработчик существует). Если обработчик есть, переход на `loc_765513` (скорее всего, к выполнению этого обработчика).

4. Копирование обработчика:

```
assembly
```

```
lea   edi, _PopPowerStateHandlers[ecx]
mov   esi, ebx
movsd
movsd
movsd
movsd
```

```
- / \ ~ , \ ,
```

- Если обработчик не найден (для текущего кода запроса), то код переходит сюда.
- `lea edi, _PopPowerStateHandlers[ecx]` — загружает адрес в таблице обработчиков `_PopPowerStateHandlers` (именно эту таблицу я упоминал в предыдущем ответе).
- `mov esi, ebx` — `ebx` здесь указывает на какую-то структуру данных, переданную из пользовательского режима.
- `movsd` (**4 раза**) — это копирование 4-х DWORD (по 4 байта) из `esi` (**пользовательские данные**) в `edi` (**таблица `_PopPowerStateHandlers`**). То есть код динамически перезаписывает таблицу обработчиков, подменяя стандартные обработчики для состояний питания (`PowerSystemWorking`, `PowerSystemSleeping`, `PowerSystemHibernate` и т.д.).

Таблица `_PopPowerStateHandlers`

Это центральная структура данных в менеджере питания Windows. Она содержит указатели на функции, которые выполняют **фактический переход** в определенное состояние питания.

c

```
typedef struct _POP_POWER_STATE_HANDLERS {
    // Каждая запись, скорее всего, содержит:
    ULONG State;           // Текущее состояние питания
    ULONG Flags;          // Флаги для этого состояния
    PVOID Handler;        // Указатель на функцию-обработчик
    PVOID ResumeHandler;  // Указатель на функцию восстановления
} POP_POWER_STATE_HANDLERS;
```

"IVT" в контексте управления питанием

Термин IVT обычно означает **Interrupt Vector Table (Таблица Векторов Прерываний)**. В вашем коде это не совсем "вектор прерываний" в классическом понимании (как в реальном режиме x86), но это **аналогия**.

- Это не IOCTL. `NtPowerInformation` — это системный вызов.
- Это **таблица функций-обработчиков**. Ваш код перезаписывает обработчики для состояний питания. Это мощный механизм, который, вероятно, используется для:

- **Подмены обработчиков для системного сна/гибернации**, например, для реализации "быстрого пробуждения" или "Modern Standby" (S0 Idle).
- **Отладки** — позволяет заменить стандартное поведение системы на отладочное.
- **Виртуализации** — в средах виртуализации (Hyper-V) или эмуляции (WSL) обработчики могут быть подменены на "пустышки" или специальные версии, чтобы гостевая ОС не могла управлять питанием напрямую.

Итог

В этом фрагменте кода:

1. Мы видим системный сервис `NtPowerInformation`, который является одним из главных "входов" для управления питанием.
2. Он использует **таблицу функций-обработчиков** `_PopPowerStateHandlers` для выполнения различных действий (переход в сон, гибернацию, выключение и т.д.).
3. Код **динамически подменяет** записи в этой таблице, что позволяет гибко управлять тем, как система реагирует на запросы на сон.

Этот механизм дает системным компонентам (например, `winlogon.exe`, `services.exe`) и драйверам мощный и гибкий способ управления питанием. Это подтверждает, что управление питанием в Windows — это сложная, централизованная иерархия, основанная на системных вызовах ядра, а не на IOCTL.

Посмотрите граф не для сна, а выключения.

Отлично, давайте проследим путь выключения системы в Windows, используя ваш код и знание архитектуры Windows. Выключение — это отдельный, критически важный путь в коде, который отличается от сна и гибернации.

Граф вызовов для процесса выключения

Я проанализировал предоставленный код и проследил путь от пользовательского запроса до финального отключения питания. Вот полная последовательность:

Диаграмма

Код

Детальный разбор по вашему коду

1. Точка входа: NtSetSystemPowerState

```
assembly
```

```
loc_71EA5B: ; Код, который обрабатывает SYSTEM_POWER_STATE_CONTEXT
```

```
...
```

```
call _PopTransitionSystemPowerStateEx@4
```

Функция `NtSetSystemPowerState` — это системный сервис (syscall), который вызывается из `ntdll.dll`. В вашем коде это видно как `_NtSetSystemPowerState@12` (в начале файла).

2. Подготовка к переходу: PopTransitionSystemPowerStateEx

```
assembly
```

```
loc_71EA5B: ; Проверка параметров
```

```
...
```

```
call SeSinglePrivilegeCheck ; Проверка привилегий SE_SHUTDOWN_NAME
```

```
...
```

```
call _PopAcquirePolicyLock@0 ; Захват блокировки
```

...

```
call _PopSetPowerActionState@4 ; Установка флага PopAction в состояние выключения
```

Эта функция проверяет, имеет ли вызывающий процесс право на выключение системы (привилегия `SE_SHUTDOWN_NAME`), захватывает глобальные блокировки и устанавливает флаг `PopAction` в значение `PopActionShutdown` (код которого равен 1 или 2).

3. Диспетчер действий: `PopExecutePowerAction`

```
assembly

loc_71EC80:      ; Вызов PopExecutePowerAction
push  1
push  dword ptr [esi+4]
lea  eax, [esi+0Ch]
push  eax
call  PopExecutePowerAction
```

`PopExecutePowerAction` — это центральная функция-диспетчер. Она определяет, какое действие было запрошено (выключение, перезагрузка, сон), и вызывает соответствующий обработчик из таблицы `_PopPowerStateHandlers`.

4. Выбор обработчика выключения: `_PopPowerStateHandlers[5]`

```
assembly

; Ваш фрагмент NtPowerInformation, который работает с этой таблицей
lea  edi, _PopPowerStateHandlers[ecx] ; ecx = PowerSystemShutdown (5)
mov  esi, ebx
movsd
movsd
movsd
movsd ; Копирование обработчика выключения
```

Для выключения используется индекс 5 (т.к. `PowerSystemShutdown = 5`). В таблице `_PopPowerStateHandlers` по этому индексу хранится функция-обработчик, которая и будет выполнять выключение.

5. Выполнение выключения: `PopInvokeSystemStateHandler`

```
assembly
```

```

loc_71BA48:      ; PopInvokeSystemStateHandler для stateIndex = 5
...
cmp     ebx, 4   ; Проверка, не является ли состояние S4/S5
jz     short loc_71BA78
cmp     ebx, 5
jz     short loc_71BA78
...
call   _PopGracefulShutdown@4 ; Основная функция выключения

```

Для состояния 5 (выключение) вызывается `PopGracefulShutdown`. Эта функция выполняет "вежливое" выключение системы, уведомляя все подсистемы (файловую систему, драйверы, сервисы) о том, что система выключается.

6. Финальный переход в состояние S5 (PowerOff)

```

assembly

loc_72887D:      ; Код в PopInvokeSystemStateHandler
...
call   ds:__imp__HalReturnToFirmware@4 ; Вызов аппаратного выключения

```

В конце процесса выключения вызывается функция `HalReturnToFirmware` из аппаратно-зависимого уровня (HAL). Эта функция выполняет финальный переход в состояние S5 (Soft Off):

- **Для ACPI:** Вызывается метод `_S5 (_PTS)` через `acpi.sys`, который через `_PTS` и `_S5` инструктирует BIOS выключить питание.
- **Для старых систем:** Через порты ввода-вывода (обычно порт `0xCF9`) отправляется команда на выключение питания.

vs Сравнение с "вводом/выводом" и IOCTL

- **Только системные вызовы!** Как и в случае со сном, выключение не использует IOCTL. Это исключительно работа через системные сервисы ядра.
- **Отличия от сна:**
 - При выключении (S5) система не сохраняет состояние памяти. Все данные, находящиеся в оперативной памяти теряются.

...находясь в оперативной памяти, термостат...

- Процесс выключения не включает фазу "сохранения дампа" (как в гибернации) или "подготовки к выходу из сна".
- Порядок уведомлений подсистем отличается, но принцип "уведомить драйверы → перейти в состояние S5" остается неизменным.

Итог

Граф вызовов для выключения в Windows выглядит следующим образом:

Пользовательский процесс → NtSetSystemPowerState → PopTransitionSystemPowerStateEx → PopExecutePowerAction → _PopPowerStateHandlers[5] → PopInvokeSystemStateHandler → PopGracefulShutdown → **Уведомление подсистем** → HalReturnToFirmware → **Выключение питания.**

Это полностью централизованный, защищенный и **не** IOCTL путь, который гарантирует, что система корректно завершит работу перед выключением питания.

Странно:

The following routine is not supported and should not be used:

HalReturnToFirmware

Вы абсолютно правы! Это очень важное замечание. Давайте разберемся, почему HalReturnToFirmware помечена как "неподдерживаемая" и что на самом деле происходит при выключении в современных версиях Windows.

Почему HalReturnToFirmware не поддерживается?

Документация Microsoft действительно указывает, что HalReturnToFirmware (и другие HAL-функции) **не предназначены для прямого вызова** из драйверов или приложений пользовательского режима. Это внутренняя функция HAL (Hardware Abstraction Layer), которая:

1. **Является внутренним API:** HAL — это слой абстракции между ядром Windows и аппаратным обеспечением. Его функции предназначены только для внутреннего использования ntoskrnl.exe и критически важных системных компонентов.

2. **Меняется между версиями:** Реализация HAL может меняться в зависимости от версии Windows и типа оборудования (например, для ACPI или для старых BIOS-систем).
3. **Нестабильна для вызова:** Прямой вызов этой функции из непредназначенного контекста может привести к нестабильности системы, зависанию или синему экрану.

Что на самом деле происходит при выключении?

Вместо прямого вызова `HalReturnToFirmware` современные версии Windows используют **многоуровневый и документированный подход:**

Диаграмма

Код

Реальный путь выключения в современных системах:

1. `PopGracefulShutdown` выполняет все необходимые подготовительные шаги:
 - Уведомляет все зарегистрированные драйверы через `IRP_MJ_SHUTDOWN`.
 - Сбрасывает кэши файловых систем.
 - Завершает все системные процессы.
2. `PopInvokeSystemStateHandler` с индексом 5 вызывает обработчик для состояния

PowerSystemShutdown .

3. Обработчик в acpi.sys :

- Вместо прямого вызова HAL, acpi.sys выполняет **AML-код** (ACPI Machine Language), который находится в прошивке BIOS/UEFI.
- AML-код вызывает метод `_PTS` (Prepare To Sleep) с параметром `5` (`S5 — Soft Off`).
- Затем вызывается метод `_S5` (Soft Off), который инструктирует чипсет материнской платы выключить питание.

4. Аппаратное выключение:

- Чипсет получает команду выключения питания.
- Процессор перестает выполнять инструкции.
- Блок питания отключает напряжение на компонентах системы.



Почему в коде виден HalReturnToFirmware ?

То, что вы видите в коде, — это **внутренняя реализация** для конкретной архитектуры (x86/x64). В вашем файле `ntos_36.txt` видно:

```
assembly  
  
loc_72887D:  
    call ds:__imp__HalReturnToFirmware@4
```

Это действительно вызов HAL-функции, но:

1. **Это внутренний код ядра:** Этот вызов происходит внутри `ntoskrnl.exe` в совершенно безопасном контексте — после того, как все драйверы уже получили уведомление о выключении и система готова к остановке.
2. **HAL — это прослойка:** `HalReturnToFirmware` в случае ACPI-систем фактически просто вызывает AML-метод `_S5` через `acpi.sys`. Это не примитивная команда "выключить питание", а документированное взаимодействие с ACPI.
3. **Историческая причина:** В ранних версиях Windows (до ACPI) действительно существовал прямой вызов BIOS через прерывания или порты ввода-вывода. Сейчас это абстрагировано.

✓ Вывод

`HalReturnToFirmware` действительно не должна вызываться из драйверов или приложений пользовательского режима, но для ядра Windows это **внутренняя, рабочая функция**, которая является последним звеном в цепочке выключения. Она обеспечивает корректный переход в состояние S5 через ACPI или другие аппаратные механизмы.

Поэтому в коде вы видите реальную рабочую последовательность выключения, которая завершается вызовом этой HAL-функции из контекста ядра, когда система уже полностью готова к остановке питания.

Продолжить с DeepSeek