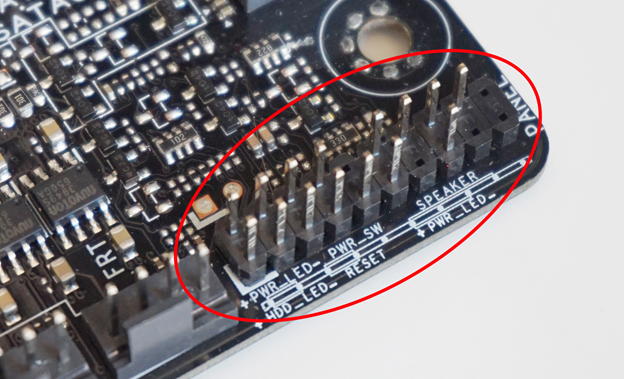
От включения до полной загрузки Windows

* [C++](https://habr.com/ru/hub/cpp/),
* [Системное программирование](https://habr.com/ru/hub/system_programming/),
* [Разработка под Windows](https://habr.com/ru/hub/win_dev/),
* [Компьютерное железо](https://habr.com/ru/hub/hardware/)

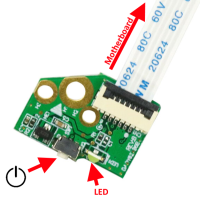
Мы продолжаем разбираться как работает ПК на примере клавиатуры и Windows 10. В этой статье поговорим о том как происходит единение софта и железа.

Старт системы

Полностью компьютер выключен когда он отключен от питания и конденсаторы на материнской плате разрядились. До эры смартфонов мобильные телефоны часто глючили и если перезагрузка не лечила проблему, то приходилось доставать батарею и ждать 10 секунд, потому что сбрасывалось программное состояние ОС, в то время как чипы на материнской плате и контроллеры устройств оставались активными сохраняя состояние, драйвера ОС к ним просто реконнектились. 10 секунд — время на разрядку конденсаторов, состояние чипов сбрасывается только при полном отключении.  
Если же ПК подключен к розетке или батарее, то он находится в режиме Stand-By, это значит что по шине питания подаётся маленькое напряжения (5В) от которого запитываются некоторые чипы на материнке. Как минимум это *системный контроллер*, по сути это мини-компьютер запускающий большой компьютер. Получив уведомление о нажатии кнопки Power он просит блок питания/батарею подать больше напряжения и после инициализирует весь чип-сет, в том числе и процессор. Инициализация включает в себя перекачку кода и данных прошивки материнки (*BIOS/UEFI*) в оперативную память и настройку CPU на её исполнение.  
Думать что кнопка Power это рубильник который подаёт электричество на CPU и тот начинает исполнять с заранее известного адреса прошивку BIOS неправильно. Возможно старые компьютеры так и работали. Кнопка включения находится на своей плате, вместе со светодиодами состояний и к материнке она подключается через специальный разъём. На картинке ниже видны контакты для кнопки Power, Reset, а также светодиодов с состоянием Power и чтения жёсткого диска. Нажатие кнопки включения переводится в сигнал на контакты материнки, откуда он достигает системный контроллер.



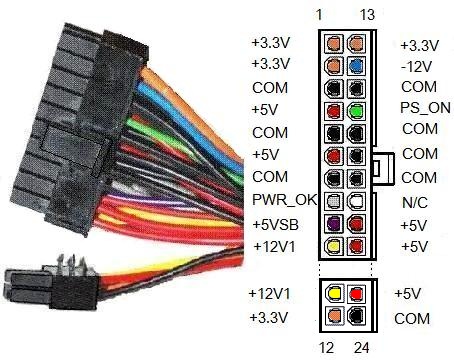
*Контакты на материнке для подключения кнопки включения, светодиодов состояния Power, жёсткого диска и динамиков.*



*Плата ноутбука с кнопкой включения и светодиодом состояния*  
  
Cистемный контроллер обладает огромными полномочиями – включать и выключать компьютер, исполнять код в режиме ядра. Помимо него могут быть и другие чипы со сравнимыми возможностями, такие как *Intel Management Engine* или *AMD Secure Technology* (часть CPU), которые так же работают когда компьютер «выключен». Чип с Intel ME имеет в себе x86 CPU с операционной системой [MINIX 3](https://ru.wikipedia.org/wiki/MINIX_3). Что он может делать:

1. Включать и выключать компьютер, т.е. выполнять программы имея доступ ко всей вычислительной мощности, периферии машины и сети.
2. Обходить ограничения файервола.
3. Видеть все данные в CPU и RAM, что даёт доступ к запароленным файлам.
4. Красть ключи шифрования и получать доступ к паролям
5. Логировать нажатия клавиш и движения мыши
6. Видеть что отображается на экране
7. Вредоносный код в Intel ME не может быть детектирован антивирусом, потому как на такой низкий уровень он добраться не может
8. И конечно же скрытно отправлять данные по сети используя свой стек для работы с сетью.

Это вызывает серьёзные вопросы безопасности, потому как он может быть хакнут или использовать в шпионских целях.  
  
*Прикладная иллюстрация как блок питания получает сигнал от материнки на включение. Если вы задумаете установить мощную видеокарту (Nvidia 2070 S) на офисный ПК, то просто вставить её недостаточно, потому как она требует питание в 600W, в то время как такой ПК имеет блок на ~500W. Первое что придёт в голову – купить новый блок питания на 650W с отдельной линией для видеокарты. Но и здесь будут разочарования, потому как разъёмы материнки будут не совпадать с разъёмами БП, а если его отдельно воткнуть в розетку и подключить к видюхе тоже ничего не будет – в блоке питания вентилятор не крутится и изображения нет. Так происходит, потому что БП должен получить сигнал от материнки на полное включение. Очевидное решение – новая материнка с совместимыми разъёмами, однако она стоит ~$300. Есть решение проще, хоть оно и вызывает опасения пожаробезопасности. Берём скрепку, разгибаем и вставляем в зелёный (PS\_ON) и один из чёрных пинов (COM). Теперь всё должно работать.*



Поиск загрузчика ОС

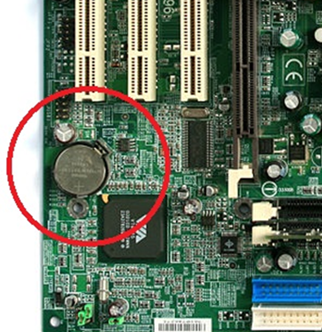
Есть два вида прошивки материнки – *BIOS (Basic Input Output System)* на старых машинах и *UEFI (Unified Extensible Firmware Interface)* на новых. Windows 10 поддерживает обе и абстрагирует различия между ними. UEFI правильней называть ОС чем прошивкой, потому как он предлагает больше возможностей, к примеру богатый графический интерфейс вместо текстового, наличие мышки, больший объём доступной памяти, улучшенная модель безопасности и валидации файлов ОС, взаимодействие с железом через API, вместо прерываний как в BIOS.



*Пример экрана монитора BIOS.*  
  
Программа BIOS хранится на отдельном чипе, подключенном к Южному мосту. Этот чип можно достать и перепрошить новой программой, по факту это просто носитель памяти, а не самостоятельный микрокомпьютер.



Настройки BIOS (системное время, например), хранятся на другом чипе который как правило находится возле круглой батарейки, которая на самом деле является литиевым аккумулятором, подзаряжающимся во время работы ПК. Называется он *CMOS*, что означает *Complementary Metal Oxide Semiconductor*, а по-русски просто — КМОП, что есть *комплементарная структура металл-оксид-полупроводник*.



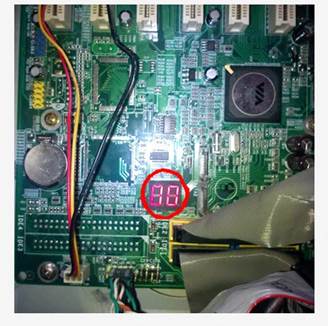
Первым делом программа BIOS выполняет проверку подсистем, эта процедура называется [POST](https://ru.wikipedia.org/wiki/POST_(%D0%B0%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5)) – Power On Self Test. Тест может быть сокращённый либо полный, это задаётся в настройках BIOS. Процитирую [Википедию](https://ru.wikipedia.org/wiki/POST_(%D0%B0%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5)), что в себя включают эти тесты:  
Сокращённый тест включает:

1. Проверку целостности программ BIOS в ПЗУ, используя контрольную сумму.
2. Обнаружение и инициализацию основных контроллеров, системных шин и подключённых устройств (графического адаптера, контроллеров дисководов и т. п.), а также выполнение программ, входящих в BIOS устройств и обеспечивающих их самоинициализацию.
3. Определение размера оперативной памяти и тестирования первого сегмента (64 килобайт).

Полный регламент работы POST:

1. Проверка всех регистров процессора;
2. Проверка контрольной суммы ПЗУ;
3. Проверка системного таймера и порта звуковой сигнализации (для IBM PC — ИМС i8253 или аналог);
4. Тест контроллера прямого доступа к памяти;
5. Тест регенератора оперативной памяти;
6. Тест нижней области ОЗУ для проецирования резидентных программ в BIOS;
7. Загрузка резидентных программ;
8. Тест стандартного графического адаптера (VGA или PCI-E);
9. Тест оперативной памяти;
10. Тест основных устройств ввода (НЕ манипуляторов);
11. Тест CMOS
12. Тест основных портов LPT/COM;
13. Тест накопителей на гибких магнитных дисках (НГМД);
14. Тест накопителей на жёстких магнитных дисках (НЖМД);
15. Самодиагностика функциональных подсистем BIOS;
16. Передача управления загрузчику.

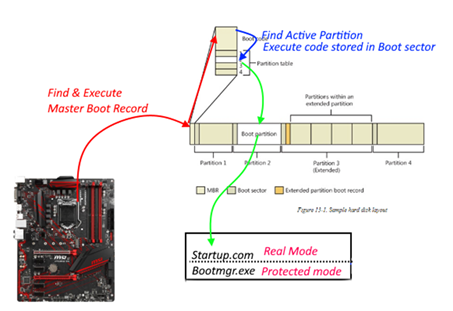
По результатам этого теста может быть обнаружена неисправность, к примеру нерабочая видеокарта или клавиатура. Поскольку экран монитора может не работать результаты тестов сообщаются в виде последовательности звуковых сигналов разной высоты. Что конкретно они значат надо смотреть в документации к материнской плате. Старые компьютеры часто бибикали во время старта — это программа BIOS сообщала о результатах тестов. Иногда может дополнительно использоваться индикатор, показывающий номер ошибки.



Если всё прошло успешно, BIOS начинает процесс поиска загрузчика ОС. Для этого он начинает просматривать все подключенные к материнской плате жёсткие диски. Данные на физических дисках адресуются в единицах называемых *сектор*, обычно он 512 байт, однако современный стандарт – 4096 байт. Установщик Windows в самый первый сектор на диске записывает специальный программный код и данные о разделах. Этот сектор называется *Master Boot Record*. Диск разбивается на разделы (partitions), отформатированный своей файловой системой. Максимум 4 раздела, каждый из который может быть *расширенным (extended partition)*, такой можно рекурсивно делить на 4 раздела и теоретически их число не ограничено. Как только BIOS находит Master Boot Record он считывает оттуда код и передаёт ему управление. Этот код поочередно просматривает данные о разделах и находит тот который помечен как активный, в нём находится код загрузчика Windows (Это **не** раздел с C:\Windows\System32!), этот раздел называется *system partition*. Как правило он занимает 100Мб и скрыт от пользователя. В первом секторе этого раздела хранится загрузочный код, которому передаётся управление. Это *volume boot sector*, код в нём ищет файл *Bootmgr*, с которого и начинается процесс загрузки Windows. Файл Bootmgr создан через соединение в один файлов *Startup.com* и *Bootmgr.exe*.  
  
Процессор начинает свою работу в режиме который называется *«Реальный»*. Это режим совместимости, в нём CPU работает так же как и старые 16-bit процессоры, не имевшие поддержки виртуальной памяти и работавшие напрямую с физической памятью через 20-bit шину адресов, позволявшую адресовать 1Мб памяти. Простые MS-DOS программы выполнялись в этом режиме и имели расширение .COM. Первое что делает Startup.com (Bootmgr) – переключает процессор в режим *«Защищённый»*, где под защитой понимается защита процессов друг от друга. В этом режиме поддерживается виртуальная память и 32х битные адреса, которыми можно адресовать 4Гб оперативной памяти. Следующим этапом Bootmgr заполняет таблицу виртуальных адресов на первые 16Мб RAM и включает трансляцию с виртуальных адресов в физические. В этом режиме и работает Windows. Поскольку на этом этапе подсистемы ОС ещё не созданы, Bootmgr имеет свою простую и неполную реализацию файловой системы NTFS, благодаря которой он находит *BCD файл (Boot Configuration Data)*, в котором хранятся настройки параметров загрузки ОС. Вы можете редактировать его через утилиту *BcdEdit.exe*. В этих настройках BCD может быть указано, что Windows была в состоянии гибернации, и тогда Bootmgr запустит программу *WinResume.exe*, которая считывает состояние из файла *Hyberfil.sys* в память и перезапускает драйвера. Если BCD говорит, что есть несколько ОС, то Bootmgr выведет на экран их список и попросит пользователя выбрать. Если ОС одна, то Bootmgr запускает WinLoad.exe, этот процесс и выполняет основную работу по инициализации Windows:

1. Выбирает соотвествующую версию ядра Windows. Можете думать о нём как о Windows10.exe, хотя на самом деле он называется NtOsKrnl.exe. Какие есть версии? Согласно википедии:
   * ntoskrnl.exe — однопроцессорное ядро Windows. без поддержки режима [PAE](https://ru.wikipedia.org/wiki/PAE)
   * ntkrnlmp.exe (англ. NT Kernel, Multi-Processor version) — многопроцессорное ядро Windows. без поддержки режима [PAE](https://ru.wikipedia.org/wiki/PAE)
   * ntkrnlpa.exe — однопроцессорное ядро Windows с поддержкой режима [PAE](https://ru.wikipedia.org/wiki/PAE).
   * ntkrpamp.exe — многопроцессорное ядро Windows с поддержкой режима [PAE](https://ru.wikipedia.org/wiki/PAE).
2. Загружает HAL.dll (Hardware Abstraction Layer), который абстрагирует особенности материнки и CPU.
3. Загружает файл шрифтов vgaoem.fon
4. Загружает файлы в которых содержится инфомрация о представлениях даты времени, форматов чисел и пр. Эта функциональность называется [National Language System](https://docs.microsoft.com/en-us/windows/win32/intl/about-national-language-support).
5. Загружает в память реестр SYSTEM, в нём содержится информация о драйверах которые надо загрузить. Информация о всех драйверах находится в *HKLM\SYSTEM\CurrentControlSet\Services\*. Драйвера которые надо загрузить имеют ключ start = SERVICE\_BOOT\_START (0). Об устройстве реестра мы поговорим в другой статье.
6. Загружает драйвер файловой системы для раздела на котором располагаются файлы драйверов.
7. Загружает драйвера в память, но пока не инициализирует их из-за круговых зависимостей.
8. Подготавливает регистры CPU для выполнения ядра Windows выбранного на первом шаге – NtOsKrnl.exe.

Во время загрузки драйверов WinLoad проверяет их цифровые подписи и если они не совпадают, то будет синий (*BSOD*) или зелёный (*GSOD*, для insider preview сборок) «экран смерти».



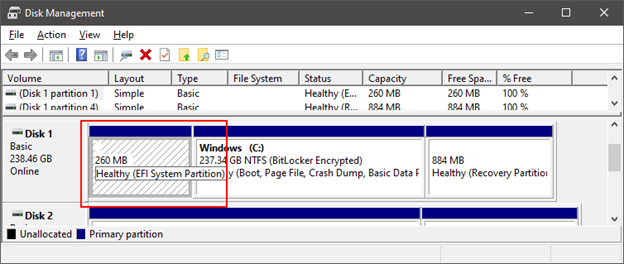
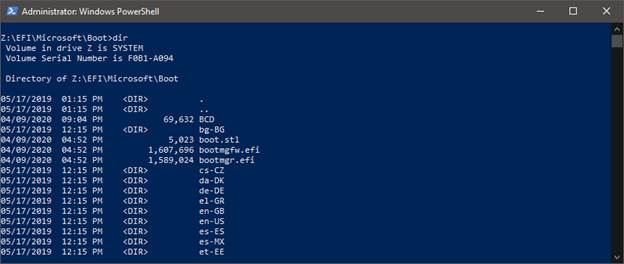
Запуск на UEFI



*Пример экрана загрузки UEFI*  
  
BIOS существует больше 30 лет и в попытках исправить его недостатки компания Intel в 1998 году создала стандарт Intel Boot Initiative, позже переименованный в EFI и в 2005 году пожертвованный организации EFI Forum. Недостатки BIOS:  
• Работает только в 16-битном режиме  
• Может адресовать только 1Mb оперативной памяти  
• Часто имеет проблемы совместимости  
• MBR ограничен только четырьмя главными разделами диска  
• Диск с ОС не может быть больше чем 2.2Tb.  
• Имеет очень ограниченные возможности для валидации загрузчика ОС.  
На смену BIOS пришёл UEFI, по сути это миниатюрная ОС которая может работать и в 32-bit и в 64-bit. Для совместимости есть опция *Compatibility Support Module*, которая включается в настройках и эмулирует работу BIOS.



В UEFI загрузка происходит в родной для процессора битности – 32 или 64, есть доступ ко всей памяти, поддерживается виртуальная память, включен*Secure Boot* и есть возможность запустить antimalware до начала загрузки ОС. Порядок загрузки ОС в UEFI:

1. Инициализация и запуск Firmware, запуск чип-сета.
2. POST тест, аналогично BIOS
3. Загрузка EFI-драйверов и поиск диска подпадающего под требования EFI для загрузочного диска
4. Поиск папки с именем EFI. Спецификация UEFI требует чтобы был раздел для *EFI System Partition*, отформатированный под файловую систему FAT, размером 100Мб – 1Гб или не более 1% от размера диска. Каждая установленная Windows имеет свою директорию на этом разделе – *EFI\Microsoft*.  
   
5. Читает из настроек UEFI сохранённых в NVRAM (энергонезависимая память) путь к файлу загрузчика.
6. Находит и запускает *EFI/Microsoft/Boot/BootMgrFw.efi*.
7. BootMgrFw.efi находит раздел реестра BCD, который хранится в отдельном файле с именем BCD. Из него он находит *WinLoad.efi*, который расположен в *C:\Windows\System32\winload.efi*.

Чтобы посмотреть содержимое раздела *EFI System Partition* откройте консоль с правами админа (WinKey+X => Windows PowerShell (Admin)) и выполните команды *mountvol Z: /s, Z:, dir*. *CD* — меняет директорию.  
Главное отличие компонентов BootMgr и WinLoad для UEFI от своих копий для BIOS тем что они используют *EFI API*, вместо прерываний BIOS и форматы загрузочных разделов *MBR BIOS* и *EFI System Partition* сильно отличаются.

Инициализация ядра

Напоминаю, что мы рассматриваем загрузку ПК в контексте работы клавиатуры, поэтому не стоит заострять внимание на всех этапах. Надо понять где в этом процессе находится клавиатура, важные для понимания этапы **выделены**.  
На предыдущем этапе был запущен компонент *WinLoad.exe/WinLoad.efi*, который запускает *NtOsKrnl.exe* указав ему параметры загрузки в глобальной переменной *nt!KeLoaderBlock* (память режима ядра доступна всем процессам), которые *WinLoad*собрал во время своей работы. Они включают:

1. Пути к System (загрузчик Windows) и Boot (*C:\Windows\System32*) директориям.
2. Указатель на таблицы виртуальной памяти которые создал WinLoad
3. Дерево с описанием подключенного hardware, оно используется для создания HKLM\HARDWARE ветки реестра.
4. Копия загруженного реестра *HKLM\System*
5. Указатель на список загруженных (но не инициализированных) драйверов участвующих в старте Windows.
6. Прочая информация необходимая для загрузки.

Инициализация ядра Windows происходит в два этапа. До этого происходит инициализация *Hardware Abstraction Layer*, который в числе всего прочего настраивает контроллеры прерывания для каждого CPU.  
На этой же стадии загружаются в память строки с сообщениями для BSOD, потому как в момент падения они могут быть недоступны или повреждены.

* Первая фаза инициализации ядра:
  1. *Слой Executive*инициализирует свои объекты состояний – глобальные объекты, списки, блокировки. Производится проверка *Windows SKU (Stock Keeping Unit)*, примеры Windows 10 SKU — Home, Pro, Mobile, Enterprise, Education.
  2. Если включен *Driver Verifier*, то он инициализируется.
  3. Менеджер памяти создаёт структуры данных, необходимые для работы внутренних API для работы с памятью (*memory services*), резервирует память для внутреннего пользования ядром.
  4. Если подключен отладчик ядра (*kernel debugger*) ему отправляется уведомление загрузить символы для драйверов загружаемых во время старта системы.
  5. Инициализируется информация о версии билда Windows.
  6. Старт Object Manager – позволяет регистрировать именованные объекты к которым могут получать доступ по имени другие компоненты. Яркий пример – мьютекс по которому приложение позволяет запустить единственный экземпляр. Здесь же создаётся храниться *handle table*, по которой устанавливается соответствие к примеру между HWND и объектом описывающим окно.
  7. Старт*Security Reference Monitor* подготавливает всё необходимое для создания первого аккаунта.
  8. *Process Manager* подготавливает все списки и глобальные объекты для создания процессов и потоков. Создаются процесс Idle и System (в нём исполняется “Windows10.exe” он же NtOsKrnl.exe), они пока не исполняются, потому как прерывания выключены.
  9. Инициализация *User-Mode Debugging Framework*.
  10. **Первая фаза инициализации Plug and Play Manager.** PnP – это стандарт который реализовывается на уровне производителей периферии, материнских плат и ОС. Он позволяет получать расширенную информацию о подключенных устройствах и подключать их без перезагрузки ПК.
* Вторая фаза инициализации ядра. Она содержит 51 шаг, поэтому я пропущу многие из них:
  1. По завершению первой фазы главный поток процесса *System (NtOsKrnl.exe)* уже начал исполнение. В нём производится вторая фаза инициализации. Поток получает самый высокий приоритет – 31.
  2. **HAL настраивает таблицу прерываний и включает прерывания.**
  3. Показывается *Windows Startup Screen*, которая по умолчанию представляет из себя чёрный экран с progress bar.
  4. Executive слой инициализирует инфраструктуру для таких объектов синхронизации как Semaphore, Mutex, Event, Timer.
  5. Объекты для User-Mode Debugger проинициализированы.
  6. Создана symbolic link \SystemRoot.
  7. *NtDll.dll*отображена в память. Она отображается во все процессы и содержит Windows APIs.
  8. Инициализирован драйвер файловой системы.
  9. Подсистема межпроцессного общения между компонентами *Windows ALPC* проинициализирована. Можете думать о ней как о named pipes или Windows Communication Foundation для межпроцессного общения.
  10. **Начинается инициализация I/O Manager**, который создаёт необходимые структуры данных для инициализации и хранения драйверов подключенной к компьютеру периферии. Этот процесс очень сложный.  
      Здесь же инициализируются компоненты *Windows Management Instrumentation* и *Event Tracing for Windows* (на него полагается *[Windows Performance Analyzer](https://docs.microsoft.com/en-us/windows-hardware/test/wpt/windows-performance-analyzer)*). После этого шага все драйвера проинициализированы.
  11. Запускается процесс *SMSS.exe* (*Session Manager Sub System*). Он отвечает за создание режима пользователя, в котором будет создана визуальная часть Windows.

Запуск подсистем – SMSS, CSRSS, WinInit

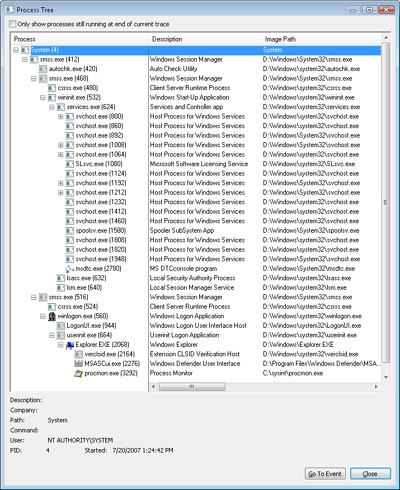
*SMSS.exe* отличается от пользовательских процессов, это нативный процесс и это даёт ему дополнительные полномочия. SMSS.exe работает с ядром в обход Windows API, он использует то что называется *Native API*. Windows API – обёртка вокруг Native API. SMSS.exe первым делом запускает подсистему Windows (*CSRSS.exe – Client Server Runtime Sub System*) и заканчивает инициализацию реестра.  
  
Процесс и потоки SMSS.exe помечены как критические, это значит что если они неожиданно завершаться, к примеру из-за ошибки, это приведёт к падению системы. Для общения с подсистемами, к примеру вызову API создающему новую сессию, *SMSS*создаёт ALPC-порт с именем *SmApiPort*. Загружаются из реестра переменные среды окружения, запускаются программы такие как Check Disk (autochk.exe, эти программы записаны в реестре *HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute*). SMSS.exe запускается для каждой пользовательской сессии. Глобальные переменные (очередь сообщений например) у каждой сессии своя за счёт механизма виртуальной памяти. В Windows есть контексты потока, процесса и сессии. Каждый SMSS.exe запускает свой экземпляр подсистемы, на данный момент это только CSRSS.exe (Windows), в прошлом поддерживались операционные системы OS/2 (os2ss.exe) и POSIX (psxss.exe), но эта идея была неудачной. Самый первый SMSS.exe засыпает в ожидании процесса WinInit.exe. Остальные экземпляры вместо этого создают процесс WinLogon который показывает UI для входа.  
  
*WinInit.exe* инициализирует подсистемы для создания графической оболочки – *[Windows Station](https://docs.microsoft.com/en-us/windows/win32/winstation/window-stations)* и [десктопы](https://docs.microsoft.com/en-us/windows/win32/winstation/desktops), это не тот рабочий стол который вы видите, это иная концепция Windows. Далее он запускает процессы:

1. *Services.exe* –*Services Control Manager (SCM)* запускает сервисы и драйвера помеченные как AutoStart. Сервисы запускаются в процессах *svchost.exe*. Есть утилита *tlist.exe*, которая запущенная с параметром *tlist.exe -s* напечатает в консоли имена сервисов в каждом из svchost.exe.
2. *LSASS.exe* – Local System Authority.
3. *LSM.exe* – Local Session Manager.

*WinLogon.exe* – загружает провайдеры аутентификации (*credential providers*), которые могут быть password, Smartcard, PIN, Hello Face. Он порождает процесс *LogonUI.exe* который и показывает пользователю интерфейс для аутентификации, а после валидирует введённые данные (логин и пароль, PIN).  
  
Если всё прошло успешно, то WinLogon запускает процесс указанный в ключе реестра *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\WinLogon\Userinit*. По умолчанию это процесс *UserInit.exe*, который:

1. Запускает скрипты указанные в реестрах:
   * HKCU\Software\Policies\Microsoft\Windows\System\Scripts
   * HKLM\SOFTWARE\Policies\Microsoft\Windows\System\Scripts
2. Если групповая политика безопасности определяет User Profile Quota, запускает *%SystemRoot%\System32\Proquota.exe*
3. Запускает оболочку Windows, по умолчанию это Explorer.exe. Этот параметр конфигурируется через реестр:
   * HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell
   * HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell

WinLogon уведомляет *Network Provider* о залогинившемся пользователе, на что тот восстанавливает и подключает системные диски и принтеры сохранённые в реестре. Network Provider представляет из себя файл *mpr.dll* из системной папки, который хостится в процессе *svchost.exe*, т.е. сервис Windows.  
  
Дерево процессов выглядит следующим образом, на нём можно увидеть кто и кого создал (показаны не все процессы, может немного отличаться от последний версий Windows).



Где здесь клавиатура?

Во время запуска ядро Windows считывает из реестра информацию о контроллере системной шины, как правило это шина PCI (реже MSI), к ней подключены контроллеры портов ввода-вывода, в том числе и USB, PS/2. Информация о нём записывается во время установки Windows. Система загружает для него драйвер и рекурсивно обходит все порты так же загружая для каждого из них свой драйвер. Драйвера могут комбинироваться в узлы (*driver node*), к примеру драйвер клавиатуры, будет соединён с драйвером порта PS2. А вот порт USB сложнее — сначала драйвер порта, потом драйвер для работы с протоколом HID и только потом клавиатура.  
  
Каждый порт контроллируется своим чипом, который мониторит подключение, принимает/отправляет сигналы между CPU и устройством. Если чип-сет Южный мост не встроен в CPU, как это часто делают в ноутбуках, а существует отдельным чипом на материнке, то правильней говорить: сигнал между Южным мостом и контроллером порта. Чип контроллирующий порт имеет выделенную линию с контроллером прерываний (PIC или APIC), по которой он может попросить обратить на себя внимание CPU, к примеру считать данные от клавиатуры (порт PS/2, с USB другая история). Поскольку ОС загрузила для порта драйвер, она может отдавать ему команды, читать и отправлять данные. В нашем примере был загружен драйвер из *C:\Windows\System32\i8042prt.sys*. Давайте вспомним [предыдущую статью](https://habr.com/ru/post/489166/). В старых компьютерах с PIC на чипе *Intel 8259* было 15 линий прерываний, где клавиатура была подключена к ножке IRQ1, таймер IRQ0, а мышка к IRQ12, который на самом деле был пятой ножкой второго чипа 8259, который мультиплексировал свои прерывания через ножку IRQ2 первого контроллера. В современных PIC могут быть 255 контактов для сигналов прерываний. Во время загрузки ОС программирует APIC/PIC возвращать определённое число когда скажем пришло прерывание от порта клавиатуры или USB и по этому номеру CPU находит в*таблице векторов прерываний* функцию которую надо выполнить. Номер прерываний определяют *HAL* и *Plug’n’Play Manager*. Контроллер прерываний ищет сигнал на своих ножках в определённом порядке, к примеру в бесконечном цикле проверяет напряжение на ножках от 1 до MAX\_PIN. Этот порядок определяет приоритет, к примеру клавиатура будет замечена раньше мышки, а таймер раньше клавиатуры. Чтобы не зависеть от особенностей работы контроллеров прерываний Windows абстрагирует концепцию *IRQ (Interrupt Request)* в *IRQL (Interrupt Request Level)*. Будь у контроллера прерываний хоть 15 хоть 255 линий они все будут отображены на 32 IRQL для x86 и 15 IRQL для x64 и IA64.

|  |  |
| --- | --- |
| https://habrastorage.org/webt/t_/qv/us/t_qvust1xn4vcfd2jdohtuobuyk.jpeg | https://habrastorage.org/webt/pb/fo/ix/pbfoix2zoiino27wde6kpewiavm.jpeg |

Что означают приоритеты IRQL:

1. High – когда происходит краш системы, обычно это вызов функции KeBugCheckEx.
2. Power Fail – не используется. Изначально был придуман для Windows NT.
3. Interprocessor Interrupt – нужен отправить запрос другому CPU на мультипроцессорной системе выполнить действие, например обновить TLB cache, system shutdown, system crash (BSOD).
4. Clock – нужен чтобы обновлять системные часы, а так же вести статистику сколько времени потоки проводят в режиме пользователя и ядра.
5. Profile – используется для real-time clock (local APIC-timer) когда механизм kernel-profiling включен.
6. Device 1 … Device N – прерывания от устройств I/O. Во время прерывания данные от клавиатуры, мыши и других устройств считываются в отдельные буфера и сохраняются в объектах типа DPC (Deferred Procedure Call), чтобы обработать их позже и дать возможность устройствам переслать данные. После приоритет снижается до Dispatch DPC
7. Dispatch DPC — как только данные от устройств получены можно начинать их обрабатывать.
8. APC — [Asynchronous Procedure Call](https://docs.microsoft.com/en-us/windows/win32/sync/asynchronous-procedure-calls). Через этот механизм вы можете исполнить код когда поток будет спать вызвав WaitForSingleObject, Sleep и другие.
9. Passive/Low — здесь исполняются все приложения в User Mode.

Если вы всегда программировали в режиме пользователя, то никогда не слышали про IRQL, потому что все пользовательские программы выполняеются с приоритетом Passive/Low (0). Как только происходит событие с б*о*льшим уровнем приоритета (событие от клавиатуры, таймер планировщика потоков), процессор сохраняет состояние прерванного потока, которое представляет из себя значения регистров CPU, и вызывает диспетчер прерываний (*interrupt dispatcher*, просто функция), который повышает приоритет IRQL через API *KeRaiseIrql*в HAL и вызывает непосредственно сам код обработчика (*interrupt’s service routine*). После этого IRQL CPU понижается до прежнего уровня через функцию *KeLowerIrql* и прерванный поток начинает обработку с того же места где его прервали. На этом механизме основан *планировщик потоков*. Он устанавливает таймер, который с определённым интервалом (квант времени) генерирует прерывание с приоритетом DPC/Dispatch (2) и в своей *interrupt’s service routine* по определённому алгоритму назначает новый поток на исполнение.  
  
Механизм IRQL реализовывается на уровне софта в Hardware Abstraction Layer (*HAL.dll*), а не железа. В Windows системах есть драйвер шины (*bus driver*), который определяет наличие устройств подключенных к шинам – PCI, USB и др. и номера прерываний которые могут быть назначены каждому устройству. Драйвер шины сообщает эту информацию Plug and play manager, который уже решает какие номера прерываний назначить каждому устройству. Далее арбитр прерываний внутри PnP Mgr (*PnP interrupt arbiter*) устанавливает связи между IRQ и IRQL.  
  
Когда приходит прерывание от клавиатуры, любой исполняемый в данный момент поток (это может быть ваша программа) назначается на его обработку. Interrupt dispatcher повышает приоритет IRQL CPU до одного из уровней *Device1-DeviceN*. После этого менеджер виртуальной памяти не сможет найти страницу если она не загружена в RAM (не сможет обработать *Page Fault*), планировщик потоков не сможет прервать выполнение, потому что они все работают с меньшим уровнем IRQL. Главная задача драйвера клавиатуры в этот момент считать полученные данные и сохранить их для дальнейшей обработки. Данные записываются в объект типа \_DPC ([Deferred Procedure Call](https://ru.wikipedia.org/wiki/DPC)), который сохраняется в список DPC потока (что-то вроде *std::list<DPC>*, в ядре ОС вместо массивов используются связанные списки). Как только прерывания от всех внешних устройств обработаны, IRQL потока понижается до уровня DPC в котором и производится обработка отложенных процедур (DPC). В коде обработчика DPC для клавиатуры вызывается функция из драйвера клавиатуры [Kbdclass.sys](https://docs.microsoft.com/en-us/previous-versions/ff542324(v%3Dvs.85)):

VOID KeyboardClassServiceCallback(

\_In\_ PDEVICE\_OBJECT DeviceObject,

\_In\_ PKEYBOARD\_INPUT\_DATA InputDataStart,

\_In\_ PKEYBOARD\_INPUT\_DATA InputDataEnd,

\_Inout\_ PULONG InputDataConsumed

);

Так вот, драйвер клавиатуры (kbdclass.sys) получает данные от порта (USB, PS2) через прерывание и записывает их через [WriteFile](https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile), компонент внутри ядра Windows просыпается, считывает их используя API *ReadFile* и добавляет в очередь сообщений с клавиатуры. API для работы с файлом могут использоваться для чтения данных с драйверов. С этого момента начинается обработка данных стеком ввода Windows, об этом в следующей статье.  
  
Если у вас есть ПК с PS2 портом и вы умеете пользоваться WinDbg в режиме ядра, то можете легко найти обработчик прерываний клавиатуры напечатав команду *!idt*, которая выведет на экран всю *таблицу векторов прерываний*. Прерывание вклинивается в ход выполнения программы, слово вектор здесь подразумевает направление, направление исполнения программы. WinDbg был сделан специально для отладки Windows, самая последняя версия называется WinDbgX. Он имеет текстовый интерфейс, который отпугивает людей привыкших к Visual Studio, однако предоставляет гораздо больше возможностей, в частности исполнение скриптов. Прерывание фиолетового порта PS2 выделено красным. Функция которая его обрабатывает называется *I8042KeyboardInterruptService*, которая находится в файле i8042prt.sys.

BOOLEAN

I8042KeyboardInterruptService(

IN PKINTERRUPT Interrupt,

IN PVOID Context

);

Routine Description:

This is the interrupt service routine for the keyboard device when

scan code set 1 is in use.

Arguments:

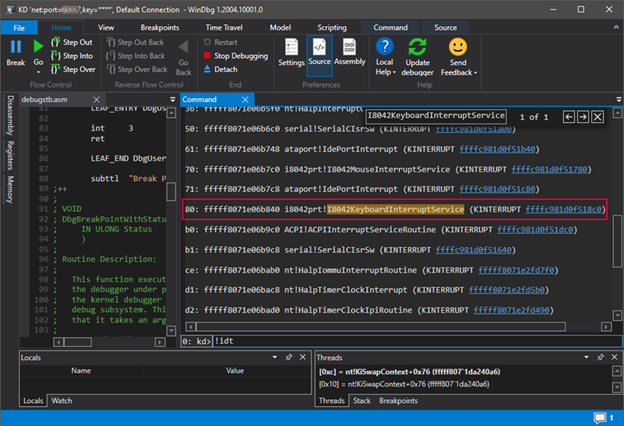
Interrupt - A pointer to the interrupt object for this interrupt.

Context - A pointer to the device object.

Return Value:

Returns TRUE if the interrupt was expected (and therefore processed);

otherwise, FALSE is returned.



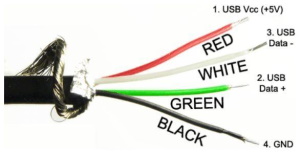
Сейчас возникает вопрос, откуда у обработчика прерываний аргумент? Кто его передаёт? Ведь CPU ничего не знает о нём. Если поставите в неё breakpoint, то удивитесь ещё больше увидев несколько функций выше по стеку:  
  
*0: kd> kC  
# Call Site  
00 i8042prt!I8042KeyboardInterruptService  
01 nt!KiCallInterruptServiceRoutine  
02 nt!KiInterruptSubDispatch  
03 nt!KiInterruptDispatch  
04 nt!KiIdleLoop*  
  
Объяснение здесь простое – это не та функция которая сохранена в регистре *IDT* процессора. То что вы видите на картинке выше на самом деле объекты типа *\_KINTERRUPT*. В таблице прерываний сохранён специальный ассемблерный код (nt!KiIdleLoop), который знает как найти объект описывающий прерывание в памяти. Что же интересного есть в нём?

1. Указатель на объект представляющий драйвер в памяти.
2. Указатель на функцию i8042prt!I8042KeyboardInterruptService, которая и вызывает код считывающий данные из порта PS2 через ассемблерную команду IN AL, 0x60 – сохранить значение из порта номер 0x60 в регистре AL.
3. Функция dispatcher – ей передаётся указатель функцию из пункта №2 и она вызывает её.
4. Состояние регистров CPU. Перед вызовом прерывания состояние CPU будет сохранено сюда, и отсюда же будет восстановлено.
5. Приоритет прерывания. Не тот который определяет контроллер прерываний, а тот который Windows считает нужным. Это IRQL (Interrupt Request Level) – абстракция над IRQ.

Как только обработчик прерываний клавиатуры будет вызван, он уведомит драйвер клавиатуры о полученных данных, после чего будет уведомлено ядро ОС, которое обработав данные отправит их дальше по стеку ввода, где они могут быть доставлены приложению, которое на них отреагирует, или перед этим в обработчик языков (азиатские иероглифы, автокоррекция, автозаполнение).  
Ядро ОС напрямую не взаимодействует с драйвером клавиатуры, для этих целей используется Plug’n’Play Manager. Этот компонент предоставляет API [IoRegisterPlugPlayNotification](https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-ioregisterplugplaynotification), который вызовет предоставленную callback-функцию когда устройство будет добавлено или удалено.

Пару слов о USB

Ознакомление с работой порта USB потребовало бы отдельной статьи описывающей его работу и плюс описание обработки данных HID на Windows. Это очень сильно усложнило бы материал, к тому же уже есть хорошие статьи по теме, поэтому PS2 идеальный пример из-за своей простоты.  
  
USB создавался как универсальный порт для всех устройств, будь то клавиатура, фотоаппарат, сканнер, игровой руль с педалями, принтер и пр. Вдобавок он поддерживает вложенность портов – USB материнки => монитор с USB => клавиатура с USB к которой подключена мышка, флешка и USB-hub к которому подключен жёсткий диск. Взглянув на контакты USB 2.0 вы увидите что они не заточены под передачу каких-то определённых данных, как у PS2. Их всего четыре – витая пара для передачи битов данных, плюс и минус питания.



*Провода кабеля USB 2.0*  
  
USB 3.0 быстрее за счёт дополнительных пяти контактов. Как видите там нету линии CLOCK для синхронизации, поэтому логика передачи данных сложнее. Слева USB 2.0 и справа USB 3.0 для сравнения.

|  |  |
| --- | --- |
| https://habrastorage.org/webt/wa/je/yu/wajeyux1l1hfkyjegako54570bg.jpeg | https://habrastorage.org/webt/3e/uj/k-/3eujk-vjjoia_1ku9izzjtlbtgo.jpeg |

Все данные передаются через протокол *HID (Human Interface Device)*, который описывает форматы, порядок взаимодействия и передачи данных и всё остальное. Стандарт USB 2.0 занимает 650 страниц, документ *HID Class Specification*, описывающий работу устройств (мыши, клавиатуры и пр) – 97 страниц, их рекомендуется изучить если вы работаете с USB.  
  
Первым делом подключенное устройство должно рассказать о себе, для этого оно отправляет несколько структур данных, в которых указывается ID устройства и ID производителя по которым Plug’n’Play manager может найти в реестре информацию, загрузить и соединить драйвера. USB устройства пассивны, т.е. хост должен сам с определённым интервалом проверять наличие данных. Частота опроса и размер пакета данных задаются в одном из дескрипторов устройства USB. Максимальный размер пакета – 64 байта, что для информации о нажатых клавишах более чем достаточно.  
  
В Windows есть встроенная поддержка HID, она не такая простая как связь драйвера порта PS2 с драйвером клавиатуры, потому что драйвер HID должен уметь обрабатывать все поддерживаемые протоколом сценарии. Вне зависимости от провайдера данных — порты PS2, USB или Remote Desktop или виртуальная машина – на самом верху *driver node* будет находится Kbdclass, от которого ядро ОС и будет получать информацию. Уведомление о подсоединении клавиатуры будет обрабатываться через *Plug’n’Play Manager*, так что для ядра Windows не имеет значение какой порт или источник данных от устройства используется.