Немалая часть задач, связанных с обслуживанием локальных вычислительных сетей, представляет собой выполнение рутинных операций, ручная реализация которых может потребовать значительного времени. Вероятно, решения, позволяющие автоматизировать выполнение административных задач, которые могли бы повысить производительность, возникли почти сразу же с появлением профессии системного администратора.  
 Наиболее распространенным средством «экономии времени и избавления от головной боли» стала запись и последовательное пакетное исполнение необходимых операций - исполнение сценариев или скриптов в интерпретаторе команд операционной системы.  
 Попытки улучшить состояние дел в области управления и администрирования Windows с помощью командного интерфейса привели не к адаптации чужеродного для системы языка сценариев или созданию супер-утилиты, работающей в DOS, а к появлению **PowerShell** – новой командной оболочки.

Немалая часть задач, связанных с обслуживанием локальных вычислительных сетей, представляет собой выполнение рутинных операций, ручная реализация которых может потребовать значительного времени. Вероятно, решения, позволяющие автоматизировать выполнение административных задач, которые могли бы повысить производительность, возникли почти сразу же с появлением профессии системного администратора.

Наиболее распространенным средством «экономии времени и избавления от головной боли» стала запись и последовательное пакетное исполнение необходимых операций - исполнение сценариев или скриптов в интерпретаторе команд операционной системы.

В составе MS-DOS и Windows 9x таким интерпретатором, позволяющим выполнять обработку пакетных файлов (bat-файлов), являлся **command.com**, впоследствии (начиная с выхода Windows NT) замененный **cmd.exe**. Позднее появился Windows Script Host.

**Windows Script Host** (WSH; первоначально назывался Windows Scripting Host) – один из элементов Microsoft Windows, как часть операционной системы он начал поставляться, начиная с Windows 98. Позволяет запускать сценарии, написанные с помощью скриптовых языков **VBScript**/**JScript** и, как дополнение, некоторых других. Сценарии, исполняемые в WSH, предоставляют гораздо больше возможностей, чем использование командных (bat- и cmd-) файлов. Исполнение возможно в графической среде (wscript.exe) или в консоли (cscript.exe).

Тем не менее, процесс написания и выполнения сценариев в ОС Windows не развит так хорошо, как, например, в UNIX-системах. Одна из причин этого – сам графический интерфейс ОС Windows, видимо и сделавший ее столь популярной среди обычных, не корпоративных пользователей. Возможность управления некоторыми элементами среды Windows с помощью графического интерфейса не всегда можно реализовать с помощью системных утилит, выполняемых в командной строке. С другой стороны, возможности каких-то системных программ, поставляемых в составе Windows, не всегда представлены в GUI. К тому же интерпретаторы в Windows имеют довольно ограниченный набор команд, «зашитых» в саму оболочку. Windows Script Host не интегрирован с командной строкой и сам по себе представляет потенциальную опасность – его использует достаточно большое количество вредоносных программ.

Попытки улучшить состояние дел в области управления и администрирования Windows с помощью командного интерфейса привели не к адаптации чужеродного для системы языка сценариев или созданию супер-утилиты, работающей в DOS, а к появлению **Windows PowerShell** – новой командной оболочки. По некоторым данным, ее появление связано с использованием платформы .NET при создании командного интерфейса для WMI. В данный момент PowerShell является отдельным приложением, который можно установить на любую систему, использующую платформу **.Net 2.0** (Windows XP, Vista, Server 2003). Начиная с Server 2008, PowerShell будет являться встроенным компонентом Windows-систем. Если же у вас не Server 2008, для знакомства с PowerShell предварительно необходимо будет [его загрузить](http://www.microsoft.com/windowsserver2003/technologies/management/powershell/download.mspx) (возможно, вам понадобится и [установка .NET](http://www.microsoft.com/downloads/details.aspx?familyid=0856eacb-4362-4b0d-8edd-aab15c5e04f5&displaylang=en)).

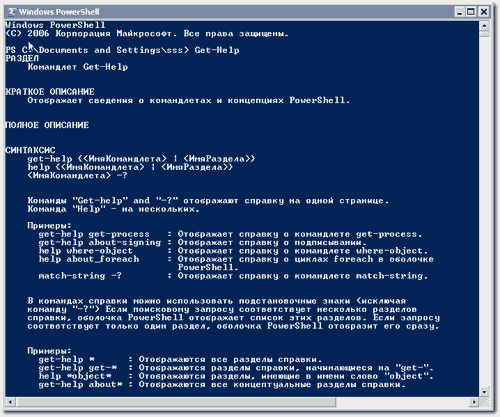
#### Знакомство

Запустив PowerShell, вы не обнаружите поначалу никаких различий между ним и cmd.exe (разве что цвет фона окна у PowerShell по умолчанию - синий). Более того, вскоре вы обнаружите, что операции копирования/вставки в PowerShell реализованы также безобразно, как и в cmd.exe. Но первое впечатление о схожести этих оболочек, скажем так, не совсем соответствует действительности.

То обстоятельство, что работа оболочки PowerShell основана на .NET Framework, является главным ее отличием от предыдущих командных оболочек Windows. PowerShell полностью объектно-ориентирована. Результатом выполнения команды в PowerShell является не некий «текст сам по себе», а объект платформы .NET. Этот объект представляет собой собственно данные и имеет набор присущих ему свойств и методов.

Внутренние команды (точнее, командные структуры) для работы с объектами в PowerShell называются **командлетами**. Для них придумано специальное единообразное именование в виде комбинации действие-цель. Например, для получения данных используется действие “set”, для получения – “get”, для вывода - “out” и т. д. Цель – это тип объекта, к которому будет применено действие. Командлеты можно рассматривать как мини-программы, исполняемые в среде PowerShell. Для повышения функциональности можно создавать собственные командлеты или устанавливать командлеты сторонних разработчиков. Кроме командлетов, PowerShell позволяет выполнять функции, внешние сценарии (хранятся в файлах с расширением ps1) и внешние исполняемые файлы.

В состав PowerShell включена довольно обширная справочная система. Для начала работы с ней можно выполнить команду Get-Help.

  
[Увеличить изображение](http://www.windowsfaq.ru/images/stories/articles/soft/pshell/get-help-b.png)

Для получения детальной справки по какому-либо командлету или разделу основных сведений, необходимо указать его название в качестве параметра команды.

##### Параметры

Строго говоря, следуя духу единообразного именования в PowerShell, все передаваемые командлету имена параметров должны следовать за символом «-». Однако для простоты написания названия некоторых параметров можно опускать. Например, для вызова справки по командлету Get-Content вместо полного указания

* **Get-Help** –name **Get-Content**

можно ввести команду

* **Get-Help** **Get-Content**

Параметр может иметь какое-либо значение (в только что приведенном примере значением параметра name являлось Get-Content) или не иметь его. В этом случае он является аналогом переключателя какой-либо функциональности команды. Например, если необходимо получить полную информацию о командлете Get-Content, введите

* **Get-Help** **Get-Content** –Detailed

##### Конвейер

В PowerShell реализован механизм передачи данных от одного процесса другому или вывод их в файл. Поскольку, как отмечалось выше, PowerShell оперирует не текстом, а объектами, при перенаправлении элементом обмена информации является объект, вместе со своей структурой. Такая возможность позволяет оперировать с объектами - отбирать их по заданному фильтру, сортировать, группировать их и т. д. Для организации такого конвейера (в документации на английском языке используется термин *pipeline* - трубопровод или канал) в тексте сценария используется знак вертикальной черты. При обнаружении такого знака интерпретатор передает объекты от одного командлета другому в качестве входных параметров.

В качестве примера конвейера и возможности получать доступ к свойствам передаваемых по нему объектов, приведем следующую ситуацию. Для проверки, не выполняются ли на компьютере некие подозрительные программы, мы хотим получить список всех запущенных процессов, получить пути и названия файлов, их запускающих, а также посмотреть дату создания таких файлов. В дополнение, отсортируем такой список по дате создания в убывающем порядке и отберем 10 наиболее "свежих" из них. Добавим к выводной информации также время последней модификации файла. Процессы с именами "System" и "Idle" из рассмотрения исключим, так как они не содержат пути к файлам.

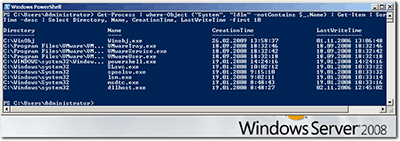
Как говорится, хорошо сформулированный вопрос - уже половина решения. Взгляните:

* **Get-Process** |
* **where-Object** {"System", "Idle" -notContains $\_.Name} |
* **Get-Item** | **Sort** CreationTime -desc |
* **Select** Directory, Name, CreationTime, LastWriteTime *-first* 10

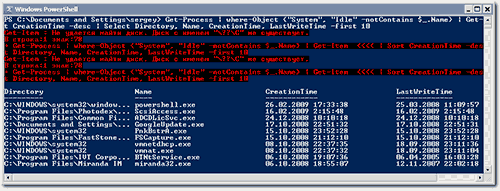
Вводя код, вы всегда можете разбить строку, поставив в месте переноса знак «`» после пробела. Можно даже просто нажать клавишу Enter, не закончив строки. В этом случае PowerShell изменит приглашение на >>, давая пользователю понять, что интерпретатор считает код не завершенным и ожидает окончания его ввода.  
Как и множество других скриптовых языков, PowerShell позволяет использовать переменные. Обозначением переменной служит знак "$". В случае передачи объекта по конвейеру, переменная $\_ указывает на сам передаваемый объект.

Рассмотрим действия кода "по шагам". Сначала мы получаем список процессов с помощью командлета Get-Process. Эти данные передаются по конвейеру далее и фильтруются по условиям, заданным в where-Object (мы откидываем процессы с именами "System" и "Idle").

Следующий элемент конвейера - Get-Item возвращает атрибуты отобранных объектов. Осталось их отсортировать (время создания в убывающем порядке) и выбрать интересующие нас значения (имена папки и исполняемого файла, время создания и последней модификации файла). Последний параметр, -first 10 указывает, что выводиться будут лишь первые 10 элементов из списка объектов. Попробуем выполнить в среде Server 2008:

  
[Увеличить изображение](http://www.windowsfaq.ru/images/stories/articles/soft/pshell/get-process-08-b.png)

Замечательно, то что надо. Однако при попытке выполнить тот же код в среде Windows XP или Server 2003 обнаружилось, что там это выглядит не столь гладко:

  
[Увеличить изображение](http://www.windowsfaq.ru/images/stories/articles/soft/pshell/get-process-xp-b.png)

При просмотре результатов выполнения

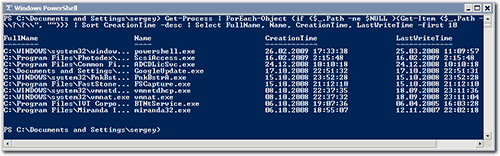
* **Get-Process** | **Select** Path

выяснилось, что пути двух процессов - winlogon и csrss - в Windows XP и Server 2003 PowerShell интерпретирует как \??\C:\WINDOWS\system32\. За разъяснением такого поведения я обратился к [Василию Гусеву](http://xaegr.wordpress.com/), специалисту по PowerShell. Он пояснил, что эти процессы не используют Win32API, и столь разная реакция на них в XP/Vista со стороны .NET, вероятно, вызвана различием платформ этих операционных систем.

Решив, что использовать механизмы обработки ошибок (в части обхода "непонятного" пути с подавлением вывода сообщения об ошибке) или исключения из списка процессов winlogon и csrss в данном случае не годится (возможно, они инфицированы, а дату их модификации в результатах мы уже не увидим), команды были изменены следующим образом:

* **Get-Process** |
* **ForEach-Object** {
* if ($\_.Path -ne $NULL )
* {
* **Get-Item** ($\_.Path -replace "\\\?\?\\", "")
* }
* } | **Sort** CreationTime -desc | **Select** FullName, Name, CreationTime, LastWriteTime *-first* 10

А читатель может получить некоторое представление об использовании в PowerShell условий и регулярных выражений. Небольшие пояснения к коду. На втором этапе конвейера применен командлет ForEach-Object, позволяющий выполнить заданную операцию для каждого объекта из набора, передаваемого на его вход. Как указывалось выше, текущий объект, над которым выполняется операция, представлен переменной $\_. В качестве заданной операции здесь выступает условие вида if (*условие*){*исполняемый код, если условие истинно*}. Так же, как и в cmd.exe, для операторов сравнения используются не символы вида < или >, а аббревиатуры - в данном случае это "не равно"(not equal): -ne. Итак, если путь процесса содержит какое-либо значение (в случае с "System" и "Idle" путь просто отсутствует), с помощью функции replace все символы "\??\" в пути будут удалены (пожалуй, более детально затрагивать вопрос регулярных выражений мы пока не будем), а командлет Get-Item предоставит доступ к свойствам текущего процесса. Ну а далее - все, как и в первом примере. Результат выполнения теперь одинаков:

  
[Увеличить изображение](http://www.windowsfaq.ru/images/stories/articles/soft/pshell/get-process-b.png)

##### Получение сведений об объектах

Возможно, у читателя уже возник вопрос - а как, вообще говоря, можно узнать, какую информацию можно получить в результате выполнения той или иной команды? Какие действия можно произвести с полученными данными? Например, в вышеописанном случае, откуда можно было узнать, что мы сможем получить дату создания файла? Одним из простых способов анализа объекта, возвращаемого командой, является передача этого объекта на вход командлета Get-Member. Этот командлет выводит сведения о типе объекта и всех его элементов. Как правило, объекты имеют большое количество разнообразных свойств и результатом работы Get-Member может стать весьма объемный текст, который не очень удобно просматривать. В этом случае можно либо разделять информацию на части, либо ее отфильтровывать. Пример получения информации об объекте, возвращаемом командлетом Get-Process, просмотр которой можно осуществлять постранично:

* **Get-Process** | **Get-Member** | **Out-Host** *-Paging*

По заполнении страницы, пользователь может выбрать один из вариантов - вывести еще одну страницу, вывести еще одну строку или прекратить вывод данных.

Фильтрация данных выполняется при помощи параметра MemberType, определяющего, сведения какого рода должны быть выведены. Например, команда

* **Get-Process** | **Get-Member** *-MemberType* Properties

выведет лишь свойства объекта, а

* **Get-Process** | **Get-Member** *-MemberType* Methods

- лишь его методы. Еще один способ посмотреть свойства объекта - присвоить переменной объект, затем набрать в консоли имя переменной, поставить точку и нажать клавишу Tab. С каждым нажатием клавиши PowerShell будет перебирать и подставлять методы и свойства объекта. Перебор в обратную сторону возможен с помощью сочетания клавиш Shift+Tab.

##### Безопасность

Как уже отмечалось, использование сценариев **VBScript**/**JScript** представляет потенциальную опасность для системы - для их исполнения достаточно щелкнуть по значку мышью. Опасность еще более возрастает, если пользователь вошел под учетной записью, входящей в группу администраторов. В PowerShell скрипт с расширением ps1 невозможно запустить на исполнение с помощью мыши - в системе такой файл будет открыт не в командной оболочке, а в Блокноте. Для запуска сценария необходимо запустить саму оболочку PowerShell, ввести имя файла и нажать клавишу Enter.

В новой оболочке так же невозможна подмена команд. Суть этого приема, применяемого злоумышленниками, заключается в следующем. Обычно у пользователя, не имеющего прав администратора, есть некоторые папки с разрешениями на запись и выполнение файлов. Характерный пример - папка C:\Documents and Settings\*имя\_пользователя*. Вредоносная программа создает в такой папке исполняемый файл с именем, совпадающим с именем команды оболочки или именем исполняемой системной программы. К примеру, я создал в "своей" папке документов ipconfig.vbs, выводящий простое сообщение. Теперь, если, запустив cmd.exe, и находясь в своей папке, я попытаюсь выполнить команду Windows **ipconfig**, то получу вот такой результат:

  
[Увеличить изображение](http://www.windowsfaq.ru/images/stories/articles/soft/pshell/joke-b.png)

Для полноты иллюстрации можно поместить в папку с документами и исполняемый файл, переименованный в нашем случае в ipconfig.exe. Тогда даже при вызове с указанием расширения будет запускаться файл из текущей папки, а не из \system32. С PowerShell такой фокус не пройдет - для вызова скрипта, путь к которому не совпадает с путями, заданными в системной переменной %Path, необходимо явно указать его расположение. Даже в том случае, когда скрипт расположен в папке, являющейся для оболочки текущей, необходимо указать путь в таком виде: .\имя\_файла. Точка с обратным слешем указывают интерпретатору на текущую папку.

Еще одним механизмом обеспечения безопасности является политика выполнения сценариев. Изначально оболочка настроена так, что даже при правильном вызове сценария его выполнение будет запрещено, а пользователь получит соответствующее сообщение. Политика выполнения может переключаться в один из четырех режимов:

* Restricted - настройка по умолчанию, запуск любых сценариев запрещен
* AllSigned - разрешен запуск сценариев, имеющих цифровую подпись надежного издателя; сценарии, созданные пользователем, также должны быть заверены центром сертификации
* RemoteSigned - разрешен запуск сценариев, если они не являются доверенными, но созданы локальным пользователем; сценарии, загруженные из Интернета, не имеющие подписи, не исполняются
* Unrestricted - разрешен запуск любых сценариев

Текущий режим политики можно узнать, выполнив команду Get-ExecutionPolicy в оболочке. Для изменения режима выполните команду Set-ExecutionPolicy с необходимым названием политики в качестве параметра.

Также к инструментам, помогающим повысить безопасность при работе с PowerShell, я бы отнес параметры команд из разряда "а что будет, если...". Их два - whatif и confirm. Первый позволяет определить, какое действие и с каким объектом будет произведено, однако само действие реализовано не будет. Что-то вроде моделирования. Второй перед выполнением действия будет запрашивать подтверждения пользователя, и в случае утвердительного ответа - запускать необходимую команду фактически. То есть, такой вид подстраховки.

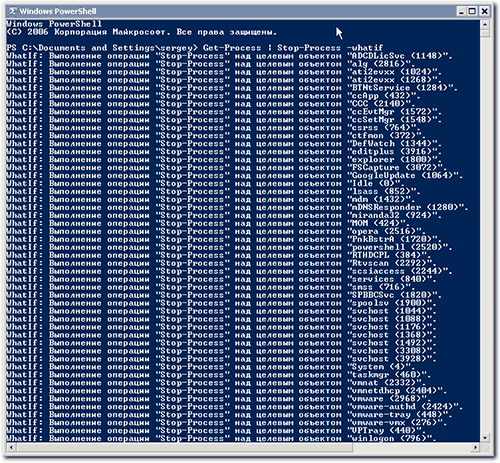
Приведу, пожалуй, наиболее яркий и забавный пример использования этих параметров. Если пользователь попытается выполнить команду

* **Get-Process** | **Stop-Process**

то через несколько секунд его будет ждать синий экран со STOP-ом. PowerShell, как и следует из текста команды, последовательно начнет "прибивать" все запущенные в системе процессы, что и приведет к ее критическому останову. Если же запустить

* **Get-Process** | **Stop-Process** *-whatif*

ничего страшного не произойдет - просто PowerShell покажет, что бы он сделал, если бы команда выполнялась без ключа -whatif:

  
[Увеличить изображение](http://www.windowsfaq.ru/images/stories/articles/soft/pshell/stop-process-b.png)

##### Псевдонимы

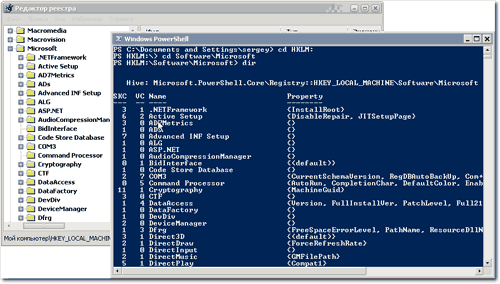
Оболочка имеет встроенный механизм псевдонимов команд. С одной стороны, псевдонимы используются для упрощения ввода команд. Как правило, в этом случае в качестве псевдонима используется сокращенное наименование командлета (например, gc для Get-Content или fl для Format-List). С другой стороны, этот механизм обеспечивает совместимость интерфейсов различных командных интерпретаторов. К примеру, имея опыт работы с cmd.exe, вы привыкли выводить содержимое папки с помощью команды dir. Выполнение этой команды в PowerShell приведет к тому же результату, хотя на самом деле оболочка вместо псевдонима dir будет выполнять командлет Get-ChildItem. Список всех доступных псевдонимов можно получить с помощью команды Get-Alias. Пользователь может создавать собственные псевдонимы, используя команду Set-Alias.

##### Диски PowerShell

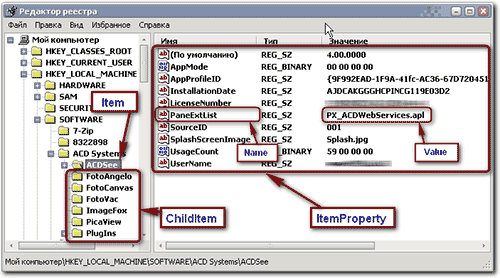
Так же, как Windows оперирует с данными, используя файловую систему, оболочка PowerShell работает с хранилищами данных, представленных в виде дисков. Физические диски системы являются не единственным встроенным в оболочку видом хранилищ, с которыми обеспечивается взаимодействие. Пользователь может работать с реестром, встроенными переменными и переменными среды, хранилищами сертификатов точно так же, как и с обычными дисками, папками и файлами. Реализация такого взаимодействия и обеспечение абстракций, позволяющих пользователю применять одинаковые команды и методы к различным хранилищам данных, выполняется *провайдерами* - программами .NET.

Список провайдеров, доступных в данный момент оболочке, можно получить командой Get-PSProvider. Изначально в PowerShell присутствуют следующие "диски" - псевдонимы (Alias), переменные среды (Env), физические диски системы (C, D, и т. д.), функции, системный реестр, внутренние переменные (Variable) и хранилище сертификатов.

Вот пример чтения содержимого ветки реестра HKLM\Software\Microsoft

  
[Увеличить изображение](http://www.windowsfaq.ru/images/stories/articles/soft/pshell/hklm-b.png)

Как видно, использованы те же команды, что для получения сведений о файловой системе. Но структура получаемых данных, естественно, различна. Кроме названия и свойств для каждого элемента выводится номер подраздела (SKC) и номер записи (VC). С помощью PowerShell пользователь может просматривать сведения о реестре, добавлять, удалять и модифицировать ключи. Позволю привести себе что-то вроде шпаргалки по работе с элементами реестра:

  
[Увеличить изображение](http://www.windowsfaq.ru/images/stories/articles/soft/pshell/registry-b.png)

И код для примера выполнения различных манипуляций с ключами реестра и их параметрами:

* # Создаем новый подраздел с именем valks в ветке HKEY\_CURRENT\_USER\Software
* **New-Item** *-path* HKCU:\Software\valks
* # Добавляем в созданный раздел новый строковый параметр с именем Param1 и значением StringValue
* **New-ItemProperty** *-path* HKCU:\Software\valks *-name* Param1 *-propertyType* String *-value* StringValue
* # Создадим подраздел SubFolder
* **New-Item** *-path* HKCU:\Software\valks\SubFolder
* # Добавляем еще один параметр - Param2 типа DWord и значением 12
* **New-ItemProperty** *-path* HKCU:\Software\valks *-name* Param2 *-propertyType* DWord *-value* 12
* # Получаем список всех параметров
* **Get-ItemProperty** HKCU:\Software\valks
* # Получаем значение параметра Param2
* **Get-ItemProperty** HKCU:\Software\valks | **Format-list** Param2
* # Или можем считать раздел в переменную $key
* $key = **Get-ItemProperty** HKCU:\Software\valks
* # И вывести значение нужного параметра
* **Write-Host** "Значение параметра Param2: " $key.Param2
* # Изменим значение параметра Param2 на 193
* **Set-ItemProperty** HKCU:\Software\valks *-name* Param2 *-value* 193
* # Изменим название параметра Param1 на Параметр1
* **Rename-ItemProperty** *-path* HKCU:\Software\valks *-name* Param1 *-newname* Параметр1
* # Удаляем Параметр1
* **Remove-ItemProperty** HKCU:\Software\valks *-name* Параметр1
* # Удаляем весь подраздел valks
* **Remove-Item** HKCU:\Software\valks

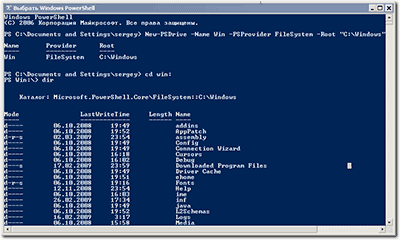
Вот еще небольшой пример в виде функции, которая осуществляет поиск программ, автоматически загружающихся при старте системы. Область поиска определяется массивом, включающим в себя некоторые известные точки автозапуска в реестре. Код содержит комментарии, надеюсь, они пояснят суть работы.

* function GetAutoexec ($hives) {
* # Если функции не передается входной массив ключей реестра,
* # используем этот:
* $hives = "HKCU:\Software\Microsoft\Windows\CurrentVersion\Run", `
* "HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run", `
* "HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\policies\Explorer\Run"
* # Выодим заголовок и переносим строку
* **Write-Host** "Список автозагрузки**`n**"
* # Начинаем перебирать элементы массива - ветви реестра
* Foreach ($hive in $hives){
* # Выводим название ветви зеленым цветом
* **Write-Host** "Ветвь $hive" *-ForegroundColor* Green
* # Проверяем, существует ли ветвь в реестре
* if (**Test-Path** $hive){
* # Получаем очередной ключ реестра
* [Microsoft.Win32.RegistryKey]$param = **Get-Item** $hive
* # для каждого ключа...
* foreach ($p in $param){
* # ...получаем список его параметров
* foreach ($key in $p.getvalueNames()){
* # выводим название параметра и его значение
* "Загрузка $key из " + $p.GetValue($key)
* }
* }
* }
* # переносим строку
* **Write-Host** "**`n**"
* }
* }
* # осуществляем вызов самой функции
* GetAutoexec

Пользователь может создавать собственные диски, используя существующие провайдеры. Вот пример создания диска PowerShell с именем Win, содержимое которого будет являться корневой папкой C:\Windows:

* **New-PSDrive** *-Name* Win –PSProvider FileSystem *-Root* "C:\Windows"

После создания диска PowerShell к нему можно обращаться точно так же , как к обычному диску системы.

  
[Увеличить изображение](http://www.windowsfaq.ru/images/stories/articles/soft/pshell/disk_posh-b.png)

Однако необходимо знать, что по завершении сеанса работы с PowerShell он будет автоматически удален. Так же, как и псевдонимы, функции и переменные, созданные пользователем в течение сеанса. Для того, чтобы сохранить перечисленные изменения, необходимо создать профиль PowerShell.

##### Профили PowerShell

Профиль - это файл с расширением ps1. Фактически, это тот же скрипт, выполняемый оболочкой при ее запуске. Профили в оболочке не создаются автоматически - они должны быть созданы пользователем самостоятельно. Созданные профили будут загружаться при каждом запуске PowerShell, если политикой выполнения разрешено загружать конфигурационные файлы. Возможна обработка до четырех различных профилей. Расположение файлов в порядке последовательности их загрузки:

* %windir%\system32\WindowsPowerShell\v1.0\profile.ps1 - профиль, применяемый ко всем пользователям и оболочкам
* %windir%\system32\WindowsPowerShell\v1.0\ Microsoft.PowerShell\_profile.ps1 - профиль, применяемый ко всем пользователям только оболочки PowerShell
* %UserProfile%\My Documents\WindowsPowerShell\profile.ps1 - применяется для текущего пользователя во всех оболочках
* %UserProfile%\My Documents\WindowsPowerShell\Microsoft.PowerShell\_profile.ps1 - применяется для текущего пользователя только в оболочке PowerShell

Под различными оболочками здесь нужно учитывать то обстоятельство, что PowerShell может оказаться не единственным приложением, использующим файлы профилей ps1. Некоторые интегрированные среды разработки (IDE) также могут использовать их. Один из характерных примеров - инструмент [PowerGUI](http://powergui.org/index.jspa), разработанный Quest Software, предоставляющий средства графического интерфейса для работы с PowerShell.

Путь к профилю текущего пользователя только оболочки PowerShell хранится во встроенной переменной $profile. Для его создания выполните команду

* **New-Item** *-Path* $profile *-ItemType* file *-force*

После создания его можно открыть любым текстовым редактором, например Блокнотом:

* notepad $profile

и сделать в нем необходимые изменения. Не забудьте проверить, разрешен ли политикой выполнения запуск скриптов.

##### Работа с объектами WMI

**WMI** (Windows Management Interface, интерфейс управления Windows) — набор интерфейсов для управления ОС Windows с помощью специальных компонентов. Возможно управление локальным компьютером, и находящимся в сети. WMI - разновидность Web-Based Enterprise Management (WBEM) и Common Information Model (CIM), разработанная Microsoft. Входит в состав Windows Vista, Windows Server 2003, Windows XP, Windows Me и Windows 2000. Для Windows 95 и Windows 98 доступна в виде отдельно устанавливаемого компонента. Поддерживает использование скриптовых языков, таких как VBScript или Windows PowerShell для управления персональными компьютерами и серверами, работающими под управлением Microsoft Windows.

Объекты WMI являются для PowerShell вполне "родными". Достаточно выполнить команду

* **Get-WmiObject** *-List*

чтобы увидеть большое количество классов, обеспечивающих доступ к объектам WMI в оболочке. В случае подключения к WMI на удаленном компьютере, состав классов будет зависеть от ОС и установленных на нем расширений WMI. Для получения сведений о доступных классах на удаленной машине, необходимо указать его IP-адрес или имя в качестве параметра:

* **Get-WmiObject** *-List* *-ComputerName* Server

Для успешного подключения на удаленном компьютере должен быть запущен интерфейс WMI, а используемая учетная запись должна входить в группу локальных администраторов.

Если не использовать специальное указание, некоторые сведения не выводятся, видимо из соображений "не захламлять экран". Для получения более детальной информации можно воспользоваться командами форматирования и отбора данных.

* **PS** C:\Documents and Settings\Администратор> **Get-WmiObject** *-Class* Win32\_OperatingSystem
* SystemDirectory : C:\WINDOWS\system32
* Organization : Nrest
* BuildNumber : 3790
* RegisteredUser : Сергей
* SerialNumber : 69889-650-3137304-45684
* Version : 5.2.3790
* **PS** C:\Documents and Settings\Администратор> **Get-WmiObject** *-Class* Win32\_OperatingSystem |
* **Format-List** Locale, Version, CurrentTimeZone, OSLanguage, InstallDate
* Locale : 0419
* Version : 5.2.3790
* CurrentTimeZone : 180
* OSLanguage : 1049
* InstallDate : 20081022233211.000000+240

А вот небольшой пример опроса всех компьютеров в локальной сети с адресом 192.168.1.0 и маской подсети 255.255.255.0:

* 1..254|
* **ForEach-Object** *-Process*
* {
* **Get-WmiObject** *-Class* Win32\_PingStatus
* -Filter ("Address='192.168.1." + $\_ + "'")
* *-ComputerName* .
* } |
* **Select-Object** *-Property* Address,ResponseTime,StatusCode

В первом элементе конвейера генерируется массив чисел от 1 до 254. На втором этапе каждое число из массива подставляется в IP-адрес, который будет пинговаться при помощи средств WMI. Результаты будут выводиться в таблицу с тремя столбцами - адрес хоста, время отклика и статус ответа. В случае ответа хоста возвращается статус с кодом "0".

##### Работа с COM-объектами

Платформа .NET имеет встроенные средства, позволяющие ей работать с COM-компонентами. Эта возможность позволяет управлять работой различных приложений, поддерживающих COM. В качестве примера покажем функцию для автоматизации работы с Internet Explorer. Мы откроем IE и перейдем по адресу [http://windowsfaq.ru](http://windowsfaq.ru/). Если в качестве параметра функции будет передана строка, будем искать ее с помощью поискового механизма самого сайта, если параметр будет отсутствовать - будем искать слово windows. Вот код с комментариями:

* # Объявляем функцию, устанавливаем параметр по умолчанию - windows
* function WinfaqSearch ([string]$str = "windows") {
* # Создаем COM-объект - Internet Explorer
* $ie = **New-Object** *-Comobject* InternetExplorer.application
* # Указываем браузеру адрес перехода
* $ie.Navigate("http://windowsfaq.ru")
* # Делаем запущенный экземпляр IE видимым
* $ie.Visible = $True
* # На всякий случай, ждем загрузки страницы 5 секунд
* **Start-Sleep** 5
* # Получаем текст веб-страницы
* $doc=$ie.document
* # Ищем поле ввода формы поиска на странице
* $text = $doc.getElementById("mod\_search\_searchword")
* # Вводим в него нужное значение
* $text.value = $str
* # Получаем саму форму, отвечающую за поиск
* $forms = @($ie.Document.forms | where {$\_.action -match "index.php\?option=com\_search&Itemid=5"})
* # Отправляем в нее запрос
* $forms[0].Submit()
* # Спрашиваем, надо ли закрыть экземпляр IE
* if (($resp = **Read-Host** "Закрыть Internet Explorer ? [Y]Да/[N]Нет") -eq "y"){
* if ( $ie.Visible -eq $true ){
* $ie.Quit()
* }
* **Remove-Variable** ie
* }
* }
* WinfaqSearch

Командлеты AD Powershell доступны, начиная с Windows Server 2008 R2. Чтобы использовать командлеты AD Powershell, в домене должен быть хотя бы один контроллер на основе Windows Server 2008 R2.

**Установка модуля AD Powershell:**

В Windows Server 2008 R2, откройте окно консоли PowerShell (powershell.exe) и выполните команды:

PS C:\> import-module servermanager   
PS C:\> Add-WindowsFeature -Name "RSAT-AD-PowerShell" –IncludeAllSubFeature

Примечание: модуль AD Powershell по умолчанию уже установлен на котроллере домена.

**Загрузка модуля AD Powershell:**

Откройте окно консоли PowerShell и наберите:

PS C:\> import-module activedirectory

Если компьютер включен в домен, по умолчанию создается диск по имени AD: Вы можете перейти на этот диск командой CD и использовать привычные команды работы с файловой системой для навигации по этому диску. Пути представлены в формате X500.

PS C:\> cd AD:  
PS AD:\>   
PS AD:\> dir  
…  
PS AD:\> cd "DC=fabrikam,DC=com"  
PS AD:\DC=fabrikam,DC=com> md "OU=myNewOU"  
…   
PS AD:\DC=fabrikam,DC=com> del "OU=myNewOU"

Если Вы хотите создать новый диск, подключенный к другому домену или лесу или использовать более удобные пути в каноническом формате, введите:

PS C:\> New-PSDrive -PSProvider ActiveDirectory -Server "contoso.fabrikam.com" -Credential "Contoso\Administrator" -Root "" -Name Contoso -FormatType Canonical  
…   
PS C:\> cd Contoso:  
PS Contoso:\> dir | ft CanonicalName  
…   
PS Contoso:\> cd "contoso.fabrikam.com/"

**Как получить список командлетов, описание и примеры:**

PowerShell использует имена командлетов, состоящие из глагола и существительного. Например:

New-ADGroup  
Get-ADDomain

Чтобы получить список командлетов, посвященных AD, введите:

PS AD:\> get-help \*-AD\*  
PS AD:\> get-help New-AD\* *## просматриваем все командлеты, создающие новые объекты AD*

Чтобы получить информацию об определенном командлете и прочитать примеры применения, введите:

PS AD:\> get-help set-aduser -detailed  
PS AD:\> get-help get-aduser –examples

На заметку: Вы можете дополнять имена командлетов PowerShell или параметров с помощью кнопки < Tab>. Например, после ввода глагола из названия командлета, нажимайте кнопку <Tab> для циклического просмотра существительных, которые допустимы для этого глагола, чтобы составить название командлета.

**Типовые применения:**

Ниже приведены несколько примеров решения типовых административных задач с помощью командлетов AD:

PS C:\> New-ADUser –Name "John Smith" –SamAccountName JohnS –DisplayName "John Smith" –Title "Account Manager" –Enabled $true –ChangePasswordAtLogon $true -AccountPassword (ConvertTo-SecureString "[p@ssw0rd](mailto:p@ssw0rd)" -AsPlainText -force) -PassThru

PS C:\> New-ADGroup -Name "Account Managers" -SamAccountName AcctMgrs -GroupScope Global -GroupCategory Security -Description "Account Managers Group" –PassThru

PS C:\> New-ADOrganizationalUnit -Name AccountsDepartment -ProtectedFromAccidentalDeletion $true -PassThru

PS C:\> Get-ADUser -Filter { name –like "john\*" } *## Получаем объекты всех пользователей, чьи имена начинаются с «John»*

PS C:\> Add-ADGroupMember -Identity AcctMgrs -Members JohnS

PS C:\> Get-ADGroupMember -Identity AcctMgrs

PS C:\> Get-ADPrincipalGroupMembership -Identity JohnS *## Получаем список всех групп, в которые непосредственно включен указанный пользователь*

PS C:\> Get-ADAccountAuthorizationGroup -Identity JohnS *## Получаем список всех групп, чьи идентификаторы (SID) включены в билет доступа (token) аккаунта*

PS C:\> Unlock-ADAccount -Identity JohnS

PS C:\> Get-ADForest -Current LocalComputer

PS C:\> Get-ADDomain -Current LoggedOnUser

PS C:\> Get-ADDomainController -Filter { name -like "\*" } *## Получаем все контроллеры домена текущего домена*