Stilecht eingerichtet

Die PowerShell mit Profil-Skripten individualisieren

Die PowerShell ist nicht nur ein mächtiges Werkzeug zum Einrichten und Konfigurieren des Rechners, auch sie selbst hat zahlreiche Schrauben, an denen man drehen kann. Das passende Werkzeug dafür ist ihre eigene Skriptsprache.

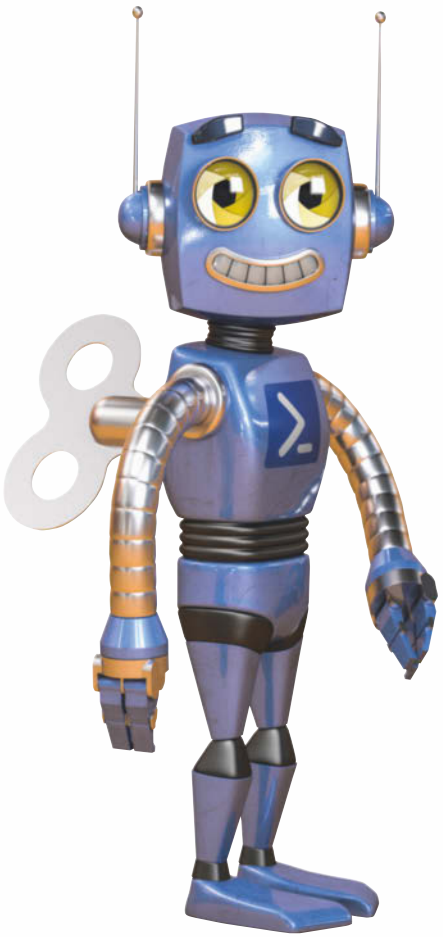
Von Hajo Schulz

Sowohl in der Systemsteuerung als auch in der Einstellungen-App von Windows sucht man vergeblich nach einer Seite, auf der sich die PowerShell konfigurieren ließe. Trotzdem gibt es zahlreiche Aspekte, in denen man sie an eigene Wünsche und Vorlieben anpassen kann. Das Mittel, mit dem man an diese Einstellungen herankommt, sind nicht etwa Registry-Einträge oder Konfigurationsdateien, sondern die PowerShell selbst: Sie kennt diverse Befehle, mit denen sich die Feinheiten der Arbeitsumgebung definieren lassen.

Die meisten dieser Befehle wirken sich nur auf die aktuelle Sitzung aus; mit dem Schließen des PowerShell-Fensters, in dem man sie ausgeführt hat, sind sie Geschichte. Für Optionen, die man bei jedem Aufruf der PowerShell wieder wie beim letzten Mal vorfinden möchte, ist das auf den ersten Blick ziemlich fatal. Bei näherem Hinsehen stellt sich aber heraus, dass die PowerShell einen außerordentlich flexiblen und mächtigen Mechanismus enthält, um sie trotzdem dauerhaft an eigene Vorlieben anzupassen: Die Konfigurationsbefehle lassen sich nämlich in Skripte verpacken, die die PowerShell bei jedem Start einer Sitzung automatisch abarbeitet.

Jedes Mal, wenn die PowerShell startet, sucht sie nach insgesamt vier Skriptdateien, den sogenannten Profilen. So sie denn vorhanden sind, lädt sie sie automatisch und führt sie aus. Je zwei dieser Skripte gelten systemweit für alle Anwender; wenn es sie gibt, sind sie im Installationsordner der PowerShell gespeichert (normalerweise unter `$env:windir\System32\WindowsPowerShell\v1.0`) und können nur mit Administrator-Rechten bearbeitet werden. Die beiden anderen gehören dem gerade angemeldeten Benutzer und liegen in dessen Dokumente-Ordner (genauer: in `$env:userprofile\Documents\WindowsPowerShell`); er kann sie nach Belieben anlegen, ändern oder löschen.

Damit, dass die PowerShell für das System und für den Benutzer je zwei Profil-Skripte ausführt, hat es folgende Bewandtnis: Jeweils eines davon ist dazu da, Befehle aufzunehmen, die bei wirklich jedem PowerShell-Start abgearbeitet werden sollen. Das andere gibt es wiederum mehrfach, und zwar eines für jeden sogenannten Host, in dem die PowerShell laufen kann. Ein frisch installiertes Windows 10 besitzt zwei solcher Hosts: die normalerweise zum Arbeiten mit der PowerShell benutzte Textkonsole sowie das Integrated Scripting Environment (ISE) zum Bearbeiten und Austesten von Skripten. Weitere PowerShell-Hosts können dazukommen, etwa wenn man sich eine alternative Skript-Entwicklungsumgebung installiert.



Die konkreten Dateinamen der Profile auf Ihrem System liefert der Befehl

```
$profile | fl * -Force
```

Dabei stecken in den Einträgen, deren Bezeichner mit „AllUsers“ beginnen, die System-Profile, in denen mit „CurrentUser“ Ihre eigenen. Die auf „AllHosts“ endenden Einträge bezeichnen die allgemeinen Profile, die mit „CurrentHost“ am Ende jene für die aktuelle Ausführungsumgebung. Letztere werden sich unterscheiden, je nachdem, ob Sie sie in der Textkonsole oder im ISE abrufen. Die Variable `$profile` selbst enthält immer dasselbe wie `$profile.CurrentUserCurrentHost`.

Die allermeisten der im Folgenden vorgestellten Ideen, mit denen Sie sich Ihr maßgeschneidertes Profil zusammenstellen können, gehören am sinnvollsten in die unter `$profile.CurrentUserAllHosts` bezeichnete Datei. Wo es um Spezifika für die Textkonsole oder das ISE geht, werden wir explizit darauf hinweisen.

Ein Tipp noch für Experimentierfreudige: Sollten Sie sich beim Basteln an den Profilen mal so verhaspeln, dass die PowerShell gar nicht mehr startet, können Sie sie immer noch mit dem Aufruf `powershell -noprofile` laden; sie überspringt dann das Abarbeiten der Skripte. Diese Option kann auch sinnvoll sein, um beim automatischen Ausführen von PowerShell-

Aktionen etwa über den Taskplaner Nebenwirkungen der Profile auszuschließen.

Funktionen und Aliase

Für Tippfaule gibt es in der PowerShell die Möglichkeit, sogenannte Aliase zu definieren. Etliche dieser Abkürzungen bringt eine Standard-Installation bereits mit, beispielsweise kann man statt `Get-ChildItem` je nach Gusto einfach `dir` oder `ls` schreiben. Mit dem Befehl `Set-Alias` lassen sich der Liste eigene Aliase hinzufügen:

```
Set-Alias -Name grep `
          -Value Select-String
```

Der `Name` und damit die Abkürzung muss mit einem Buchstaben beginnen und ausschließlich aus Ziffern und Buchstaben bestehen. Als `Value` kommen nicht nur PowerShell-eigene Cmdlets infrage, sondern auch externe Skripte, Kommandozeilenprogramme oder gar normale Windows-Anwendungen. So kann man sich mit einer Definition wie

```
Set-Alias edit 'C:\Program Files\
              ↳Notepad++\notepad++.exe'
```

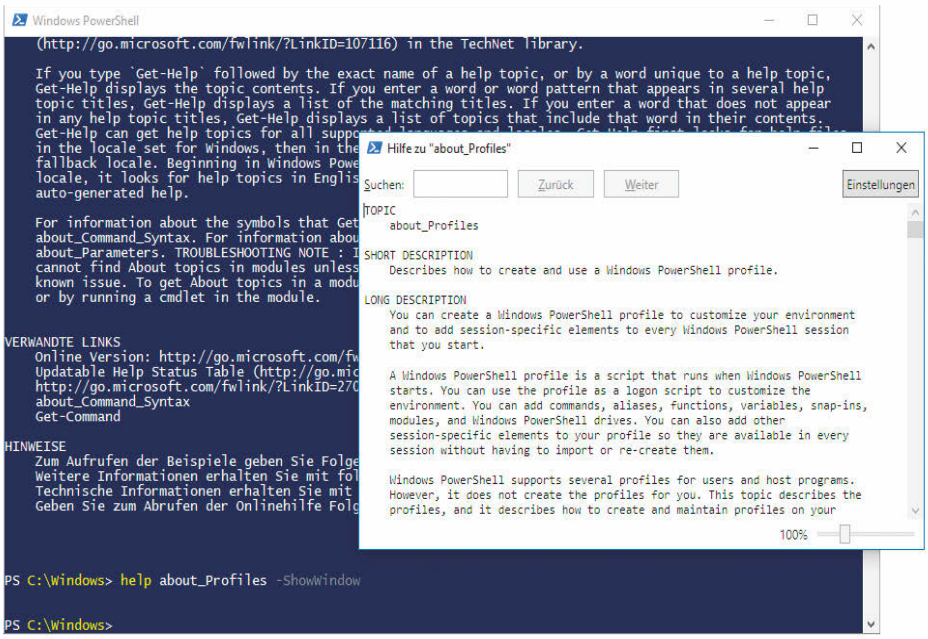
den Aufruf des eigenen Lieblings-Texteditors ein wenig einfacher machen: Anschließend funktioniert sowohl ein nacktes `edit` als auch ein Aufruf mit Parameter wie `edit liste.txt`.

Aliase stoßen an ihre Grenzen, wenn man nicht nur einen Befehl abkürzen, sondern auch gleich ein oder mehrere Argumente vordefinieren will. So ist es beispielsweise nicht möglich, einen Alias – vielleicht mit dem Namen `cdw` – für den Befehl `chdir C:\Windows` zu definieren. Auf die Abkürzung muss man trotzdem nicht verzichten: Statt eines Alias definiert man einfach eine Funktion mit dem gewünschten Namen:

```
function cdw() {
  Set-Location C:\Windows
}
```

Solche Schnellbefehle in Form simpler Funktionen können den Umgang mit der PowerShell deutlich beschleunigen und komfortabler machen. Dabei sind sie nicht auf eine Zeile beschränkt und können auch ihrerseits Parameter verwenden. Die Funktion

```
function mcd($dir) {
  $new = New-Item $dir -Type Directory
  Set-Location $new
}
```



Mit der Option -ShowWindow kann man den help-Befehl anweisen, für seine Ausgabe ein eigenes Fenster zu öffnen. Ein passender Eintrag im Profil-Skript setzt die Option ein für alle Mal.

kombiniert beispielsweise die Befehle `md` (`New-Item`) und `cd` (`Set-Location`), indem sie zuerst einen neuen Ordner anlegt und dann sofort in ihn hineinnavigiert.

Der Code-Menge und der Komplexität solcher Funktionen sind im Prinzip keine Grenzen gesetzt. Allerdings lehrt die Erfahrung, dass sie umso seltener tatsächlich benutzt werden, je aufwendiger sie programmiert sind. Um nicht bei jedem PowerShell-Start Ladezeit und Speicherplatz zu verschwenden, empfiehlt es sich daher, sie ab einer Größe von etwa 20 Zeilen nicht im Profil zu definieren, sondern in eigene Skripte auszulagern. Damit man die bei Bedarf trotzdem möglichst einfach aufrufen kann, sollten sie in einem Ordner liegen, der in der Umgebungsvariablen `$env:Path` verzeichnet ist. Als Speicherort bietet sich der Ordner an, in dem auch die benutzerspezifischen Profile liegen. Folgender Code-Schnipsel fügt ihn dem `Path` hinzu:

```
$myDir = Split-Path $profile
if($Env:Path -notlike "*$myDir*") {
  $Env:Path = "$myDir;$Env:Path"
}
```

`Split-Path` nimmt einen Datei- oder Ordnernamen entgegen und liefert ohne weitere Optionen den Namen des Verzeichnisses, in dem das dazugehörige Dateisystemobjekt gespeichert ist. Die Prüfung `$Env:Path -notlike "*$myDir*"` stellt sicher, dass `$Env:Path` den gewünschten Ordner nicht bereits enthält.

Standard-Parameter

Wer die PowerShell regelmäßig benutzt, dem wird früher oder später auffallen, dass er bei bestimmten Befehlen immer wieder dieselben Argumente oder Optionen eintippt: Kein `Send-MailMessage` ohne die immer wieder gleiche Absenderadresse, `help` liefert nur mit der Option `-Detailed` brauchbare Ergebnisse und so weiter. Diese Tipparbeit kann man sich sparen, indem man in sein Profil Zeilen nach diesem Muster einfügt:

```
$PSDefaultParameterValues[
              ↳'Send-MailMessage:From']
                 ↳ = 'admin@example.com'
```

`$PSDefaultParameterValues` ist eine Systemvariable, die eine Hashtable enthält. Deren Schlüssel müssen nach dem Muster `'Befehl:Parameter'` aufgebaut sein, die dazugehörigen Werte enthalten den gewünschten Inhalt für das jeweilige Argument. Bei Schaltern ist der Wert `$true`, Sie können also beispielsweise mit

```
$PSDefaultParameterValues[
          ↳'Get-Help:ShowWindow'] = $true
```

dafür sorgen, dass der `help`-Befehl seine Weisheiten grundsätzlich in einem eigenen Fenster anzeigt. Wenn Sie einen Parameter bei einem Aufruf wie gewohnt

explizit angeben, überstimmt das den vordefinierten Wert. Wollen Sie einen per `$PSDefaultParameterValues` gesetzten Schalter ausnahmsweise mal außer Kraft setzen, weisen Sie ihm `$false` zu:

```
help dir -ShowWindow:$false
```

Mehr Drive!

Aus der Sicht der Windows PowerShell ist die Registry so etwas Ähnliches wie ein Dateisystem: Man kann dort mit `cd` navigieren, mit `md` und `rd` Schlüssel anlegen und löschen und sich mit `dir` deren Inhalte anzeigen lassen [1]. Zu diesem Zweck definiert die PowerShell die virtuellen Laufwerke HKLM und HKCU. Die wichtigen Äste unter `HKEY_CLASSES_ROOT` und `HKEY_USERS` lassen sich auf diesem Weg aber unverständlicherweise nicht ansprechen. Dieses Problem können Sie mit folgenden Zeilen in Ihrem Profil ausmerzen:

```
New-PSDrive -Name HKCR `
            -PSProvider Registry `
            -Root HKEY_CLASSES_ROOT `
  | Out-Null
```

Das Cmdlet `New-PSDrive` richtet ein neues virtuelles Laufwerk ein und braucht dazu als Argumente den gewünschten Namen, den dazugehörigen Provider und die Angabe, wo sich der Wurzelordner befindet. Die Namen der verfügbaren Provider liefert `Get-PSProvider`. Der Einstiegspunkt HKU unter `HKEY_USERS` lässt sich nach demselben Muster einrichten.

Auf die gleiche Art schafft man sich auch Abkürzungen zu häufig benutzten Ordnern im Dateisystem: So legt etwa der Befehl

```
New-PSDrive -Name W `
  -PSProvider FileSystem `
  -Root 'D:\Projekte\Weltherrschaft' `
  | Out-Null
```

ein neues Laufwerk W: an, dessen Wurzelverzeichnis am angegebenen Ort liegt und auf das man mit den üblichen Dateisystem-Befehlen zugreifen kann. Als Wurzelordner kommen hier nicht nur Verzeichnisse auf lokalen Datenträgern infrage, sondern auch Ressourcen im Netzwerk, die man per UNC-Pfad anspricht.

Profil-Kosmetik

Aufmerksamen Lesern wird in den letzten beiden Beispielen die Floskel `|Out-Null` aufgefallen sein: Sie dient nur der Kosmetik. `New-PSDrive` liefert nämlich als Ergebnis ein Laufwerks-Objekt, das die PowerShell normalerweise in Form einer Tabelle ausgeben würde. Weil es hässlich wäre, wenn das bei jedem PowerShell-Start passierte, leitet `|Out-Null` diese Ausgabe ins Nirwana um.

Apropos Profil-Hygiene: Streng genommen sind die weiter oben gezeigten Code-Zeilen zum Einfügen des Profil-Ordners in den `Path` nicht ganz sauber. Sie lassen nämlich die eigentlich nur lokal benötigte Variable `$myDir` übrig. Profil-Skripte laufen im globalen Kontext – und das ist auch gut so, denn dadurch bleiben alle hier definierten Funktionen, Aliase, Laufwerke und so weiter erhalten, nachdem die PowerShell mit der Abarbeitung des Skriptes fertig ist. Aber dasselbe gilt eben auch für unbedacht angelegte temporäre Variablen. Verhindern lässt sich das, indem man Code-Abschnitte, die solche Variablen verwenden, in ein Paar geschweifter Klammern einschließt und ein &-Zeichen davorschreibt:

```
& {
  $temp = 'Irgendwas'
  # $temp verwenden ...
}
# Hier ist $temp nicht mehr definiert.
```

Dadurch erzeugt man – ähnlich wie innerhalb einer Funktion – einen eigenen Ausführungskontext. Die Gültigkeit aller darin definierten Variablen, Funktionen und so weiter bleibt auf den durch das Klammerpaar markierten Bereich beschränkt. Wer mehr über diese Gültig-

```
function prompt
{
  # Den aktuellen Pfad auf maximal 60 Zeichen kürzen
  function Trim-Path() {
    $max = 60
    $slash = [IO.Path]::DirectorySeparatorChar # Windows: \, *ix: /
    $path = (Get-Location).Path
    if($path.Length -gt $max) {
      $dirs = $path.Split($slash)
      if($dirs.Count -gt 3) {
        $head = $dirs[0] + $slash + $dirs[1] + $slash
        $tail = $dirs[-1]
        for($count = $dirs.Count - 2; $count -gt 1; $count -= 1) {
          if(($head.Length + $tail.Length + $dirs[$count].Length + 4) ⮒
                                                                ↳-gt $max) {
            break
          }
          $tail = $dirs[$count] + $slash + $tail
        }
        if($count -ge 2) {
          $tail = '...' + $slash + $tail
        }
        $path = $head + $tail
      }
    }
    return $path
  }

  $cr = if($psISE) { "" } else { "`n" }
  Write-Host "$($cr)PS " -NoNewline
  Write-Host (Trim-Path) -NoNewline -ForegroundColor Yellow
  return "> "
}
```

Anders als in anderen Konsolen ist der Prompt in der PowerShell keine Variable, sondern eine Funktion. Die gezeigte Version beschneidet die Angabe des aktuellen Verzeichnisses auf eine sinnvolle Länge und färbt sie ein.

keitsbereiche wissen will, sollte den Artikel `help about_Scopes` lesen. (Warum eine Änderung an `$Env:Path` im Beispiel trotzdem global durchschlägt? Weil das gar keine echte Variable ist, sondern eine Abkürzung für den Eintrag `Path` auf dem PSDrive `Env`.)

Prompt bedient

Wenn die PowerShell bereit ist, einen neuen Befehl entgegenzunehmen, signalisiert sie das, indem sie einen sogenannten Prompt ausgibt:

```
PS C:\Windows> _
```

Anders als bei der Eingabeaufforderung und den meisten anderen Textkonsolen ist diese Ausgabe nicht in einer mit Makros garnierten Zeichenkette gespeichert. Stattdessen ruft die PowerShell jedes Mal, wenn sie einen neuen Prompt anzeigen will, die Funktion `prompt()` auf und gibt deren Rückgabewert als Text aus.

Wer den Prompt eigenen Wünschen anpassen will, muss einfach diese Funktion neu definieren. Um Informationen wie den Rechnernamen, den freien Plattenplatz oder die aktuelle Uhrzeit in den Prompt aufzunehmen, muss man sich keine kryptischen Kürzel merken: Man kann einfach die Befehle verwenden, die man auch sonst benutzen würde, um diese Angaben abzufragen, und muss deren Ergebnisse nur zu einer Zeichenkette zusammenbasteln.

Statt den Prompt als String zurückzugeben, kann die Funktion `prompt()` auch das Cmdlet `Write-Host` verwenden, um ihn komplett oder teilweise direkt auf den Bildschirm zu bringen. Sinnvoll ist das vor allem, wenn man einen bunten Prompt bevorzugt: `Write-Host` verdaut die optionalen Argumente `BackgroundColor` und `ForegroundColor`; für gültige Werte siehe `help Write-Host`. Außerdem kennt `Write-Host` die Option `-NoNewline`, die den normalerweise am Ende der Ausgabe automatisch eingefügten Zeilenumbruch unterdrückt – wichtig, wenn der Prompt einzeilig bleiben soll.

Die nebenstehend gezeigte Version einer Prompt-Funktion demonstriert einige der Möglichkeiten, die sich daraus ergeben, dass man den Prompt frei programmieren kann. Die eingebettete Funktion `Trim-Path()` behebt ein eher kosmetisches Problem: Wenn man in sehr tiefen Ordner- oder Registry-Strukturen navigiert, wird die Angabe des aktuellen Verzeichnisses im Prompt gerne mal so lang, dass man auf derselben Zeile kaum noch einen Befehl eintippen kann. `Trim-Path()` beschneidet den angezeigten Pfad so, dass er eine festgelegte Maximallänge (`$max = 60`) nicht mehr überschreitet. Dazu zerlegt die Funktion den Pfad zunächst in seine Bestandteile, wobei sie beachtet, dass das dabei verwendete Trennzeichen (`$slash`) sich je nach Betriebssystem unterscheidet. Dann baut sie den Pfad aus dessen ersten beiden Bestandteilen (`$head`) und so vielen Verzeichnissen von hinten, wie in die gewünschte Länge passen (`$tail`), wieder zusammen. Fallen dabei Teile weg, werden sie durch drei Punkte ersetzt.

Vor der eigentlichen Ausgabe prüft `prompt()` durch Abfrage der Systemvariablen `$psISE` zunächst, ob sie gerade im ISE oder in der normalen Textkonsole läuft; nur im zweiten Fall beginnt sie den Prompt mit einem zusätzlichen Zeilenumbruch – das Konsolenfenster im ISE fügt den schon von sich aus ein. Dann gibt sie per `Write-Host` zunächst das übliche Kennzeichen „PS“ aus und dahinter – gelb hervorgehoben – den durch `Trim-Path` gegebenenfalls beschnittenen aktuellen Pfad. Die Ausgabe des abschließenden >-Zeichens überlässt sie der PowerShell, indem sie es einfach als Ergebnis zurückliefert.

Noch mehr Profil

Bestimmt fallen Ihnen noch andere als die hier gezeigten Möglichkeiten ein, sich durch pfiffige Definitionen, Funktionen oder andere Code-Schnipsel in Ihrem Profil-Skript das Leben mit der PowerShell angenehmer zu gestalten. Vielleicht sind Sie ja auch ein erfahrener PowerShell-Anwender und besitzen bereits eine Sammlung häufig benutzter Funktiönchen oder trickreicher Prompt-Bestandteile. Dann lassen Sie doch andere c’t-Leser davon profitieren: Unter ct.de/ykrs haben wir ein Diskussionsforum eingerichtet, in dem Sie Ihre Profil-Tricks teilen können. Hier finden Sie auch eine Profil-Datei zum Herunterladen, die alle in diesem Artikel vorgestellten Bestandteile enthält. *(hos@ct.de)* c't

Literatur

[1] Hajo Schulz, Aufbruch mit Power, PowerShell: Loslegen mit Microsofts mächtiger Kommandozeile, c’t 2/2018, S. 166

[2] Hajo Schulz, Power-Automatik, Loslegen mit der PowerShell, Teil 2, c’t 6/2018, S. 168

Diskussionsforum, Referenz-Informationen: ***ct.de/ykrs***