Remove Ghost Devices

removeghosts.ps1

```
<#
.SYNOPSIS
   Removes ghost devices from your system

.DESCRIPTION
    This script will remove ghost devices from your system. These are devices that are present
but have an "InstallState" as false. These devices are typically shown as 'faded'
    in Device Manager, when you select "Show hidden and devices" from the view menu. This
script has been tested on Windows 2008 R2 SP2 with PowerShell 3.0, 5.1, Server 2012R2
    with Powershell 4.0 and Windows 10 Pro with Powershell 5.1. There is no warranty with this
script. Please use cautiously as removing devices is a destructive process without
    an undo.

.PARAMETER filterByFriendlyName
This parameter will exclude devices that match the partial name provided. This paramater needs
to be specified in an array format for all the friendly names you want to be excluded.
"Intel" will match "Intel(R) Xeon(R) CPU E5-2680 0 @ 2.70GHz". "Loop" will match "Microsoft
Loopback Adapter".

.PARAMETER narrowByFriendlyName
This parameter will include devices that match the partial name provided. This paramater needs
to be specified in an array format for all the friendly names you want to be included.
"Intel" will match "Intel(R) Xeon(R) CPU E5-2680 0 @ 2.70GHz". "Loop" will match "Microsoft
Loopback Adapter".

.PARAMETER filterByClass
This parameter will exclude devices that match the class name provided. This paramater needs
to be specified in an array format for all the class names you want to be excluded.
This is an exact string match so "Disk" will not match "DiskDrive".

.PARAMETER narrowByClass
```

```
This parameter will include devices that match the class name provided. This paramater needs
to be specified in an array format for all the class names you want to be included.
This is an exact string match so "Disk" will not match "DiskDrive".

.PARAMETER listDevicesOnly
listDevicesOnly will output a table of all devices found in this system.

.PARAMETER listGhostDevicesOnly
listGhostDevicesOnly will output a table of all 'ghost' devices found in this system.

.PARAMETER force
If specified, each matching device will be removed WITHOUT any confirmation!

.EXAMPLE
Lists all devices
. "removeGhosts.ps1" -listDevicesOnly

.EXAMPLE
Save the list of devices as an object
$Devices = . "removeGhosts.ps1" -listDevicesOnly

.EXAMPLE
Lists all 'ghost' devices
. "removeGhosts.ps1" -listGhostDevicesOnly

.EXAMPLE
Lists all 'ghost' devices with a class of "Net"
. "removeGhosts.ps1" -listGhostDevicesOnly -narrowByClass Net

.EXAMPLE
Lists all 'ghost' devices with a class of "Net" AND a friendly name matching "Realtek"
. "removeGhosts.ps1" -listGhostDevicesOnly -narrowbyfriendlyname Realtek -narrowbyclass Net

.EXAMPLE
Save the list of 'ghost' devices as an object
$ghostDevices = . "removeGhosts.ps1" -listGhostDevicesOnly

.EXAMPLE
Remove all ghost devices EXCEPT any devices that have "Intel" or "Citrix" in their friendly
name
```

```
. "removeGhosts.ps1" -filterByFriendlyName @("Intel","Citrix")

.EXAMPLE
Remove all ghost devices that have "Intel" in their friendly name
. "removeGhosts.ps1" -narrowByFriendlyName Intel

.EXAMPLE
Remove all ghost devices EXCEPT any devices that are apart of the classes "LegacyDriver" or
"Processor"
. "removeGhosts.ps1" -filterByClass @("LegacyDriver","Processor")

.EXAMPLE
Remove all ghost devices EXCEPT for devices with a friendly name of "Intel" or "Citrix" or
with a class of "LegacyDriver" or "Processor"
. "removeGhosts.ps1" -filterByClass @("LegacyDriver","Processor") -filterByFriendlyName
@("Intel","Citrix")

.EXAMPLE
Remove all ghost network devices i.e. the ones with a class of "Net"
. "removeGhosts.ps1" -narrowByClass Net

.EXAMPLE
Remove all ghost devices without confirmation
. "removeGhosts.ps1" -Force

.NOTES
Permission level has not been tested.  It is assumed you will need to have sufficient rights
to uninstall devices from device manager for this script to run properly.
#>

Param(
  [array]$FilterByClass,
  [array]$NarrowByClass,
  [array]$FilterByFriendlyName,
  [array]$NarrowByFriendlyName,
  [switch]$listDevicesOnly,
  [switch]$listGhostDevicesOnly,
  [switch]$Force
)
```

```
#parameter futzing
$removeDevices = $true
if ($FilterByClass -ne $null) {
    write-host "FilterByClass: $FilterByClass"
}

if ($NarrowByClass -ne $null) {
    write-host "NarrowByClass: $NarrowByClass"
}

if ($FilterByFriendlyName -ne $null) {
    write-host "FilterByFriendlyName: $FilterByFriendlyName"
}

if ($NarrowByFriendlyName -ne $null) {
    write-host "NarrowByFriendlyName: $NarrowByFriendlyName"
}

if ($listDevicesOnly -eq $true) {
    write-host "List devices without removal: $listDevicesOnly"
    $removeDevices = $false
}

if ($listGhostDevicesOnly -eq $true) {
    write-host "List ghost devices without removal: $listGhostDevicesOnly"
    $removeDevices = $false
}

if ($Force -eq $true) {
    write-host "Each removal will happen without any confirmation: $Force"
}

function Filter-Device {
    Param (
        [System.Object]$dev
    )
    $Class = $dev.Class
    $FriendlyName = $dev.FriendlyName
    $matchFilter = $false
```

```
    if (($matchFilter -eq $false) -and ($null -ne $FilterByClass)) {
        foreach ($ClassFilter in $FilterByClass) {
            if ($ClassFilter -eq $Class) {
                Write-verbose "Class filter match $ClassFilter, skipping"
                $matchFilter = $true
                break
            }
        }
    }
    if (($matchFilter -eq $false) -and ($null -ne $NarrowByClass)) {
        $shouldInclude = $false
        foreach ($ClassFilter in $NarrowByClass) {
            if ($ClassFilter -eq $Class) {
                $shouldInclude = $true
                break
            }
        }
        $matchFilter = !$shouldInclude
    }
    if (($matchFilter -eq $false) -and ($null -ne $FilterByFriendlyName)) {
        foreach ($FriendlyNameFilter in $FilterByFriendlyName) {
            if ($FriendlyName -like '*'+$FriendlyNameFilter+'*') {
                Write-verbose "FriendlyName filter match $FriendlyName, skipping"
                $matchFilter = $true
                break
            }
        }
    }
    if (($matchFilter -eq $false) -and ($null -ne $NarrowByFriendlyName)) {
        $shouldInclude = $false
        foreach ($FriendlyNameFilter in $NarrowByFriendlyName) {
            if ($FriendlyName -like '*'+$FriendlyNameFilter+'*') {
                $shouldInclude = $true
                break
            }
        }
        $matchFilter = !$shouldInclude
    }
    return $matchFilter
}
```

```
function Filter-Devices {
    Param (
        [array]$devices
    )
    $filteredDevices = @()
    foreach ($dev in $devices) {
        $matchFilter = Filter-Device -Dev $dev
        if ($matchFilter -eq $false) {
            $filteredDevices += @($dev)
        }
    }
    return $filteredDevices
}
function Get-Ghost-Devices {
    Param (
        [array]$devices
    )
    return ($devices | Where-Object {$_.InstallState -eq $false} | Sort-Object -Property
FriendlyName)
}

# NOTE: White spaces are important in $setupapi for some reason!
$setupapi = @"
using System;
using System.Diagnostics;
using System.Text;
using System.Runtime.InteropServices;
namespace Win32
{
    public static class SetupApi
    {
         // 1st form using a ClassGUID only, with Enumerator = IntPtr.Zero
        [DllImport("setupapi.dll", CharSet = CharSet.Auto)]
        public static extern IntPtr SetupDiGetClassDevs(
           ref Guid ClassGuid,
           IntPtr Enumerator,
           IntPtr hwndParent,
           int Flags
        );
```

```
// 2nd form uses an Enumerator only, with ClassGUID = IntPtr.Zero
[DllImport("setupapi.dll", CharSet = CharSet.Auto)]
public static extern IntPtr SetupDiGetClassDevs(
   IntPtr ClassGuid,
   string Enumerator,
   IntPtr hwndParent,
   int Flags
);

[DllImport("setupapi.dll", CharSet = CharSet.Auto, SetLastError = true)]
public static extern bool SetupDiEnumDeviceInfo(
    IntPtr DeviceInfoSet,
    uint MemberIndex,
    ref SP_DEVINFO_DATA DeviceInfoData
);

[DllImport("setupapi.dll", SetLastError = true)]
public static extern bool SetupDiDestroyDeviceInfoList(
    IntPtr DeviceInfoSet
);
[DllImport("setupapi.dll", CharSet = CharSet.Auto, SetLastError = true)]
public static extern bool SetupDiGetDeviceRegistryProperty(
    IntPtr deviceInfoSet,
    ref SP_DEVINFO_DATA deviceInfoData,
    uint property,
    out UInt32 propertyRegDataType,
    byte[] propertyBuffer,
    uint propertyBufferSize,
    out UInt32 requiredSize
);
[DllImport("setupapi.dll", SetLastError = true, CharSet = CharSet.Auto)]
public static extern bool SetupDiGetDeviceInstanceId(
    IntPtr DeviceInfoSet,
    ref SP_DEVINFO_DATA DeviceInfoData,
    StringBuilder DeviceInstanceId,
    int DeviceInstanceIdSize,
    out int RequiredSize
);
```

```
        [DllImport("setupapi.dll", CharSet = CharSet.Auto, SetLastError = true)]
        public static extern bool SetupDiRemoveDevice(IntPtr DeviceInfoSet,ref SP_DEVINFO_DATA
DeviceInfoData);
    }
    [StructLayout(LayoutKind.Sequential)]
    public struct SP_DEVINFO_DATA
    {
       public uint cbSize;
       public Guid classGuid;
       public uint devInst;
       public IntPtr reserved;
    }
    [Flags]
    public enum DiGetClassFlags : uint
    {
        DIGCF_DEFAULT       = 0x00000001,  // only valid with DIGCF_DEVICEINTERFACE
        DIGCF_PRESENT       = 0x00000002,
        DIGCF_ALLCLASSES    = 0x00000004,
        DIGCF_PROFILE       = 0x00000008,
        DIGCF_DEVICEINTERFACE   = 0x00000010,
    }
    public enum SetupDiGetDeviceRegistryPropertyEnum : uint
    {
         SPDRP_DEVICEDESC          = 0x00000000, // DeviceDesc (R/W)
         SPDRP_HARDWAREID          = 0x00000001, // HardwareID (R/W)
         SPDRP_COMPATIBLEIDS           = 0x00000002, // CompatibleIDs (R/W)
         SPDRP_UNUSED0             = 0x00000003, // unused
         SPDRP_SERVICE             = 0x00000004, // Service (R/W)
         SPDRP_UNUSED1             = 0x00000005, // unused
         SPDRP_UNUSED2             = 0x00000006, // unused
         SPDRP_CLASS               = 0x00000007, // Class (R--tied to ClassGUID)
         SPDRP_CLASSGUID           = 0x00000008, // ClassGUID (R/W)
         SPDRP_DRIVER              = 0x00000009, // Driver (R/W)
         SPDRP_CONFIGFLAGS         = 0x0000000A, // ConfigFlags (R/W)
         SPDRP_MFG             = 0x0000000B, // Mfg (R/W)
         SPDRP_FRIENDLYNAME        = 0x0000000C, // FriendlyName (R/W)
         SPDRP_LOCATION_INFORMATION    = 0x0000000D, // LocationInformation (R/W)
         SPDRP_PHYSICAL_DEVICE_OBJECT_NAME = 0x0000000E, // PhysicalDeviceObjectName (R)
         SPDRP_CAPABILITIES        = 0x0000000F, // Capabilities (R)
```

```
        SPDRP_UI_NUMBER           = 0x00000010, // UiNumber (R)
        SPDRP_UPPERFILTERS        = 0x00000011, // UpperFilters (R/W)
        SPDRP_LOWERFILTERS        = 0x00000012, // LowerFilters (R/W)
        SPDRP_BUSTYPEGUID         = 0x00000013, // BusTypeGUID (R)
        SPDRP_LEGACYBUSTYPE           = 0x00000014, // LegacyBusType (R)
        SPDRP_BUSNUMBER           = 0x00000015, // BusNumber (R)
        SPDRP_ENUMERATOR_NAME         = 0x00000016, // Enumerator Name (R)
        SPDRP_SECURITY            = 0x00000017, // Security (R/W, binary form)
        SPDRP_SECURITY_SDS        = 0x00000018, // Security (W, SDS form)
        SPDRP_DEVTYPE             = 0x00000019, // Device Type (R/W)
        SPDRP_EXCLUSIVE           = 0x0000001A, // Device is exclusive-access (R/W)
        SPDRP_CHARACTERISTICS         = 0x0000001B, // Device Characteristics (R/W)
        SPDRP_ADDRESS             = 0x0000001C, // Device Address (R)
        SPDRP_UI_NUMBER_DESC_FORMAT       = 0X0000001D, // UiNumberDescFormat (R/W)
        SPDRP_DEVICE_POWER_DATA       = 0x0000001E, // Device Power Data (R)
        SPDRP_REMOVAL_POLICY          = 0x0000001F, // Removal Policy (R)
        SPDRP_REMOVAL_POLICY_HW_DEFAULT   = 0x00000020, // Hardware Removal Policy (R)
        SPDRP_REMOVAL_POLICY_OVERRIDE     = 0x00000021, // Removal Policy Override (RW)
        SPDRP_INSTALL_STATE           = 0x00000022, // Device Install State (R)
        SPDRP_LOCATION_PATHS          = 0x00000023, // Device Location Paths (R)
        SPDRP_BASE_CONTAINERID        = 0x00000024  // Base ContainerID (R)
    }
}
"@
Add-Type -TypeDefinition $setupapi

    #Array for all removed devices report
    $removeArray = @()
    #Array for all devices report
    $array = @()

    $setupClass = [Guid]::Empty
    #Get all devices
    $devs = [Win32.SetupApi]::SetupDiGetClassDevs([ref]$setupClass, [IntPtr]::Zero,
[IntPtr]::Zero, [Win32.DiGetClassFlags]::DIGCF_ALLCLASSES)

    #Initialise Struct to hold device info Data
    $devInfo = new-object Win32.SP_DEVINFO_DATA
    $devInfo.cbSize = [System.Runtime.InteropServices.Marshal]::SizeOf($devInfo)
```

```
    #Device Counter
    $devCount = 0
    #Enumerate Devices
    while([Win32.SetupApi]::SetupDiEnumDeviceInfo($devs, $devCount, [ref]$devInfo)) {

        #Will contain an enum depending on the type of the registry Property, not used but
required for call
        $propType = 0
        #Buffer is initially null and buffer size 0 so that we can get the required Buffer
size first
        [byte[]]$propBuffer = $null
        $propBufferSize = 0
        #Get Buffer size
        [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs, [ref]$devInfo,
[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_FRIENDLYNAME, [ref]$propType, $propBuffer,
0, [ref]$propBufferSize) | Out-null
        #Initialize Buffer with right size
        [byte[]]$propBuffer = New-Object byte[] $propBufferSize

        #Get HardwareID
        $propTypeHWID = 0
        [byte[]]$propBufferHWID = $null
        $propBufferSizeHWID = 0
        [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs, [ref]$devInfo,
[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_HARDWAREID, [ref]$propTypeHWID,
$propBufferHWID, 0, [ref]$propBufferSizeHWID) | Out-null
        [byte[]]$propBufferHWID = New-Object byte[] $propBufferSizeHWID

        #Get DeviceDesc (this name will be used if no friendly name is found)
        $propTypeDD = 0
        [byte[]]$propBufferDD = $null
        $propBufferSizeDD = 0
        [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs, [ref]$devInfo,
[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_DEVICEDESC, [ref]$propTypeDD,
$propBufferDD, 0, [ref]$propBufferSizeDD) | Out-null
        [byte[]]$propBufferDD = New-Object byte[] $propBufferSizeDD

        #Get Install State
        $propTypeIS = 0
        [byte[]]$propBufferIS = $null
```

```
        $propBufferSizeIS = 0
        [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs, [ref]$devInfo,
[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_INSTALL_STATE, [ref]$propTypeIS,
$propBufferIS, 0, [ref]$propBufferSizeIS) | Out-null
        [byte[]]$propBufferIS = New-Object byte[] $propBufferSizeIS

        #Get Class
        $propTypeCLSS = 0
        [byte[]]$propBufferCLSS = $null
        $propBufferSizeCLSS = 0
        [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs, [ref]$devInfo,
[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_CLASS, [ref]$propTypeCLSS,
$propBufferCLSS, 0, [ref]$propBufferSizeCLSS) | Out-null
        [byte[]]$propBufferCLSS = New-Object byte[] $propBufferSizeCLSS
        [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs,
[ref]$devInfo,[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_CLASS, [ref]$propTypeCLSS,
$propBufferCLSS, $propBufferSizeCLSS, [ref]$propBufferSizeCLSS)  | out-null
        $Class = [System.Text.Encoding]::Unicode.GetString($propBufferCLSS)

        #Read FriendlyName property into Buffer
        if(![Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs,
[ref]$devInfo,[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_FRIENDLYNAME,
[ref]$propType, $propBuffer, $propBufferSize, [ref]$propBufferSize)){
            [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs,
[ref]$devInfo,[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_DEVICEDESC,
[ref]$propTypeDD, $propBufferDD, $propBufferSizeDD, [ref]$propBufferSizeDD)  | out-null
            $FriendlyName = [System.Text.Encoding]::Unicode.GetString($propBufferDD)
            #The friendly Name ends with a weird character
            if ($FriendlyName.Length -ge 1) {
                $FriendlyName = $FriendlyName.Substring(0,$FriendlyName.Length-1)
            }
        } else {
            #Get Unicode String from Buffer
            $FriendlyName = [System.Text.Encoding]::Unicode.GetString($propBuffer)
            #The friendly Name ends with a weird character
            if ($FriendlyName.Length -ge 1) {
                $FriendlyName = $FriendlyName.Substring(0,$FriendlyName.Length-1)
            }
        }
```

```
        #InstallState returns true or false as an output, not text
        $InstallState = [Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs,
[ref]$devInfo,[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_INSTALL_STATE,
[ref]$propTypeIS, $propBufferIS, $propBufferSizeIS, [ref]$propBufferSizeIS)

        # Read HWID property into Buffer
        if(![Win32.SetupApi]::SetupDiGetDeviceRegistryProperty($devs,
[ref]$devInfo,[Win32.SetupDiGetDeviceRegistryPropertyEnum]::SPDRP_HARDWAREID,
[ref]$propTypeHWID, $propBufferHWID, $propBufferSizeHWID, [ref]$propBufferSizeHWID)){
            #Ignore if Error
            $HWID = ""
        } else {
            #Get Unicode String from Buffer
            $HWID = [System.Text.Encoding]::Unicode.GetString($propBufferHWID)
            #trim out excess names and take first object
            $HWID = $HWID.split([char]0x0000)[0].ToUpper()
        }

        #all detected devices list
        $device = New-Object System.Object
        $device | Add-Member -type NoteProperty -name FriendlyName -value $FriendlyName
        $device | Add-Member -type NoteProperty -name HWID -value $HWID
        $device | Add-Member -type NoteProperty -name InstallState -value $InstallState
        $device | Add-Member -type NoteProperty -name Class -value $Class
        if ($array.count -le 0) {
            #for some reason the script will blow by the first few entries without displaying
the output
            #this brief pause seems to let the objects get created/displayed so that they are
in order.
            Start-Sleep 1
        }
        $array += @($device)

        <#
        We need to execute the filtering at this point because we are in the current device
context
        where we can execute an action (eg, removal).
        InstallState : False == ghosted device
        #>
        if ($removeDevices -eq $true) {
```

```
            #we want to remove devices so let's check the filters...
            $matchFilter = Filter-Device -Dev $device

            if ($InstallState -eq $False) {
                if ($matchFilter -eq $false) {
                    $message  = "Attempting to remove device $FriendlyName"
                    $confirmed = $false
                    if (!$Force -eq $true) {
                        $question = 'Are you sure you want to proceed?'
                        $choices  = '&Yes', '&No'
                        $decision = $Host.UI.PromptForChoice($message, $question, $choices, 1)
                        if ($decision -eq 0) {
                            $confirmed = $true
                        }
                    } else {
                        $confirmed = $true
                    }
                    if ($confirmed -eq $true) {
                        Write-Host $message -ForegroundColor Yellow
                        $removeObj = New-Object System.Object
                        $removeObj | Add-Member -type NoteProperty -name FriendlyName -value
$FriendlyName
                        $removeObj | Add-Member -type NoteProperty -name HWID -value $HWID
                        $removeObj | Add-Member -type NoteProperty -name InstallState -value
$InstallState
                        $removeObj | Add-Member -type NoteProperty -name Class -value $Class
                        $removeArray += @($removeObj)
                        if([Win32.SetupApi]::SetupDiRemoveDevice($devs, [ref]$devInfo)){
                            Write-Host "Removed device $FriendlyName"  -ForegroundColor Green
                        } else {
                            Write-Host "Failed to remove device $FriendlyName" -
ForegroundColor Red
                        }
                    } else {
                        Write-Host "OK, skipped" -ForegroundColor Yellow
                    }
                } else {
                    write-host "Filter matched. Skipping $FriendlyName" -ForegroundColor
Yellow
                }
```

```
        }
    }
    $devcount++
}

#output objects so you can take the output from the script
if ($listDevicesOnly) {
    $allDevices = $array | Sort-Object -Property FriendlyName
    $filteredDevices = Filter-Devices -Devices $allDevices
    $filteredDevices | Format-Table
    write-host "Total devices found                : $($allDevices.count)"
    write-host "Total filtered devices found       : $($filteredDevices.count)"
    $ghostDevices = Get-Ghost-Devices -Devices $array
    $filteredGhostDevices = Filter-Devices -Devices $ghostDevices
    write-host "Total ghost devices found          : $($ghostDevices.count)"
    write-host "Total filtered ghost devices found : $($filteredGhostDevices.count)"
    return $filteredDevices | out-null
}

if ($listGhostDevicesOnly) {
    $ghostDevices = Get-Ghost-Devices -Devices $array
    $filteredGhostDevices = Filter-Devices -Devices $ghostDevices
    $filteredGhostDevices | Format-Table
    write-host "Total ghost devices found           : $($ghostDevices.count)"
    write-host "Total filtered ghost devices found  : $($filteredGhostDevices.count)"
    return $filteredGhostDevices | out-null
}

if ($removeDevices -eq $true) {
    write-host "Removed devices:"
    $removeArray  | Sort-Object -Property FriendlyName | Format-Table
    write-host "Total removed devices     : $($removeArray.count)"
    return $removeArray | out-null
}
```

Revision #2
Created 2023-11-10 04:49:09 UTC by Ryan
Updated 2025-03-12 15:24:59 UTC by Ryan