**USB Bootable Diagnostic Toolkit Guide**

### ChatGPT Question

*Research how to build a bootable Windows To Go USB solution for laptops and desktops that supports automated diagnostics using PowerShell or Python, runs on both UEFI and Legacy BIOS, and generates reports in English without using a cloud server.*

*Provide a detailed checklist covering tools, automation scripts, offline reporting, and hardware tests including display, keyboard, WiFi/Bluetooth, battery, CPU, RAM, storage, camera, and Windows Autopilot readiness. Ensure it supports auto-WiFi connection and offline report collection for PDF download.*

*Gerson Almeida*

*2025/06*

USB Bootable Diagnostic Toolkit Guide

# Overview and Goals

This guide outlines a comprehensive plan to build a **USB bootable Windows To Go diagnostic tool** for an IT refurbishing business. The goal is an **automated, offline test suite** to verify hardware on used laptops/desktops before resale, reducing RMA issues. The tool will run on both UEFI and Legacy BIOS systems, use **PowerShell/Python scripting** for automation, and produce an English-only interface/report. All diagnostics run locally (no cloud services), saving results on the USB (with option to export as PDF). Key tests include display, keyboard, trackpad, Wi-Fi, Bluetooth, battery health, CPU, RAM, storage, camera, and Windows Autopilot readiness.

# Creating the Windows To Go USB (UEFI & BIOS Boot)

By following this guide, you can set up a reusable Windows To Go USB that **automatically diagnoses hardware** for both laptops and desktops in a consistent, scripted manner. The checklist above ensures you cover all components – from display and input devices to internals like CPU, RAM, and storage – using reliable open-source or built-in tools. The final outcome is a detailed diagnostic report (HTML/PDF) for each device tested, identified by its article ID. This process will streamline your refurbishment QA and significantly reduce the chance of shipping a faulty device to customers. All tests run offline, under a unified interface, making it a self-contained solution for your IT business’s needs.

**Windows Image:** Use Windows 10 (or 11) ISO in English.

Windows 10 Enterprise is ideal (supports Windows To Go officially), but Windows 10/11 Pro can work via third-party tools (Rufus).

**USB Preparation:** Use Rufus in Windows To Go mode. Enable dual boot (*Alt-E cheat mode in Rufus*) to support both UEFI and Legacy BIOS ([elevenforum.com](https://www.elevenforum.com/t/installation-using-media-creation-tool-uefi-issue.22632/)). Format as NTFS (with an EFI system partition) if large >4GB files are needed; Rufus will handle partitioning for dual-mode.

* Alternatively, apply the Windows image to USB via DISM:
* Partition USB with both a FAT32 (boot) and NTFS (OS) partition.
* Use dism /Apply-Image to apply the Windows WIM to the NTFS partition.
* Use bcdboot to install boot files to both partitions.

Ensure the USB boots on a sample UEFI-only system and a Legacy BIOS-only system as a test.

* **Drivers:** Windows To Go will detect hardware at boot. Consider adding common drivers (Wi-Fi/Bluetooth) to the image or enabling Windows Update (if internet available) to pull drivers. No external server should be required – embed any critical drivers in the image if possible.

# Open-Source Tools and Scripting Summary

Each of these either runs locally or is packaged on the USB. No internet connectivity is required except optional Wi-Fi testing. All results are stored locally in the USB, respecting the no-cloud requirement.

To maximize transparency and cost-effectiveness, we prioritized open-source and built-in Windows tools:

* **Windows To Go** – base OS environment (Windows 10) for diagnostics.
* **Copy Utilities:** Create a C:\Tools directory on the USB for all diagnostic executables and scripts:
* **PowerShell & Python:** Make sure PowerShell (built-in) is updated. Install Python 3 if you plan to use Python scripts (e.g. via offline installer) and add it to PATH. Use PowerShell for easy access to WMI and system info (battery, CPU, etc.).

Optionally call Python scripts for complex tests (e.g. webcam capture or any GUI automation).  
Ensure scripts do not require internet (except perhaps for Wi-Fi connectivity test) and handle all logic locally.

* **PixCheck or PixelTest**: An open-source screen tester to cycle through colors for display check (if available, or use a custom script).
* **Keyboard Tester**: For example, the open-source (keyboardtester.exe) which highlights keys as pressed ([github.com](https://github.com/10yard/keyboardtester))
* **CPU Stress Tool**: Place a tool like (CPUstres.exe - from Sysinternals) or a portable stress-test utility. Ensure any CLI options for duration or logging are known.
* **Sysinternals Suite (CpuStres)** – free Microsoft tool for CPU load (not OSS, but trusted). Alternatively, scripts or OCCT (free) for stress tests.
* **smartmontools (smartctl)** – open-source SMART diagnostic tool for drives ([smartmontools.org](https://www.smartmontools.org/))
* **OpenCV** – open-source computer vision library used via Python to test webcam.
* **Get-WindowsAutoPilotInfo.ps1** – Microsoft-provided script for Autopilot (source available via PowerShell gallery).
* **DiskSpd**: Copy (diskspd.exe) (open-source CLI disk tester by Microsoft) ([en.wikipedia.org](https://en.wikipedia.org/wiki/DiskSpd)) for storage performance tests.
* **smartctl**: Include (smartctl.exe) from smartmontools for checking drive SMART health [smartmontools.org](https://www.smartmontools.org/)
* **Camera Test Script**: If using Python/OpenCV, ensure OpenCV (pip install opencv-python offline) and prepare a script to capture an image from the webcam. Alternatively, include an (ffmpeg.exe) to take a snapshot from the default camera.
* **Autopilot Script**: Download (Get-WindowsAutoPilotInfo.ps1) (Microsoft script) and save it in Tools. This script will fetch the device’s hardware hash for Autopilot enrollment [learn.microsoft.com](https://learn.microsoft.com/en-us/autopilot/add-devices)
* **Automation Scripts**: Develop a master PowerShell script (e.g. Start-Diagnostics.ps1) and supporting scripts:

Setup Checklist for the Diagnostic Environment

* Install Required Tools**:** On the Windows To Go OS, install or copy open-source diagnostic utilities:
* ***Battery****:* rely on built-in powercfg and PowerShell (no extra install needed).  
  <https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/powercfg-command-line-options#option_systempowerreport>  
  powercfg /systempowerreport
* ***Display****:* pixel test utility (e.g. **PixCheck** or similar) for full-screen color/stuck pixel tests.
* ***Input****:* a keyboard tester (open-source) that shows on-screen key presses [github.com](https://github.com/10yard/keyboardtester).
* ***Network****:* include Wi-Fi drivers (if not in OS) and any Bluetooth command tools or Python libs (e.g. Bleak for BLE).
* Network Auto-Connect**:** Pre-configure a Wi-Fi profile for your workshop network. Use netsh wlan to add the Wi-Fi profile and auto-connect at boot [reddit.com](https://www.reddit.com/r/Intune/comments/17p5ia3/most_efficient_way_of_collecting_hardware_hashes/). For example:

powershell

netsh wlan add profile filename="WiFiProfile.xml" user=all

netsh wlan connect name="YourSSID"

* ***CPU/RAM****:* stress test tools (e.g. CPUstres from Sysinternals or open-source OCCT/Prime95).
* ***Storage****:* *DiskSpd* (open-source disk benchmark) [en.wikipedia.org](https://en.wikipedia.org/wiki/DiskSpd) and smartmontools (for SMART health) [smartmontools.org](https://www.smartmontools.org/).
* ***Camera****:* Python with OpenCV or an ffmpeg binary for webcam testing (optional).
* ***Autopilot****:* download **Get-WindowsAutopilotInfo.ps1** script for hardware hash collection [learn.microsoft.com](https://learn.microsoft.com/en-us/autopilot/add-devices).
* **Configure Auto-Start:** Set up the Windows To Go environment to auto-run a master **PowerShell script at startup** (e.g. via Startup folder or Scheduled Task). This script will orchestrate the diagnostics:

1. Connect to Wi-Fi automatically (using pre-saved profile).
2. Prompt for an **Article ID** or device identifier.
3. Sequentially execute each hardware test and log the results.
4. Generate a local report (HTML/text) and optionally convert to PDF.

* Ensure the profile XML (exported with credentials in plaintext [reddit.com](https://www.reddit.com/r/Intune/comments/17p5ia3/most_efficient_way_of_collecting_hardware_hashes/) is included on the USB. This allows the diagnostics to access network (if needed for updates or to export reports) even with no cloud dependency.

# Automation Strategy (Orchestrating the Tests)

* The heart of the solution is an automation script that runs on boot. Outline of the **PowerShell automation workflow**:
* **Startup Launch:** Configure Start-Diagnostics.ps1 to run at user logon (place a shortcut in Shell:Startup or use HKLM\Software\Microsoft\Windows\CurrentVersion\Run). The script should run with Administrator privileges (enable auto-logon or have a standard password for the WTG user if needed).
* Initial Prompts:
* **Wi-Fi Connection:** The script ensures Wi-Fi is connected (using netsh or Connect-WiFi commands). If the profile is pre-set, it can simply verify connection; otherwise, it can attempt netsh wlan connect.
* **Device Identification:** Prompt the operator for the device’s Article ID or inventory number. For example:
* powershell
* $deviceID = Read-Host "Enter Device Article ID or Serial"
* This ID will be used in the report file name and within the report to identify the unit under test.
* **Run Diagnostics Sequentially:** The script will then call each test in turn, logging output. Use try/catch to handle any tool errors so that one failure doesn’t halt the suite. After each test, collect results (pass/fail or metrics). The sections below detail tool suggestions and scripting for each test category.
* **Result Aggregation:** Accumulate all test results into a single report object (or file). This could be as simple as building an HTML string or a structured text/CSV. PowerShell can easily append to a text file or build HTML.
* **Report Generation:** Save the consolidated report to the USB (e.g. Reports\DeviceID\_Diagnostics.html). If PDF export is required, you can:
* Print the HTML to PDF using a command-line tool like **wkhtmltopdf** or **Pandoc** (include one if available), or
* Install a PDF printer and script a print-to-PDF. This provides a PDF report that can be moved off the device.
* **Notification/Completion:** Optionally, beep or display a message when all tests are done. The tech can then copy the PDF or review results.
* Throughout the script, ensure the interface remains **English-only** (e.g., any output or tool selected should be configured for English).

# Diagnostic Test Suite Components

# 1- Battery Health and Performance Test

**Objective:** Check the battery’s condition (capacity health, ability to hold charge) and basic discharge functionality.

**Tool/Method:** Leverage Windows’ built-in battery info and possibly a short stress test on battery:

* Use the Windows **Battery Report** feature: run powercfg /batteryreport which generates an HTML report of battery stats. Parse or extract key fields:
* Design Capacity vs. Last Full Charge (in mWh) – calculate health %. (The report or WMI will show these. Generally if full charge is <80% of design, the battery is degraded[powershell.one](https://powershell.one/wmi/root/cimv2/win32_battery).)
* Cycle Count (if available via BatteryStaticData WMI or the report).
* Recent usage (the battery report includes a 3-day usage graph; not crucial for our use).
* Use **WMI (Win32\_Battery)**: Query properties like EstimatedChargeRemaining (current %), BatteryStatus, etc. Also query the newer WMI classes in ROOT\WMI namespace: BatteryFullChargedCapacity and BatteryStatus which give full charge capacity and current charge in mWh. These can provide precise numbers to compute health percentage.
* **Battery Stress (optional):** If time allows, you can test discharge: e.g. if the battery is above 50%, run a CPU load for a few minutes on battery to see how the percentage drops. This can uncover a battery that suddenly dies. However, doing this for every device might be time-consuming. It might be sufficient to note the wear level and ensure it can run on battery at least for a short while.
* Automation:
* If device is on AC, the script can record battery health metrics. Optionally prompt the tech to disconnect AC for a moment to ensure the device stays powered on battery.
* Run a quick battery load test: for example, if on battery, log the battery %, then run a CPU load for 1–2 minutes, then log the % again to see drop. Reconnect AC after test.
* Use powercfg /batteryreport and save the HTML in the report folder for detailed reference. Also parse the generated battery-report.html for key numbers: Design vs Full Charge capacity (PowerShell can parse the XML in it or use regex).

**Logging:** Report the battery health as a percentage (e.g. “Battery Health: 85% of design capacity” and whether it’s above the 80% threshold for good health [powershell.one](https://powershell.one/wmi/root/cimv2/win32_battery). Include current charge, cycle count (if available), and whether the battery passed a short discharge test. If the battery is failing (e.g., won’t hold charge or wear is high), flag that in the summary.

# 2- Display Test (Screen & Discoloration)

**Objective:** Verify the screen has no dead pixels, discoloration, or major artifacts.

**Tool/Method:** Use a display test pattern utility or script:

* An open-source tool like **PixCheck** (if available) can cycle solid colors (red, green, blue, white, black) to expose dead or stuck pixels. This tool is designed to help find screen pixel defects [github.com](https://github.com/victornpb/pixels).
* **Automation:** If using a tool, have the script launch it and wait a fixed duration for each pattern. True automated detection of discoloration is not feasible without a sensor; the test may require the technician to observe the screen. The process can be:
* Display solid colors full-screen in sequence (PowerShell can use a WinForms window or an HTML fullscreen, or PixCheck if it has an automation mode).
* Pause a few seconds on each color. Possibly beep or log each transition.
* At the end, prompt the user: “Press Y if display passed (no issues) or N if issues were observed.” Record the input for the report.
* **Open-Source Alternative:** If no GUI tool, create a simple PowerShell-driven full-screen window that changes background color. .NET in PowerShell can create a form; or use a Python script with OpenCV/PyGame to show color screens.
* **Logging:** Note in the report whether the screen test passed and any operator comments (e.g. “stuck pixel at top-right”). This test relies on human validation for visual quality.

# 3- Keyboard and Trackpad Test

**Objective:** Ensure all keys and the trackpad function properly.

**Tool/Method:** It’s difficult to fully automate without user interaction, but you can guide the process:

* **Keyboard:** Launch an on-screen keyboard test utility (like the open-source **KeyboardTester** [github.com](https://github.com/10yard/keyboardtester) or similar) which highlights keys as they are pressed. The technician can then press all keys one by one. The utility’s visual feedback confirms each key’s switch works.
* **Trackpad:** Simply moving the cursor and using left/right click can be tested. Windows built-in “Mouse Properties” or a simple cursor movement is enough. Some diagnostics use drawing programs to test trackpad (e.g. ask user to draw in MSPaint to ensure pointer tracking and clicking works).

**Automation:** The master script can:

* + Display instructions: “Press each key on the keyboard in sequence. Ensure each key highlights on the on-screen tester. Move the trackpad and click buttons to verify pointer responds.”
  + Optionally, after a timeout or on user confirmation, record that the test is complete. You might require the user to input something like typing the alphabet or a sentence, which the script can capture via Read-Host to confirm keys are producing input.
  + If any key failed, the user can indicate failure.

**Logging:** If possible, capture key scan-codes via a PowerShell .NET hook or a small C# helper to detect unpressed keys. Simpler is to rely on user confirmation. Log “All keys OK” or list any non-working keys. For the trackpad, log “Pointer/Click OK” if movement was observed. (These will likely be subjective entries by the operator, as full automation is limited here.)

# 4. Wi-Fi and Bluetooth Test

**Objective:** Validate wireless adapters are functioning: Wi-Fi connects and transfers data; Bluetooth can enable and discover devices.

**Tool/Method:**

* **Wi-Fi Test:** The script should confirm that the Wi-Fi adapter is present and can connect:
* Use Get-NetAdapter -InterfaceDescription \*Wireless\* or Get-PnpDevice -Class Net to ensure the Wi-Fi NIC is detected.
* Since we configured auto-connect, check for an IP address Get-NetIPConfiguration. Then test connectivity: e.g. ping the gateway or a known IP (like 8.8.8.8) and/or perform a brief speed test on the LAN. If internet access is available (though no cloud storage, using internet for testing is fine), you can download a small file or use PowerShell Invoke-WebRequest to a known site to measure throughput.
* Log the Wi-Fi signal strength and link speed (PowerShell: netsh wlan show interfaces gives signal quality and rate).
* **Automation:** All of the above can run without user input. If Wi-Fi fails to connect, the script should flag it and perhaps attempt a troubleshoot (ensure the profile is correct or suggest checking the wireless adapter switch).
* **Bluetooth Test:** This is a bit more involved:
* Ensure the Bluetooth radio is detected (Get-PnpDevice -Class Bluetooth should show an “OK” status device).
* Use PowerShell or a small utility to scan for nearby Bluetooth devices. For example, a Python approach with the **Bleak** library can scan BLE advertisements, or use a tool like **BluetoothCLI** if available. If no specific tool, at least attempt to turn the adapter on and make it discoverable (using Enable-PnpDevice or opening Bluetooth settings via command).
* Optionally, if you have a known test device (like a beacon or speaker in pairing mode), try to discover it by name to confirm the radio works.
* **Automation:** You might automate launching a Bluetooth scan and parsing results. Without a known device, simply logging that the adapter can scan is enough. For example, use Start-BluetoothScan.ps1 that calls powershell.exe

/c \"[Windows.Devices.Bluetooth.BluetoothAdapter]::GetDefaultAsync()\" or other API to scan. (This may require a custom script or .NET code.)

* If a device is found or the scan API returns devices, consider that a pass. If the adapter is missing or errors out, mark fail.
* **Logging:** For Wi-Fi, record SSID connected, signal%, and a simple ping result (latency or success/fail). For Bluetooth, record “Adapter OK” and whether any devices were discovered (e.g. "3 devices found in scan"). Mention if a known test device was detected to confirm functionality.

# 5- CPU/RAM Stress and Health Test

**Objective:** Verify the CPU is stable under load and within expected performance.

**Tool Suggestions:** Use an open-source or free CPU stress utility or a scripted workload:

* **Microsoft CPUstres** (Sysinternals): A lightweight tool to max out CPU threads[learn.microsoft.com](https://learn.microsoft.com/en-us/sysinternals/downloads/cpustres). It has a GUI but you can configure threads/activity level. Alternatively, **OCCT** (free) or **Prime95** torture test can be used – these are not open-source but widely trusted for stress testing.
* **Open-Source Alternative:** A simple way is to use PowerShell to spin CPU: e.g. [System.Threading.Tasks.Parallel]::For(0, [Environment]::ProcessorCount, { while($true){ } }) which will peg all cores. Or run a small Python script computing prime numbers on multiple threads. Ensure it runs for a few minutes.
* **Thermal Monitoring:** Ideally, monitor CPU temperature and throttling during the test. Open-source tools like **Open Hardware Monitor** (open source library) or using WMI (MSAcpi\_ThermalZoneTemperature) could get temperatures, but often require vendor-specific sensors. You might instead use the Windows Performance Counter for CPU frequency or rely on the system not throttling significantly. If you have **HWiNFO** or similar, it could log max temperature (but HWiNFO isn’t open-source). Open Hardware Monitor library can be used via a script to read sensors if included.
* Automation:
* Launch the CPU stress for a fixed duration (e.g. 5 minutes). This can be done by starting the stress tool with parameters (some tools allow command-line duration). If using a custom script, run it asynchronously and have the main script wait.
* While the stress test runs, you can sample CPU metrics: use Get-Counter "\Processor(\_Total)\% Processor Time" to confirm ~100% usage, and possibly sample Get-Counter "\Thermal Zone Information(\*)\Temperature" if available.
* After the test, check for system stability: Did the system crash or did the stress tool report errors (Prime95 will report hardware errors if any). If using Prime95, you can have it run a few self-test iterations and then stop it via command/script.
* **Logging:** Note the outcome: “CPU stress test passed for 5 minutes at 100% load, no errors. Max CPU temp ~85°C (if available).” If any issue (thermal throttling or errors) occurred, record it (e.g., “CPU thermal throttled” or the tool’s error output). Also capture CPU info like model and clock speed from WMI (Win32\_Processor) for the record.

# 6- RAM Test

**Objective:** Check system memory for errors.

**Tool Suggestions:** Thorough memory testing usually requires a reboot (e.g. MemTest86 or Windows Memory Diagnostic). Within the Windows To Go environment, options are:

* **Windows Memory Diagnostic:** You can invoke mdsched.exe to schedule a mem test on next boot. However, since we are booted from USB, running this will reboot out of the environment and then you’d have to boot back. This disrupts the automation, so it may be better to run separately. If doing so, a technician could manually initiate it after the main tests and record results. The results are stored in the event log which you could parse on next boot. This is one approach for deep testing.
* **In-OS Memory Test:** Use a stress test like **OCCT** or **Prime95 Blend** which also exercises RAM. OCCT has a memory test, and Prime95’s blend test hits memory hard (though not full coverage). Not fully deterministic, but might catch glaring issues.
* **Custom Script:** Allocate as much memory as possible and verify it holds data:
* For example, in PowerShell create a large byte array and fill it with a pattern, then verify the pattern. This can detect if an obvious error occurs (though Windows memory protection might hide some issues).
* There are open-source memory testers for Windows (like HCI MemTest has a free version, but not open-source; or “stress-ng” on Windows Subsystem for Linux if that’s an option, probably not here).
* Automation:
* E.g., use PowerShell to allocate chunks of memory:

***powershell***

$mem = New-Object byte[] (1024\*1024\*500) # allocate 500MB

[byte]0xAA \* $mem.Length | % { $mem[$\_] = 0xAA } # fill with 0xAA

# then verify:

for ($i=0; $i -lt $mem.Length; $i+=4096) {

if ($mem[$i] -ne 0xAA) { $memoryError = $true; break }

}

* This is a simplistic check for stuck bits. Increase allocation to use most of RAM (but leave some for OS to avoid crashing).
* Alternatively, run a known stress tool like Prime95 in background for some minutes focusing on memory.
* **Logging:** If no obvious errors are found (no crashes, no verify mismatches), mark “Memory test passed (X GB OK)”. If a memory test returns errors or if the system BSODs during heavy memory use, note that. In general, if the CPU stress test passed, memory is likely okay too, but a dedicated memory test is still good for catching bad sticks.
* **Note:** For critical refurbishments, you might still want to run an overnight MemTest86 from a separate boot for full coverage. In this automated suite, we aim for a reasonable quick check to catch major faults.

# 7- Storage (Disk) Test

**Objective:** Check the health and performance of the storage drive (HDD or SSD).

**Tools:** Use a combination of **SMART data check** and a read/write benchmark:

* **SMART Health Check:** Utilize **smartmontools** (smartctl.exe) to read SMART attributes. This open-source tool can retrieve health status from the drive firmware [smartmontools.org](https://www.smartmontools.org/). For example:

shell

smartctl.exe -H /dev/sda (for health summary)

smartctl.exe -A /dev/sda (for full SMART attributes)

* On Windows, drives are referred by number (e.g. smartctl -x /dev/sda or smartctl -x /dev/sdb for secondary). You can also use WMI (Win32\_DiskDrive and MS Storage WMI) but SMART details are easier via smartctl. Check for attributes like Reallocated Sector Count, Current Pending Sector, SSD wear leveling count, etc.

The script can interpret the SMART Health Status (PASSED/FAILED) from smartctl -H. Also parse critical attributes threshold: e.g., if Reallocated Sectors > 0 or if SSD wear leveling is high, flag it.

* Performance Benchmark: Use DiskSpd to test read/write speed:
* Run a sequential read/write test on the drive and a random I/O test. For example:

**Sequential**:

diskspd -d20 -Sh -o1 -t1 -b128K -rw=100 -w0 c:\testfile.dat

for read, and then -w100 for write (where -w sets write percentage).

* Random: diskspd -d15 -Sh -o4 -t4 -b4K -r4K -w0 c:\testfile.dat.
* These will produce output with MB/s and IOPS. Compare against typical values for the drive type (SSD vs HDD). We’re not expecting to meet top specs (especially on USB boot environment), but we want to see if the drive performs reasonably (e.g. an SSD giving ~500MB/s sequential, an HDD ~100MB/s). Very low speeds could indicate an issue.
* Keep test files small or target a specific drive letter to avoid using the USB itself. If the OS is running from USB, you should test the internal drive by its drive letter (e.g., D: if present) or physical drive number. If the internal disk is blank or unformatted, you might skip performance but at least do SMART.
* Automation:
* Identify the internal storage drive. If the Windows To Go is running from USB, the internal disk might be offline. Use Get-PhysicalDisk | Where BusType -ne USB to find non-USB disk (the internal). Or use smartctl to list drives.
* Run smartctl health check and parse output in PowerShell (capturing the console output strings).
* Run DiskSpd tests. DiskSpd outputs to console; use Invoke-Expression or direct execution and capture the output text. You may parse key lines (throughput). Alternatively, DiskSpd can output XML (-X flag) which you can load in PowerShell and extract metrics.
* No user input needed; these can run automatically. Just ensure not to run too long to delay the process (20-30 seconds per test is fine).
* Logging: Summarize the SMART health: e.g. “SMART Status: PASSED (no errors reported) [smartmontools.org](https://www.smartmontools.org/) ” or “SMART Warning: ReallocatedSectors=5 (drive may be failing)”. List any abnormal SMART attributes. Then list performance results, for instance: “Disk Read/Write: 480 MB/s read, 450 MB/s write (SSD OK) [en.wikipedia.org](https://en.wikipedia.org/wiki/DiskSpd). If performance is significantly below spec, flag it (“HDD performing below expected speed”). All these details go into the report.

# 8- Camera Functionality Test

**Objective:** Confirm the integrated webcam operates.

**Tool/Method:** Use a script or utility to access the camera:

* OpenCV (Python): An open-source approach is to use OpenCV. A Python script can attempt to open the default camera (cv2.VideoCapture(0)) and grab a frame [geeksforgeeks.org](https://www.geeksforgeeks.org/python-opencv-capture-video-from-camera/)[reddit.com](https://www.reddit.com/r/Python/comments/cpymni/7_lines_of_python_code_to_show_your_webcam_in_a/). If successful, save it as an image file (e.g., camera\_test.jpg). The script can analyze the frame to ensure it's not just black (for example, check if the average brightness or variance is above a threshold, indicating that an image was captured).
* ffmpeg: Alternatively, if including ffmpeg, use a one-liner to grab a snapshot from DirectShow camera. For example:
* shell
* ffmpeg -y -f dshow -video\_size 640x480 -i "video=Integrated Camera" -frames 1 camera.jpg
* This requires knowing the exact camera name. You can find it via ffmpeg -list\_devices true -f dshow -i dummy. If ffmpeg is too heavy, stick to OpenCV or even the Windows built-in Camera app (though that isn’t scriptable easily).
* PowerShell WIA: There's a Windows Image Acquisition (WIA) COM interface that could potentially capture from a camera if it's exposed, but that is complex. Given we have Python, OpenCV is straightforward and open-source.
* Automation:
* The script initiates the camera test by calling the Python script or ffmpeg.
* Wait for it to complete (with a timeout to avoid hanging if camera not present).
* Check if an image file was created and its file size. No image or an error exit code indicates camera failure.
* If an image is there, you might open it briefly or just trust it. Optionally, use image processing to verify it's not blank: load the image and compute average pixel value. If it's all zeros or one color, maybe the lens is covered or it failed, but that's an edge case.
* Logging: If successful, note “Webcam captured image successfully.” If not, report “Camera test failed – no image” or “Camera not detected.” If multiple cameras (some laptops have IR camera too), you could test the first one. At minimum, confirming the primary front camera works is enough. The saved image can be kept in the report folder for reference.

# 9. Windows Autopilot Readiness Check

**Objective:** Determine if the device is ready to be enrolled in Windows Autopilot (for streamlined deployment by the new owner). Specifically, gather its Autopilot **hardware hash** and ensure no roadblocks for enrollment.

**Tool/Method:** Use Microsoft’s provided PowerShell script for Autopilot:

* Get-WindowsAutoPilotInfo.ps1: This script (available from PowerShell Gallery) retrieves the device’s hardware hash (a unique identifier required for Autopilot registration) and other details [learn.microsoft.com](https://learn.microsoft.com/en-us/autopilot/add-devices). We have it downloaded in the Tools.
* Automation:
* Run the script in offline mode to output a CSV with the hash. Example command:
* powershell
* Set-ExecutionPolicy Bypass -Scope Process -Force
* .\Get-WindowsAutoPilotInfo.ps1 -OutputFile "$env:TEMP\AutoPilot.csv"
* This will use WMI to get the hash and store it locally [learn.microsoft.com](https://learn.microsoft.com/en-us/autopilot/add-devices). (No internet needed unless using -Online to directly register, which we won’t do).
* The CSV will contain hardware hash, serial, etc. Import that CSV in PowerShell to get the info, or simply attach it to the report.
* Additional readiness: If the device still has Windows installed (the internal drive), it might already be registered to an organization’s Autopilot. There’s no easy way to check that offline, but if you had internet, you could attempt an online sync (out of scope due to no cloud). However, providing the hash allows later checking in Microsoft’s portal.
* You might also check TPM availability (Autopilot doesn’t require TPM per se, but Win11 does). Use get-tpm to see if TPM is present and activated, as a forward-looking note (for Windows 11 readiness).
* Logging: Indicate that Autopilot hardware hash was retrieved. For example: “Autopilot Hardware Hash: SUCCESS – CSV file generated.” You could include the serial number of the device (from the script output) in the report to cross-check device identity. If the script fails (WMI issues), note that “Autopilot hash could not be obtained.” Usually, it should work on Windows 10/11.
* Autopilot Readiness: Summarize any other factors: e.g., “Device meets Windows Autopilot requirements” or if you checked TPM: “TPM 2.0 Present: Yes”. This ensures the reseller knows if the device can be managed via Autopilot easily.

# Report Generation and Export

After all tests, the script will have accumulated a wealth of data. Organize and present it clearly:

* Use an HTML report for a nicely formatted output (PowerShell can concatenate strings or use here-strings to build HTML with sections for each test). Include a summary at the top: Pass/Fail for each category, or any critical failures highlighted in red.
* Embed key details: e.g., battery health %, disk SMART status, etc., in a table or list form for readability. Preserve the Article ID and device identifying info at top.
* Save the report as DeviceID\_Report.html on the USB (and perhaps also on the internal disk for record, if not wiped).
* For PDF export, if you included a tool like wkhtmltopdf, you can call it on the HTML:

***shell***

wkhtmltopdf DeviceID\_Report.html DeviceID\_Report.pdf

* This creates a PDF copy. Alternatively, instruct that the HTML can be opened in a browser and printed to PDF if automated conversion isn’t available. Since no cloud allowed, all conversion must be local (wkhtmltopdf is open-source and can be used offline).
* Ensure the report (HTML/PDF) is accessible to the technician for review and can be copied off the USB for record-keeping.

# Sources

1. Windows To Go dual-boot setup [elevenforum.com](https://www.elevenforum.com/t/installation-using-media-creation-tool-uefi-issue.22632/);
2. Keyboard tester utility [github.com](https://github.com/10yard/keyboardtester);
3. DiskSpd open-source disk tool [en.wikipedia.org](https://en.wikipedia.org/wiki/DiskSpd);
4. smartmontools for drive health [smartmontools.org](https://www.smartmontools.org/);
5. Battery health criteria [powershell.one](https://powershell.one/wmi/root/cimv2/win32_battery);
6. Wi-Fi auto-connect script [reddit.com](https://www.reddit.com/r/Intune/comments/17p5ia3/most_efficient_way_of_collecting_hardware_hashes/);
7. Autopilot readiness script [learn.microsoft.com](https://learn.microsoft.com/en-us/autopilot/add-devices).

|  |  |
| --- | --- |
| **7 Tests to Run Before Buying a Used Windows PC or Laptop** | <https://www.makeuseof.com/windows-used-machine-tests/> |
| **7 things you need to check before buying a used laptop or desktop pc** | <https://www.slashgear.com/1709875/things-to-check-before-buying-used-laptop-desktop-pc/> |
| **20 Things to Check When Buying a Used Computer [2024 Guide 9]** | <https://10scopes.com/check-when-buying-a-used-computer/> |
| **How to Inspect a Used Windows PC Before You Buy It** | <https://www.maketecheasier.com/inspect-used-windows-pc/> |