**How It Works**

* **When "BEST" Is Appended**:
  + If the "Move Highest Quality Duplicate (Size & Bitrate)" checkbox ($useHighestQuality) is checked and a group (similarity or file group) contains more than one file ($duplicateCount -gt 1), the script selects the file with the largest size and highest bitrate.
  + This file’s destination filename is modified to include "\_BEST" (e.g., SongName\_BEST.mp3).
  + Other files in the group (duplicates) retain their original filenames and are moved to the "Duplicates" folder (for similarity groups) or skipped (for file groups).
* **When "BEST" Is Not Appended**:
  + If $useHighestQuality is $false or the group has only one file, the filename remains unchanged (no "\_BEST" suffix).
  + The file is processed as before, moved to its designated folder without modification.
* **Sanitization**:
  + The Sanitize-Filename function is called on the modified filename to ensure it’s safe for the filesystem, handling cases where "BEST" might introduce invalid characters (though unlikely).

**Testing the Change**

1. **Setup**:
   * Save the script as MusicFileOrganizer.ps1.
   * Create a source directory with multiple audio files, including duplicates with different sizes and bitrates (e.g., Song.mp3 at 128kbps/2MB, Song.mp3 at 320kbps/5MB).
2. **Run the Script**:
   * Open the script in PowerShell and run it (e.g., .\MusicFileOrganizer.ps1).
   * In the UI, select the source and destination directories.
   * Check the "Move Highest Quality Duplicate (Size & Bitrate)" checkbox.
   * Optionally enable "Check for Duplicated Songs by Name" to test similarity groups.
   * Click "Start".
3. **Verify Output**:
   * In the destination directory, check that the highest quality file in each duplicate group has "\_BEST" in its filename (e.g., Song\_BEST.mp3).
   * For similarity groups, other duplicates should be in the "Duplicates" subfolder with their original names.
   * For file groups, only the best file should be moved, with "\_BEST" appended, and others skipped.
   * Check the HTML log file (MusicFileOrganizer\_Log\_\*.html) for messages confirming the selection of highest quality files and their new paths.
4. **Debugging**:
   * Use the breakpoints (e.g., lines 899 and 993) in VS Code or PowerShell ISE to step through the similarity and file group processing.
   * Inspect $destFilename, $bestFile, and $isBestFile (for similarity groups) or $entry.File.FullName -eq $bestFile.File.FullName (for file groups) to confirm that "\_BEST" is applied correctly.
   * Verify that $destFile reflects the modified filename in the log messages.

**Example Scenario**

* **Source Files**:
  + Song.mp3 (Size: 2MB, Bitrate: 128kbps)
  + Song\_copy.mp3 (Size: 5MB, Bitrate: 320kbps)
* **Settings**:
  + "Move Highest Quality Duplicate" checked.
  + "Check for Duplicated Songs by Name" checked with a low similarity threshold (e.g., 1).
* **Output**:
  + Destination: <dest>\Song\Song\_BEST.mp3 (the 5MB/320kbps file).
  + Duplicates: <dest>\Duplicates\Song\Song.mp3 (the 2MB/128kbps file).
* **Log**:
  + "Selected highest quality file: Song\_copy.mp3 (Size: 5242880 bytes, Bitrate: 320000 bps)"
  + "Successfully moved: Song\_copy.mp3 -> \Song\Song\_BEST.mp3"
  + "Successfully moved duplicate: Song.mp3 -> \Duplicates\Song\Song.mp3"

**Notes**

* **Sanitization**: The Sanitize-Filename function ensures that adding "\_BEST" doesn’t create invalid filenames. If the resulting filename exceeds 200 characters, it’s truncated, but this is unlikely with typical filenames.
* **Duplicates in File Groups**: For file groups, only the best file is moved when $useHighestQuality is $true, and others are skipped (logged but not moved). If you want duplicates moved to a "Duplicates" folder like similarity groups, further modifications would be needed.
* **Breakpoints**: The updated line numbers in Set-DebugBreakpoints ensure debugging remains accurate. If you modify the script further, verify these line numbers.
* **Edge Cases**:
  + If multiple files have identical size and bitrate, the first one sorted is chosen as the best, and it gets the "\_BEST" suffix.
  + If $useHighestQuality is $false, no files get "\_BEST", and all files are processed normally.

**Guide to Editing the Script**

**1. Improve Filename Logic**

The filename logic in the script processes audio file names by removing specified terms (e.g., "Official Video"), cleaning spaces, adding custom text, and sanitizing filenames. To enhance this:

* **Current Logic**: In the **Start** button’s event handler (lines ~934–947), filenames are processed as follows:
  + Extract base name without extension (GetFileNameWithoutExtension).
  + Apply space cleanup if enabled (Clean-FilenameSpaces).
  + Remove terms from $removeList (e.g., "Lyrics").
  + Add terms from $addList (sanitized).
  + Sanitize the final filename (Sanitize-Filename).
* **Issues**:
  + Limited flexibility for complex naming patterns (e.g., extracting artist/title from metadata).
  + No support for regex-based replacements beyond simple term removal.
  + $addList appending can create unwieldy names if multiple terms are added.
* **Improvements**:
  + Add regex-based replacement for advanced pattern matching (e.g., remove dates like "2023" or "(Live)").
  + Use audio metadata (e.g., ID3 tags for MP3) to construct filenames (artist - title).
  + Allow case-sensitive term removal for precision.
* **How to Edit**:
  + **Location**: Modify the filename processing in the **Start** button’s Add\_Click event (lines ~934–947).
  + **Steps**:
    1. Add a regex replacement list to $removeList for patterns like dates or parenthetical text.
    2. Integrate metadata extraction using a library like TagLibSharp (already assumed available via Install-Dependencies).
    3. Make term removal case-sensitive by adjusting the -replace operation.

**Example Change**:

powershell

Kopiëren

*# Original (line ~934–947)*

$filename = [System.IO.Path]::GetFileNameWithoutExtension($file.FullName)

if ($cleanSpaces) {

$filename = Clean-FilenameSpaces -Filename $filename

}

foreach ($term in $removeList) {

$filename = $filename -replace [regex]::Escape($term), ""

}

$filename = $filename -replace '\s+', ' '

$filename = $filename.Trim()

if ($addList) {

$filename = "$filename $($addList -join ' ')"

}

$sanitizedFilename = Sanitize-Filename -Filename $filename

*# Improved*

$filename = [System.IO.Path]::GetFileNameWithoutExtension($file.FullName)

if ($cleanSpaces) {

$filename = Clean-FilenameSpaces -Filename $filename

}

*# Add regex patterns to removeList*

$removePatterns = $removeList + @('\(\d{4}\)', '\[Live\]') *# Remove (YYYY), [Live]*

foreach ($pattern in $removePatterns) {

$filename = $filename -replace $pattern, "" *# Case-sensitive*

}

*# Use metadata if available*

try {

$tagFile = [TagLib.File]::Create($file.FullName)

if ($tagFile.Tag.Artists -and $tagFile.Tag.Title) {

$filename = "$($tagFile.Tag.Artists -join ', ') - $($tagFile.Tag.Title)"

}

$tagFile.Dispose()

} catch {

Write-Log -Message "Failed to read metadata for $($file.FullName): $\_" -Status "Warning"

}

$filename = $filename -replace '\s+', ' '

$filename = $filename.Trim()

if ($addList) {

$filename = "$filename $($addList[0])" *# Use only first add term to avoid clutter*

}

$sanitizedFilename = Sanitize-Filename -Filename $filename

* + **Impact**: Enhances filename accuracy by leveraging metadata and regex, reduces clutter from $addList. Test with files having ID3 tags and names like "Song (2023) [Live].mp3".

**2. Change Priority**

The script prioritizes audio files when selecting the "best" file (if $useHighestQuality is enabled) based on FormatPriority, Bitrate, and Size (lines ~1178–1190). The current Get-FormatPriority function assigns priorities as follows:

* FLAC: 1, WAV/AIFF: 2, MP3: 3, OGG: 4, WMA: 5.
* **Issues**:
  + Fixed priorities may not suit all users (e.g., preferring MP3 for compatibility).
  + No user-configurable priority via GUI.
* **How to Edit**:
  + **Location**: Modify Get-FormatPriority (lines ~300–320, assumed) and add a GUI control for priority selection.
  + **Steps**:
    1. Update Get-FormatPriority to use a configurable priority map.
    2. Add a dropdown (ComboBox) to the GUI for users to select a priority preset.
  + **Example Change**:

powershell

Kopiëren

*# Original Get-FormatPriority (assumed)*

function Get-FormatPriority {

param ([string]$Extension)

$priorityMap = @{

".flac" = 1

".wav" = 2

".aiff" = 2

".mp3" = 3

".ogg" = 4

".wma" = 5

}

return $priorityMap[$Extension.ToLower()] ?? 10

}

*# Updated Get-FormatPriority*

function Get-FormatPriority {

param ([string]$Extension)

$script:priorityMap = $script:priorityMap ?? @{

"Lossless" = @{ ".flac" = 1; ".wav" = 2; ".aiff" = 2; ".mp3" = 3; ".ogg" = 4; ".wma" = 5 }

"MP3First" = @{ ".mp3" = 1; ".flac" = 2; ".wav" = 3; ".aiff" = 3; ".ogg" = 4; ".wma" = 5 }

}

$selectedPriority = $priorityComboBox.SelectedItem ?? "Lossless"

return $script:priorityMap[$selectedPriority][$Extension.ToLower()] ?? 10

}

*# Add ComboBox to GUI (before Start button, e.g., line ~600)*

$priorityLabel = New-Object System.Windows.Forms.Label

$priorityLabel.Location = New-Object System.Drawing.Point(30, 550)

$priorityLabel.Size = New-Object System.Drawing.Size(150, 20)

$priorityLabel.Text = "Format Priority:"

$form.Controls.Add($priorityLabel)

$priorityComboBox = New-Object System.Windows.Forms.ComboBox

$priorityComboBox.Location = New-Object System.Drawing.Point(180, 550)

$priorityComboBox.Size = New-Object System.Drawing.Size(150, 20)

$priorityComboBox.Items.AddRange(@("Lossless", "MP3First"))

$priorityComboBox.SelectedIndex = 0

$form.Controls.Add($priorityComboBox)

* + **Impact**: Allows users to choose between prioritizing lossless formats or MP3 via GUI. Test by selecting "MP3First" and verifying MP3 files are chosen as the best.

**3. Add File Extensions to Process**

The script processes .mp3, .wav, .flac, .aiff, .ogg, and .wma files (line ~857 in Get-FilesIncrementally).

* **How to Edit**:
  + **Location**: Update the -Include parameter in Get-FilesIncrementally call (line ~857).
  + **Steps**:
    1. Add new extensions (e.g., .m4a, .aac) to the array.
    2. Update Get-FormatPriority to assign priorities for new extensions.
  + **Example Change**:

powershell

Kopiëren

*# Original (line ~857)*

$fileCount = Get-FilesIncrementally -Path $source -Include @("\*.mp3", "\*.wav", "\*.flac", "\*.aiff", "\*.ogg", "\*.wma") -BatchSizeBytes $batchSizeBytes -BatchCallback { ... }

*# Updated*

$fileCount = Get-FilesIncrementally -Path $source -Include @("\*.mp3", "\*.wav", "\*.flac", "\*.aiff", "\*.ogg", "\*.wma", "\*.m4a", "\*.aac") -BatchSizeBytes $batchSizeBytes -BatchCallback { ... }

*# Update Get-FormatPriority (line ~300)*

function Get-FormatPriority {

param ([string]$Extension)

$script:priorityMap = $script:priorityMap ?? @{

"Lossless" = @{ ".flac" = 1; ".wav" = 2; ".aiff" = 2; ".mp3" = 3; ".ogg" = 4; ".wma" = 5; ".m4a" = 3; ".aac" = 3 }

"MP3First" = @{ ".mp3" = 1; ".flac" = 2; ".wav" = 3; ".aiff" = 3; ".ogg" = 4; ".wma" = 5; ".m4a" = 2; ".aac" = 2 }

}

$selectedPriority = $priorityComboBox.SelectedItem ?? "Lossless"

return $script:priorityMap[$selectedPriority][$Extension.ToLower()] ?? 10

}

* + **Impact**: Expands file type support. Test with .m4a and .aac files to ensure they are processed and prioritized correctly.

**4. Change Buffer Size**

The buffer size for file copying is fixed at 16MB ($bufferSizeBytes = 16MB, line ~784).

* **How to Edit**:
  + **Location**: Modify the $bufferSizeBytes assignment in the **Start** button’s Add\_Click event (line ~784).
  + **Steps**:
    1. Change 16MB to a different value (e.g., 32MB for faster copying on high-performance systems).
    2. Optionally, add a GUI textbox for user input, similar to $batchTextBox.
  + **Example Change**:

powershell

Kopiëren

*# Original (line ~784)*

$bufferSizeBytes = 16MB

*# Updated*

$bufferSizeBytes = 32MB *# Increased for faster copying*

*# Optional: Add GUI textbox (before Start button, e.g., line ~590)*

$bufferLabel = New-Object System.Windows.Forms.Label

$bufferLabel.Location = New-Object System.Drawing.Point(30, 530)

$bufferLabel.Size = New-Object System.Drawing.Size(150, 20)

$bufferLabel.Text = "Buffer Size (MB):"

$form.Controls.Add($bufferLabel)

$bufferTextBox = New-Object System.Windows.Forms.TextBox

$bufferTextBox.Location = New-Object System.Drawing.Point(180, 530)

$bufferTextBox.Size = New-Object System.Drawing.Size(100, 20)

$bufferTextBox.Text = "32"

$form.Controls.Add($bufferTextBox)

*# Update Start button to use textbox (line ~784)*

$bufferSizeMB = 32

try {

$bufferSizeMB = [double]$bufferTextBox.Text

if ($bufferSizeMB -lt 1 -or $bufferSizeMB -gt 128) {

Write-Log -Message "Error: Buffer size must be between 1 and 128 MB." -Status "Error"

$startButton.Enabled = $true

$stopButton.Enabled = $false

return

}

} catch {

Write-Log -Message "Error: Invalid buffer size '$($bufferTextBox.Text)'. Using default 32 MB." -Status "Warning"

$bufferSizeMB = 32

}

$bufferSizeBytes = [int64]($bufferSizeMB \* 1MB)

* + **Impact**: Increases copying speed for large files but may raise memory usage. Test with a 100MB .flac file to compare performance.

**5. Recommended Buffer Size and Batch Size for Audio Files**

* **Buffer Size**:
  + **Current**: 16MB, optimized for 20MB–100MB audio files.
  + **Recommendation**:
    - **8MB**: For low-memory systems (e.g., <8GB RAM) or small files (<20MB).
    - **16MB**: Balanced default for typical audio files (20MB–100MB) and most systems.
    - **32MB**: For high-performance systems (16GB+ RAM) and large files (>100MB).
    - **64MB**: For very large files (>500MB, e.g., high-resolution WAV) on systems with 32GB+ RAM.
  + **Rationale**: Larger buffers reduce disk I/O overhead but increase memory usage. Audio files are typically 20MB–100MB, so 16MB–32MB is ideal for most use cases.
* **Batch Size**:
  + **Current**: 300MB, user-configurable via $batchTextBox (10MB–1000MB).
  + **Recommendation**:
    - **100MB**: For low-memory systems or small datasets (<10GB).
    - **300MB**: Default for typical datasets (10GB–60GB) and 8GB+ RAM.
    - **500MB**: For large datasets (>60GB) on systems with 16GB+ RAM.
    - **1000MB**: For very large datasets (>100GB) on high-end systems (32GB+ RAM).
  + **Rationale**: Batch size controls memory usage during file enumeration and processing. Smaller batches reduce memory spikes, while larger batches improve throughput for big datasets.
* **How to Implement**:
  + Update comments or add a GUI tooltip to guide users on recommended sizes.
  + Example comment in **Start** button (line ~784):

powershell

Kopiëren

*# Buffer size: 8MB (low memory), 16MB (default, 20MB–100MB files), 32MB (high performance), 64MB (large files, 32GB+ RAM)*

$bufferSizeBytes = 32MB

*# Batch size: 100MB (low memory), 300MB (default, 10GB–60GB), 500MB (large datasets), 1000MB (100GB+, 32GB+ RAM)*

$batchSizeMB = 300

**User Manual**

**Music File Organizer User Manual**

**Purpose**: Organizes audio files into folders based on sanitized filenames, with options to handle duplicates, prioritize quality, and customize naming.

**Requirements**:

* Windows OS with PowerShell 5.1 or later.
* .NET Framework for Windows Forms.
* TagLibSharp for metadata (installed via script).

**Installation**:

1. Save MusicFileOrganizer.ps1 to a directory.
2. Run in PowerShell: .\MusicFileOrganizer.ps1.
3. The script installs dependencies (TagLibSharp) if needed.

**Usage**:

1. **Launch**: Run the script to open the GUI.
2. **Configure**:
   * **Source Folder**: Select the folder containing audio files (e.g., .mp3, .flac).
   * **Destination Folder**: Choose where organized files will be saved.
   * **Backup Folder** (optional): Set a folder for backups.
   * **Replacements File** (optional): Provide a .txt file with comma-separated terms to remove from filenames.
   * **Add Text File** (optional): Provide a .txt file with terms to append to filenames.
   * **Highest Quality**: Check to keep only the best file (based on format, bitrate, size) for duplicates.
   * **Duplicate Names**: Check to group similar filenames (adjust similarity via slider, 0–100).
   * **Clean Spaces**: Check to normalize spaces in filenames.
   * **Batch Size**: Enter size in MB (default: 300, range: 10–1000).
   * **Format Priority** (if added): Select priority preset (e.g., "Lossless" or "MP3First").
3. **Start**: Click **Start** to process files.
4. **Stop**: Click **Stop** to cancel processing (saves partial logs).
5. **Exit**: Click **Exit** to save logs and close.

**Outputs**:

* Organized files in destination folder, each in a subfolder named after the sanitized filename.
* Batch logs: MusicFileOrganizer\_Log\_<timestamp>\_batchN.html.
* Summary log: MusicFileOrganizer\_Summary\_Log\_<timestamp>.html.

**Troubleshooting**:

* **Error on Start**: Ensure source/destination paths are valid and writable.
* **Slow Processing**: Reduce batch size (e.g., 100MB) or buffer size (e.g., 8MB) for low-memory systems.
* **Message Loop Error**: Avoid running multiple instances; ensure dialogs close properly.

**Function Overview**

| **Function** | **Purpose** | **Key Parameters** | **Location (Line)** |
| --- | --- | --- | --- |
| Set-DebugBreakpoints | Sets breakpoints for debugging key script points. | None | ~148–171 |
| Install-Dependencies | Installs required libraries (e.g., TagLibSharp). | None | ~172–200 |
| Write-Log | Logs messages to $script:logEntries and console. | -Message, -Status (Info, Warning, Error, Success) | ~201–220 |
| Update-UIStatus | Updates GUI status label and progress bar. | -Message, -ProgressValue | ~221–240 |
| Clean-FilenameSpaces | Normalizes spaces in filenames. | -Filename | ~241–260 |
| Sanitize-Filename | Removes invalid characters from filenames. | -Filename | ~261–280 |
| Get-StringSimilarity | Calculates similarity between two strings for duplicate detection. | -String1, -String2, -SliderValue, -RemoveList | ~281–300 |
| Get-FormatPriority | Assigns priority to audio formats for best file selection. | -Extension | ~301–320 |
| Get-FileBitrate | Retrieves bitrate from audio file metadata. | -FilePath | ~321–340 |
| Test-FileAccessible | Checks if a file is accessible (not locked). | -Path | ~341–360 |
| Get-UniqueFolderPath | Generates a unique folder path to avoid conflicts. | -FolderPath | ~361–380 |
| Create-Backup | Creates a backup of the source folder. | -Source, -BackupDir, -BufferSizeBytes | ~381–410 |
| Copy-LargeFile | Copies files with a specified buffer size, supports cancellation. | -SourcePath, -DestPath, -BufferSizeBytes | ~518–536 |
| Get-FilesIncrementally | Enumerates files in batches to manage memory. | -Path, -Include, -BatchSizeBytes, -BatchCallback | ~537–614 |
| Generate-HtmlLog | Generates HTML log for a batch. | -LogFilePath, -BatchNumber | ~1200–1220 |
| Generate-SummaryLog | Generates a summary HTML log linking batch logs. | -SummaryLogFilePath, -BatchLogFiles | ~1221–1240 |

**Example Script Changes**

Below is a partial script incorporating the above edits (filename logic, priority, extensions, buffer size). Only changed sections are shown for brevity.

MusicFileOrganizer.ps1

powershell

Toon inline

**Testing Notes**

* **Filename Logic**: Test with files like "Song (2023) [Live].mp3" and verify output names use metadata (e.g., "Artist - Title.mp3") or remove patterns correctly.
* **Priority**: Switch between "Lossless" and "MP3First" in the GUI, process duplicate files, and confirm the correct format is chosen as the best.
* **Extensions**: Include .m4a and .aac files in the source folder and verify they are processed and logged.
* **Buffer Size**: Test with bufferTextBox set to 8MB, 32MB, and 64MB, monitoring memory usage and copy speed for a 100MB file.
* **Recommendations**: Process a 60GB dataset with 300MB batch and 32MB buffer, then try 100MB batch and 8MB buffer on a low-memory system to compare stability.

This guide and manual provide clear steps to customize the script and use it effectively. Let me know if you need further clarification or additional features!