How to Easily Delete Folders with VBA: A Step-by-Step Guide

Authored by

stats writer

November 19, 2025

Visual Basic for Applications (VBA) is an indispensable tool for automating repetitive tasks within Microsoft Office suite applications. When managing large datasets or structured projects, the need often arises to programmatically clean up or reorganize directory structures. This article provides a comprehensive guide on how to utilize macros to delete folders and their contents efficiently and safely. While methods involving the dedicated FileSystemObject exist--specifically the **DeleteFolder method**, which requires specifying the absolute folder path as an argument--we will primarily focus on native VBA statements like **Kill** and **RmDir**, which offer a streamlined approach for common deletion scenarios. Understanding these fundamental techniques allows developers to create robust automation routines for both file and folder management.

The process of deleting a folder involves several steps, depending on whether the folder is empty or contains files. VBA provides powerful, yet simple, statements to handle both situations. The key difference between deleting a file and deleting a folder in VBA is that directories must typically be empty before they can be removed using native statements. If a folder contains files, those files must be purged first. We will explore two primary methods here: one focused solely on clearing the contents of a directory, and another that combines file deletion with the subsequent removal of the now-empty directory itself.

It is critical to note that deleting files and folders via VBA is a permanent action; these items are generally not moved to the Recycle Bin. Therefore, implementing robust error handling is paramount to prevent runtime errors and ensure that the macro executes reliably, especially when dealing with dynamic paths or user inputs. We will examine how to incorporate the **On Error Resume Next** statement to gracefully manage potential issues, such as attempting to delete a folder that does not exist.

Choosing the Right Deletion Method in VBA

When preparing to remove directories and their contents using Visual Basic for Applications (VBA), developers typically rely on one of two main strategies. The first involves utilizing built-in VBA statements, specifically the **Kill statement** for file deletion and the **RmDir statement** for folder removal. This approach is straightforward and does not require setting object references or linking external libraries. The second strategy, often favored for more complex file operations, involves the **Microsoft Scripting Runtime Library** and its associated FileSystemObject. For the purpose of quick and effective folder removal shown in the subsequent examples, we will prioritize the native VBA statements.

These native methods provide powerful yet granular control over the deletion process. Unlike methods that delete a folder and all its contents recursively in a single command, the native approach forces the developer to handle file and folder removal sequentially. This distinction is crucial for maintaining control and ensuring that cleanup processes are executed exactly as

intended, particularly in environments where only certain file types or specific directory levels need modification. We outline the two core methodologies presented in this guide below, focusing on the commands that will be utilized in the accompanying code examples.

Method 1: Deleting Only the Contents of a Folder using the Kill Statement

The Kill statement in VBA is designed specifically for deleting files from a specified location on the disk. It is highly effective for clearing out a directory without affecting the directory structure itself. To achieve comprehensive deletion of all contents, the statement accepts wildcard characters, which allows the macro to target all files, regardless of their name or extension. This method is ideal for cleanup routines where the folder itself must persist for future data storage, acting as a temporary staging location that needs periodic emptying.

When implementing this method, the developer must specify the full path to the directory and append the file pattern `*.*`. This pattern instructs the Kill statement to match every file name (represented by the first asterisk, `*`) and every file extension (represented by the second asterisk, `*`). The following sample code demonstrates how this technique is applied to purge all files within a designated folder, named **My_Data** in this context, incorporating essential error control.

```
Sub DeleteFolderContents()

On Error Resume Next
Kill "C:UsersbobbiDesktopMy_Data*.*"
On Error GoTo 0

End Sub
```

This particular macro uses the Kill statement to successfully delete all files residing in the folder located at `C:UsersbobbiDesktopMy_Data`. It is a critical distinction that this code only addresses files; if the **My_Data** folder contained subfolders, those subfolders would remain untouched, along with any files they contained. This isolation ensures that only the intended level of the file system hierarchy is affected by the operation, offering a controlled way to manage directory contents without risking the deletion of important nested structures.

Method 2: Deleting the Entire Folder using Kill and RmDir

To completely remove a directory, including the folder container itself, a two-step approach using native VBA statements is strictly required. First, all files within the directory must be deleted using the Kill statement, rendering the folder empty. Once the directory is empty, the RmDir statement (short for Remove Directory) can be invoked to eliminate the folder structure itself. Attempting to use **RmDir** on a folder that still contains files or subdirectories will immediately result in a runtime

error 75 ("Path/File access error"), underscoring the necessity of the preceding cleanup step.

This combined method is necessary because RmDir is inherently restricted to removing only empty directories in the context of native VBA. Therefore, the implementation must first utilize the wildcard pattern with **Kill** to clear the folder's contents, and then immediately execute RmDir on the exact, full folder path. The inclusion of careful error handling, as demonstrated in the code below, is highly recommended to manage scenarios where the path might be invalid, the folder might already be gone, or access might be denied.

```
Sub DeleteFolder()

On Error Resume Next

'delete all files in folder
Kill "C:UsersbobbiDesktopMy_Data*.*"

'delete empty folder
RmDir "C:UsersbobbiDesktopMy_Data"

On Error GoTo 0

End Sub
```

Executing this macro systematically deletes all files within the **My_Data** directory using the Kill statement, followed by the complete removal of the directory itself using RmDir. The result is that the folder specified by the absolute path `C:UsersbobbiDesktopMy_Data` will no longer exist in the file system. This method is widely used for temporary directory cleanup or when a project structure needs to be fully decommissioned and removed from the operating environment.

Implementing Robust Error Handling with On Error Statements

Given the potentially destructive and irreversible nature of file operations, reliable VBA error handling is essential. When dealing with directory deletion, errors frequently occur if the specified path does not exist, if the macro lacks the necessary permissions, or if the directory still contains hidden or locked files. To prevent the macro from halting abruptly and displaying a cryptic runtime error message to the user, we employ the **On Error Resume Next** statement.

The line **On Error Resume Next** instructs the VBA execution environment to ignore any error that occurs and simply move on to the next line of code. This is particularly useful in cleanup operations where it is acceptable if, for example, the folder to be deleted is already missing or the **Kill statement** is executed on an already empty directory. Without this statement, if the Kill statement or RmDir statement cannot find the specified path, the macro would terminate immediately,

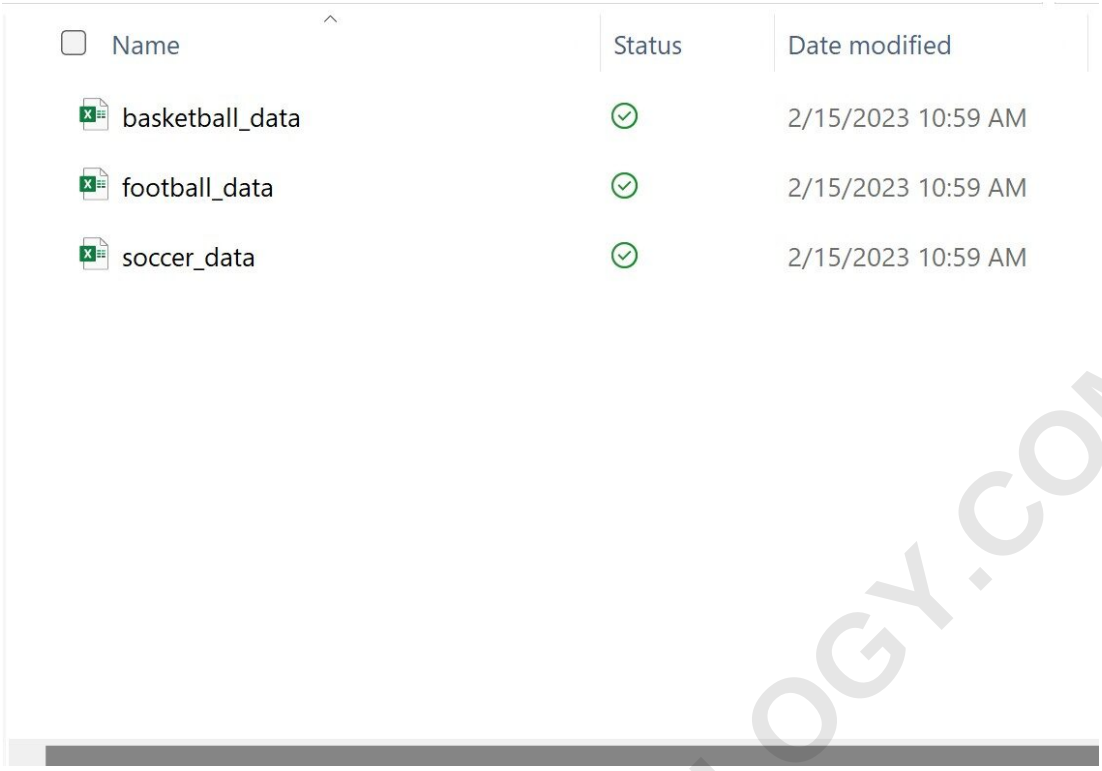
requiring manual debugging or user intervention. Using this approach ensures a smooth, uninterrupted execution flow, regardless of minor discrepancies in the environment state.

It is considered best practice to disable the permissive error handling once the critical file operations are complete. This is achieved using the command **On Error GoTo 0**. This statement resets the default error message settings, ensuring that any errors encountered in subsequent parts of the larger subroutine--outside the file deletion block--will trigger the standard VBA error handling mechanisms. This localized use of **On Error Resume Next** maintains safety and transparency for the rest of the procedure. Should a developer prefer to display an explicit error message if a file or folder is not found, then these two error handling lines should simply be removed from the code, allowing the standard runtime error dialogue to appear, or be replaced by more sophisticated error handling logic using `On Error GoTo` .

Preparing the Target Directory Structure for Deletion Examples

To clearly illustrate the effectiveness of the **Kill** and RmDir methods, we will use a hypothetical scenario involving a folder named **My_Data**. This folder is located on the user's desktop and contains several files relevant to a project. Specifically, for the purpose of these examples, we assume the **My_Data** directory currently holds three Microsoft Excel workbooks. The structure is simple, yet provides a perfect foundation for demonstrating how programmatic deletion works in practice against real files.

Before running any of the deletion macros, it is important to visualize the initial state of the directory. The following image represents the folder structure prior to executing the cleanup routines. Notice the presence of multiple files within the main directory, confirming that the folder is initially non-empty. Our goal in the first example will be to clear these files, and in the second example, to remove the entire structure itself.



Understanding the initial state is crucial for verifying the success of the VBA code. When automating file operations, developers should always ensure they have correct, absolute paths and that the target directory permissions allow for write and delete operations. If permissions are restricted, the **Kill** or RmDir statements will fail, even with **On Error Resume Next** activated, unless the macro specifically checks the error code using the `Err` object after the execution of the command.

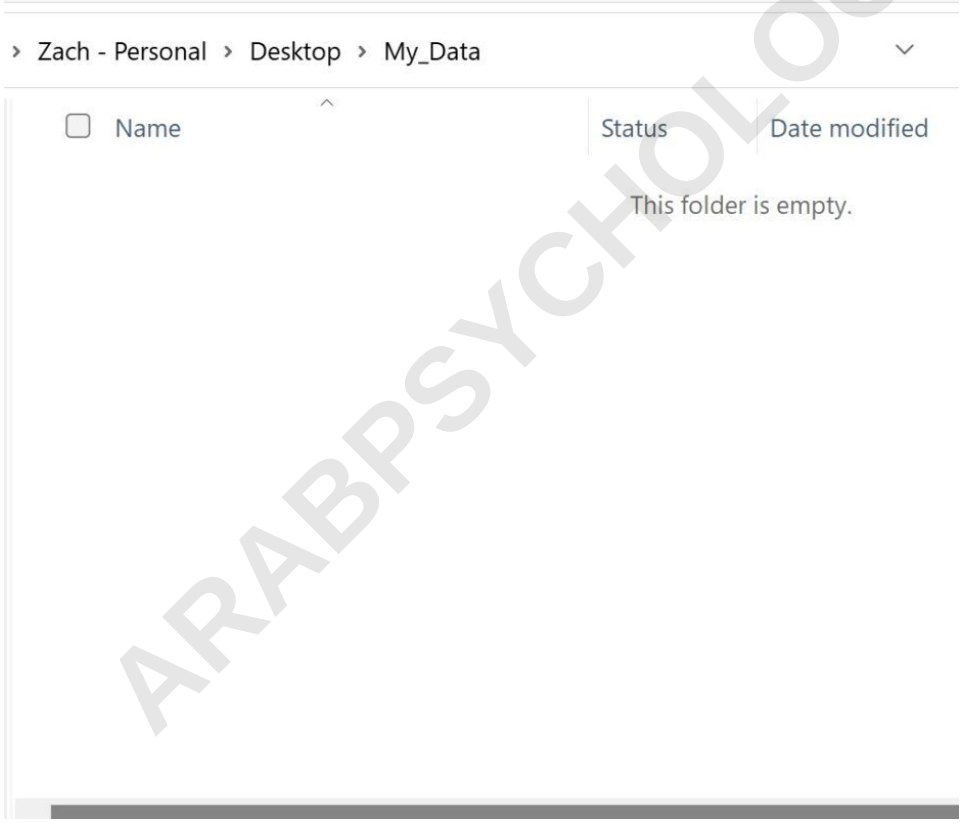
Practical Example 1: Deleting All Files within a Folder Using VBA

This first practical example demonstrates the implementation of **Method 1**, which is focused exclusively on removing the file contents of the target directory while preserving the folder container. We assume the developer needs to repeatedly refresh the data in this folder, necessitating a periodic cleanup of old files without rebuilding the entire path structure. The simplicity of the Kill statement makes this an incredibly efficient solution for purging contents based on a known, static file path.

The objective is to utilize VBA to target the **My_Data** folder and eliminate all files inside it. We create a dedicated subroutine, `DeleteFolderContents`, which contains the necessary error handling and the core **Kill** command. The use of the wildcard `*.*` ensures that every file, regardless of its specific format (e.g., .xlsx, .txt, .pdf), is deleted from the path specified in the string argument, thus achieving a complete internal purge.

```
Sub DeleteFolderContents()

On Error Resume Next
Kill "C:UsersbobbiDesktopMy_Data*.*"
On Error GoTo 0

End Sub
```

Upon execution of this macro, the VBA interpreter processes the **Kill statement**. If the folder exists, all contained files are immediately and permanently deleted. If the folder did not exist, the error would be suppressed by **On Error Resume Next**, and the macro would conclude successfully without intervention, which is often desirable in cleanup scripts. The subsequent image confirms the outcome: the folder structure remains, but all previously contained Excel files have been successfully removed, leaving the directory empty and ready for new data input.



Practical Example 2: Deleting the Entire Directory Using VBA

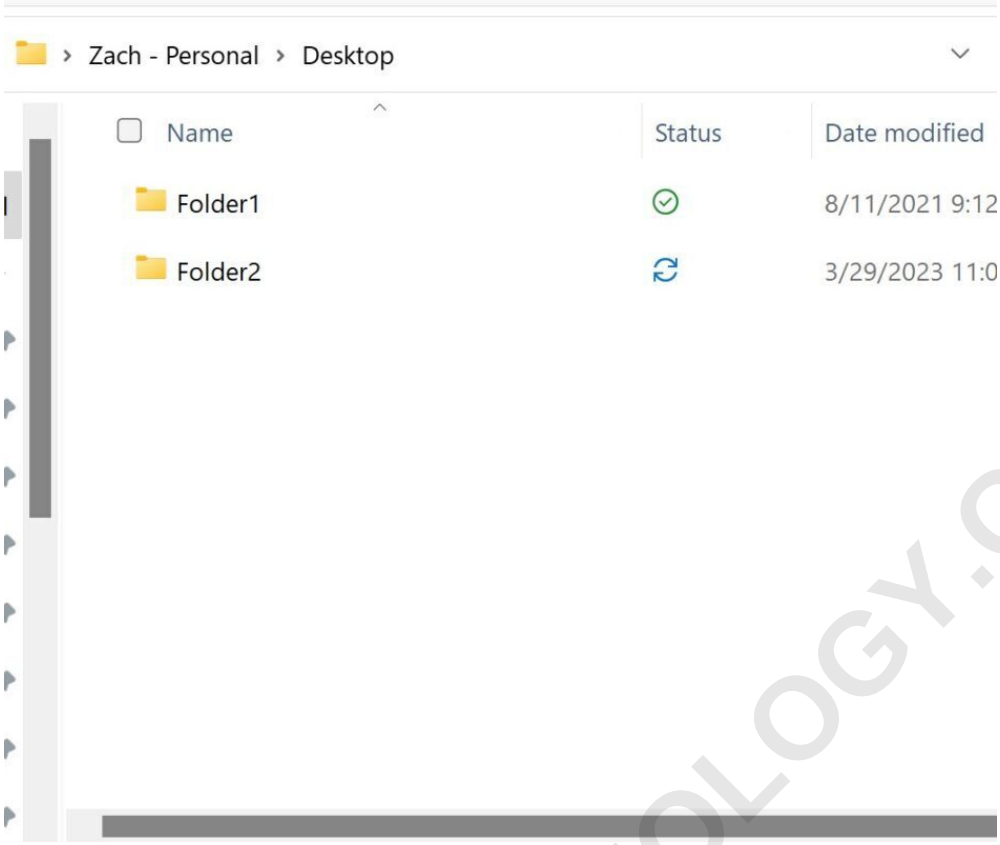
The second example addresses the scenario where the entire directory structure, including the container folder, needs to be eliminated from the file system. This requires the combined use of the Kill statement and the RmDir statement, as established in **Method 2**. This is typically necessary

when decommissioning a temporary storage location or fully cleaning up resources associated with a finished data processing pipeline. The sequence of operations is strictly enforced: file removal first, followed by the directory removal.

To perform this comprehensive deletion, we define the `DeleteFolder` subroutine. This macro sequentially executes the two required commands on the target path `C:UsersbobbiDesktopMy_Data`. Note that the **Kill statement** targets the contents using the wildcard (`*.*`), while RmDir targets the folder path directly, without any wildcard, as it operates on the directory structure itself. If the folder contained subfolders, this method would fail the RmDir step, requiring manual intervention or the use of the **FileSystemObject** for recursive handling.

```
Sub DeleteFolder()

On Error Resume Next

'delete all files in folder
Kill "C:UsersbobbiDesktopMy_Data*.*"

'delete empty folder
RmDir "C:UsersbobbiDesktopMy_Data"

On Error GoTo 0

End Sub
```

Running this macro ensures the complete obliteration of the **My_Data** directory. The primary advantage of using **On Error Resume Next** here is that if the **Kill statement** fails (perhaps because the folder was already empty), the code still proceeds to the RmDir statement, ensuring the folder is removed if possible. The following visual confirmation illustrates the disappearance of the directory from the file system, demonstrating the macro's successful and clean execution.



Advanced Consideration: Recursive Deletion using FileSystemObject

While the native **Kill** and RmDir statements are excellent for basic folder deletion, they present limitations, particularly when dealing with directories containing subfolders, as they cannot handle recursive deletion natively. For scenarios requiring the removal of complex, non-empty directory trees that include multiple nested folders, the **FileSystemObject (FSO)** remains the superior tool, provided the developer sets the necessary reference to the **Microsoft Scripting Runtime** library. The FSO provides robust methods that simplify complex file system interactions significantly.

Specifically, the **DeleteFolder method** of the FileSystemObject is capable of performing a recursive deletion in a single line of code. When this method is executed, it automatically handles the removal of all subfolders and files contained within the specified path, eliminating the need for sequential **Kill** and **RmDir** calls and custom looping logic. This offers both efficiency and code elegance for large-scale, complex cleanup operations, making it the preferred method for professional environments.

To implement the FSO method, a developer would first instantiate the object (e.g., `Set fso = CreateObject("Scripting.FileSystemObject")`) and then call the **DeleteFolder** method, passing the folder path and a boolean argument (e.g., `fso.DeleteFolder(Path, True)`) to indicate forced or recursive deletion. Although this article prioritized native VBA statements for

foundational understanding, incorporating the FSO is highly recommended for professional VBA development involving extensive file system manipulation, especially when folder emptiness cannot be guaranteed prior to deletion.