Dynamic link library (DLL) - Windows Client

By kaushika-msft

Archived: 2026-04-05 13:03:46 UTC

This article describes what a dynamic link library (DLL) is and the various issues that may occur when you use DLLs. It also describes some advanced issues that you should consider when developing your own DLLs.

Applies to: Windows 10 - all editions
Original KB number: 815065

Summary

In describing what a DLL is, this article describes dynamic linking methods, DLL dependencies, DLL entry points, exporting DLL functions, and DLL troubleshooting tools.

This article finishes with a high-level comparison of DLLs to the Microsoft .NET Framework assemblies.

For the Windows operating systems, much of the functionality of the operating system is provided by DLL. Additionally, when you run a program on one of these Windows operating systems, much of the functionality of the program may be provided by DLLs. For example, some programs may contain many different modules, and each module of the program is contained and distributed in DLLs.

The use of DLLs helps promote modularization of code, code reuse, efficient memory usage, and reduced disk space. So, the operating system and the programs load faster, run faster, and take less disk space on the computer.

When a program uses a DLL, an issue that is called dependency may cause the program not to run. When a program uses a DLL, a dependency is created. If another program overwrites and breaks this dependency, the original program may not successfully run.

With the introduction of the .NET Framework, most dependency problems have been eliminated by using assemblies.

More information

A DLL is a library that contains code and data that can be used by more than one program at the same time. For example, in Windows operating systems, the Comdlg32 DLL performs common dialog box related functions. Each program can use the functionality that is contained in this DLL to implement an **Open** dialog box. It helps promote code reuse and efficient memory usage.

By using a DLL, a program can be modularized into separate components. For example, an accounting program may be sold by module. Each module can be loaded into the main program at run time if that module is installed. Because the modules are separate, the load time of the program is faster. And a module is only loaded when that functionality is requested.

Additionally, updates are easier to apply to each module without affecting other parts of the program. For example, you may have a payroll program, and the tax rates change each year. When these changes are isolated to a DLL, you can apply an update without needing to build or install the whole program again.

The following list describes some of the files that are implemented as DLLs in Windows operating systems:

- ActiveX Controls (.ocx) files

  An example of an ActiveX control is a calendar control that lets you select a date from a calendar.

- Control Panel (.cpl) files

  An example of a .cpl file is an item that is located in Control Panel. Each item is a specialized DLL.

- Device driver (.drv) files

  An example of a device driver is a printer driver that controls the printing to a printer.

DLL advantages

The following list describes some of the advantages that are provided when a program uses a DLL:

- Uses fewer resources

  When multiple programs use the same library of functions, a DLL can reduce the duplication of code that is loaded on the disk and in physical memory. It can greatly influence the performance of not just the program that is running in the foreground, but also other programs that are running on the Windows operating system.

- Promotes modular architecture

  A DLL helps promote developing modular programs. It helps you develop large programs that require multiple language versions or a program that requires modular architecture. An example of a modular program is an accounting program that has many modules that can be dynamically loaded at run time.

- Eases deployment and installation

  When a function within a DLL needs an update or a fix, the deployment and installation of the DLL does not require the program to be relinked with the DLL. Additionally, if multiple programs use the same DLL, the multiple programs will all benefit from the update or the fix. This issue may more frequently occur when you use a third-party DLL that is regularly updated or fixed.

DLL dependencies

When a program or a DLL uses a DLL function in another DLL, a dependency is created. The program is no longer self-contained, and the program may experience problems if the dependency is broken. For example, the program may not run if one of the following actions occurs:

- A dependent DLL is upgraded to a new version.
- A dependent DLL is fixed.
- A dependent DLL is overwritten with an earlier version.
- A dependent DLL is removed from the computer.

These actions are known as DLL conflicts. If backward compatibility is not enforced, the program may not successfully run.

The following list describes the changes that have been introduced in Windows 2000 and in later Windows operating systems to help minimize dependency issues:

- Windows File Protection

  In Windows File Protection, the operating system prevents system DLLs from being updated or deleted by an unauthorized agent. When a program installation tries to remove or update a DLL that is defined as a system DLL, Windows File Protection will look for a valid digital signature.

- Private DLLs

  Private DLLs let you isolate a program from changes that are made to shared DLLs. Private DLLs use version-specific information or an empty `.local` file to enforce the version of the DLL that is used by the program. To use private DLLs, locate your DLLs in the program root folder. Then, for new programs, add version-specific information to the DLL. For old programs, use an empty `.local` file. Each method tells the operating system to use the private DLLs that are located in the program root folder.

Several tools are available to help you troubleshoot DLL problems. The following tools are some of these tools.

Dependency Walker

The Dependency Walker tool can recursively scan for all dependent DLLs that are used by a program. When you open a program in Dependency Walker, Dependency Walker does the following checks:

- Dependency Walker checks for missing DLLs.
- Dependency Walker checks for program files or DLLs that are not valid.
- Dependency Walker checks that import functions and export functions match.
- Dependency Walker checks for circular dependency errors.
- Dependency Walker checks for modules that are not valid because the modules are for a different operating system.

By using Dependency Walker, you can document all the DLLs that a program uses. It may help prevent and correct DLL problems that may occur in the future. Dependency Walker is located in the following directory when you install Visual Studio 6.0:

```
drive\Program Files\Microsoft Visual Studio\Common\Tools
```

DLL Universal Problem Solver

The DLL Universal Problem Solver (DUPS) tool is used to audit, compare, document, and display DLL information. The following list describes the utilities that make up the DUPS tool:

- Dlister.exe

  This utility enumerates all the DLLs on the computer and logs the information to a text file or to a database file.

- Dcomp.exe

  This utility compares the DLLs that are listed in two text files and produces a third text file that contains the differences.

- Dtxt2DB.exe

  This utility loads the text files that are created by using the Dlister.exe utility and the Dcomp.exe utility into the dllHell database.

- DlgDtxt2DB.exe

  This utility provides a graphical user interface (GUI) version of the Dtxt2DB.exe utility.

DLL Help database

The DLL Help database helps you locate specific versions of DLLs that are installed by Microsoft software products.

DLL development

This section describes the issues and the requirements that you should consider when you develop your own DLLs.

Types of DLLs

When you load a DLL in an application, two methods of linking let you call the exported DLL functions. The two methods of linking are load-time dynamic linking and run-time dynamic linking.

Load-time dynamic linking

In load-time dynamic linking, an application makes explicit calls to exported DLL functions like local functions. To use load-time dynamic linking, provide a header (.h) file and an import library (.lib) file when you compile and link the application. When you do this, the linker will provide the system with the information that is required to load the DLL and resolve the exported DLL function locations at load time.

Run-time dynamic linking

In run-time dynamic linking, an application calls either the `LoadLibrary` function or the `LoadLibraryEx` function to load the DLL at run time. After the DLL is successfully loaded, you use the `GetProcAddress` function to obtain the address of the exported DLL function that you want to call. When you use run-time dynamic linking, you do not need an import library file.

The following list describes the application criteria for when to use load-time dynamic linking and when to use run-time dynamic linking:

- Startup performance

  If the initial startup performance of the application is important, you should use run-time dynamic linking.

- Ease of use

  In load-time dynamic linking, the exported DLL functions are like local functions. This makes it easy for you to call these functions.

- Application logic

  In run-time dynamic linking, an application can branch to load different modules as required. It is important when you develop multiple-language versions.

The DLL entry point

When you create a DLL, you can optionally specify an entry point function. The entry point function is called when processes or threads attach themselves to the DLL or detached themselves from the DLL. You can use the entry point function to initialize data structures or to destroy data structures as required by the DLL. Additionally, if the application is multithreaded, you can use thread local storage (TLS) to allocate memory that is private to each thread in the entry point function. The following code is an example of the DLL entry point function.

```
 BOOL APIENTRY DllMain(
HANDLE hModule,// Handle to DLL module
DWORD ul_reason_for_call,// Reason for calling function
LPVOID lpReserved ) // Reserved
{
    switch ( ul_reason_for_call )
    {
        case DLL_PROCESS_ATTACHED: // A process is loading the DLL.
        break;
        case DLL_THREAD_ATTACHED: // A process is creating a new thread.
        break;
        case DLL_THREAD_DETACH: // A thread exits normally.
        break;
        case DLL_PROCESS_DETACH: // A process unloads the DLL.
        break;
    }
```

```
    return TRUE;
}
```

When the entry point function returns a FALSE value, the application will not start if you are using load-time dynamic linking. If you are using run-time dynamic linking, only the individual DLL will not load.

The entry point function should only perform simple initialization tasks and should not call any other DLL loading or termination functions. For example, in the entry point function, you should not directly or indirectly call the `LoadLibrary` function or the `LoadLibraryEx` function. Additionally, you should not call the `FreeLibrary` function when the process is terminating.

Note

In multithreaded applications, make sure that access to the DLL global data is synchronized (thread safe) to avoid possible data corruption. To do this, use TLS to provide unique data for each thread.

Export DLL functions

To export DLL functions, you can either add a function keyword to the exported DLL functions or create a module definition (.def) file that lists the exported DLL functions.

To use a function keyword, you must declare each function that you want to export with the following keyword: `__declspec(dllexport)`

To use exported DLL functions in the application, you must declare each function that you want to import with the following keyword: `__declspec(dllimport)`

Typically, you would use one header file that has a define statement and an `ifdef` statement to separate the export statement and the `import` statement.

You can also use a module definition file to declare exported DLL functions. When you use a module definition file, you do not have to add the function keyword to the exported DLL functions. In the module definition file, you declare the `LIBRARY` statement and the `EXPORTS` statement for the DLL. The following code is an example of a definition file.

```
 // SampleDLL.def
//
LIBRARY "sampleDLL"
EXPORTS HelloWorld
```

Sample DLL and application

In Visual C++ 6.0, you can create a DLL by selecting either the **Win32 Dynamic-Link Library** project type or the **MFC AppWizard (dll)** project type.

The following code is an example of a DLL that was created in Visual C++ by using the **Win32 Dynamic-Link Library** project type.

```
 // SampleDLL.cpp
//

#include "stdafx.h"
#define EXPORTING_DLL
#include "sampleDLL.h"
BOOL APIENTRY DllMain( HANDLE hModule, DWORD ul_reason_for_call, LPVOID lpReserved
)
{
    return TRUE;
}

void HelloWorld()
{
    MessageBox( NULL, TEXT("Hello World"), TEXT("In a DLL"), MB_OK);
}

// File: SampleDLL.h
//
#ifndef INDLL_H
    #define INDLL_H
    #ifdef EXPORTING_DLL
        extern __declspec(dllexport) void HelloWorld();
    #else
        extern __declspec(dllimport) void HelloWorld();
    #endif

#endif
```

The following code is an example of a **Win32 Application** project that calls the exported DLL function in the SampleDLL DLL.

```
 // SampleApp.cpp
//
#include "stdafx.h"
#include "sampleDLL.h"
int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    HelloWorld();
    return 0;
}
```

Note

In load-time dynamic linking, you must link the SampleDLL.lib import library that is created when you build the SampleDLL project.

In run-time dynamic linking, you use code that is similar to the following code to call the SampleDLL.dll exported DLL function.

```
 ...
typedef VOID (*DLLPROC) (LPTSTR);
...
HINSTANCE hinstDLL;
DLLPROC HelloWorld;
BOOL fFreeDLL;

hinstDLL = LoadLibrary("sampleDLL.dll");
if (hinstDLL != NULL)
{
    HelloWorld = (DLLPROC) GetProcAddress(hinstDLL, "HelloWorld");
    if (HelloWorld != NULL)
        (HelloWorld);
    fFreeDLL = FreeLibrary(hinstDLL);
}
...
```

When you compile and link the SampleDLL application, the Windows operating system searches for the SampleDLL DLL in the following locations in this order:

1. The application folder
2. The current folder
3. The Windows system folder

   Note

   The GetSystemDirectory function returns the path of the Windows system folder.

4. The Windows folder

   Note

   The GetWindowsDirectory function returns the path of the Windows folder.

The .NET Framework assembly

With the introduction of .NET and the .NET Framework, most of the problems that are associated with DLLs have been eliminated by using assemblies. An assembly is a logical unit of functionality that runs under the control of

the .NET common language runtime (CLR). An assembly physically exists as a .dll file or as an .exe file. However, internally an assembly is different from a Microsoft Win32 DLL.

An assembly file contains an assembly manifest, type metadata, Microsoft intermediate language (MSIL) code, and other resources. The assembly manifest contains the assembly metadata that provides all the information that is required for an assembly to be self-describing. The following information is included in the assembly manifest:

- Assembly name
- Version information
- Culture information
- Strong name information
- The assembly list of files
- Type reference information
- Referenced and dependent assembly information

The MSIL code that is contained in the assembly cannot be directly executed. Instead, MSIL code execution is managed through the CLR. By default, when you create an assembly, the assembly is private to the application. To create a shared assembly requires that you assign a strong name to the assembly and then publish the assembly in the global assembly cache.

The following list describes some of the features of assemblies compared to the features of Win32 DLLs:

- Self-describing

  When you create an assembly, all the information that is required for the CLR to run the assembly is contained in the assembly manifest. The assembly manifest contains a list of the dependent assemblies. Therefore, the CLR can maintain a consistent set of assemblies that are used in the application. In Win32 DLLs, you cannot maintain consistency between a set of DLLs that are used in an application when you use shared DLLs.

- Versioning

  In an assembly manifest, version information is recorded and enforced by the CLR. Additionally, version policies let you enforce version-specific usage. In Win32 DLLs, versioning can't be enforced by the operating system. You must make sure that DLLs are backward compatible.

- Side-by-side deployment

  Assemblies support side-by-side deployment. One application can use one version of an assembly, and another application can use a different version of an assembly. Starting in Windows 2000, side-by-side deployment is supported by locating DLLs in the application folder. Additionally, Windows File Protection prevents system DLLs from being overwritten or replaced by an unauthorized agent.

- Self-containment and isolation

  An application that is developed by using an assembly can be self-contained and isolated from other applications that are running on the computer. This feature helps you create zero-impact installations.

- Execution

  An assembly is run under the security permissions that are supplied in the assembly manifest and that are controlled by the CLR.

- Language independent

  An assembly can be developed by using any one of the supported .NET languages. For example, you can develop an assembly in Microsoft Visual C#, and then use the assembly in a Visual Basic .NET project.

Data collection

If you need assistance from Microsoft support, we recommend you collect the information by following the steps mentioned in Gather information by using TSS for deployment-related issues.

References

- Deploying and Configuring Applications
- Assemblies
- Run-Time Dynamic Linking
- Thread Local Storage
