

Dependency Walker Frequently Asked Questions (FAQ)



Q.	Dependency Walker seems to only show some of my application's dependencies. Why doesn't it show all of them?
A.	When you first open a module in Dependency Walker, it only shows implicit, forwarded, and delay-load dependencies. Many dependencies are loaded dynamically and will not be detected until you profile the application from within Dependency Walker. For more information, see Types of Dependencies Handled By Dependency Walker and Using Application Profiling to Detect Dynamic Dependencies .
Q.	Why am I seeing a lot of applications where MPR.DLL shows up in red under SHLWAPI.DLL because it is missing a function named WNetRestoreConnectionA? I also get a "Warning: At least one module has an unresolved import due to a missing export function in a delay-load dependent module" message.
A.	Some versions of SHLWAPI.DLL (like the one on Windows XP) have a delay-load dependency on the function WNetRestoreConnectionA in MPR.DLL. Missing delay-load functions are not a problem as long as the calling DLL is prepared to handle the situation. Dependency Walker flags all potential problems as it cannot detect if an application intends to handle the issue. In the case of SHLWAPI.DLL, this is not an problem as it does not require WNetRestoreConnectionA to exist and handles the missing function at runtime. This warning can be ignored. See the "How to Interpret Warnings and Errors in Dependency Walker" section in help for more details.
Q.	Why is MSJAVA.DLL showing up in yellow (missing module) and I get a "Warning: At least one delay-load dependency module was not found" message?
A.	The MSHTML.DLL module that was released with Windows XP SP2 and Windows 2003 SP1 has a delay-load dependency on MSJAVA.DLL. Missing delay-load dependencies are not a problem as long as the calling DLL is prepared to handle the missing module. Dependency Walker flags all potential problems as it cannot detect if an application intends to handle the issue. In this particular case, MSJAVA.DLL is an optional module, and MSHTML.DLL is prepared to handle it. This warning can be ignored. See the "How to Interpret Warnings and Errors in Dependency Walker" section in help for more details.
Q.	Dependency Walker says I'm missing APPHELP.DLL. Where can I get it from?
A.	APPHELP.DLL is used by Windows XP's application compatibility feature. It is a Windows XP/2003/Vista/+ only DLL. If you see this warning, you most likely installed Internet Explorer 6.0 on your pre- Windows XP computer (Windows 95/98/ME/2000). Internet Explorer 6.0 installs a new SHWAPI.DLL that has a delay-load dependency on APPHELP.DLL. This is normal as SHWAPI.DLL does not expect to find APPHELP.DLL on versions of Windows prior to Windows XP. This warning can be ignored. You do not need (or want) APPHELP.DLL on Windows 95/98/ME/2000.
Q.	Can Dependency Walker help me figure out why my component won't register? [or] Why does REGSVR32.EXE fail to register my DLL, but Dependency Walker does not show any error with my DLL?
A.	Many modules need to be "registered" on a computer before they will work. This includes most ActiveX controls, OCXs, COM components, ATL components, Visual Basic



components, and many others. These types of modules are usually registered with REGSVR32.EXE or something similar. For the most part, REGSVR32.EXE loads your DLL, calls GetProcAddress for the DLL's DllRegisterServer function, then calls that function. A common failure is when your DLL relies on another DLL that is missing or not registered. If you just open your DLL in Dependency Walker, you may or may not see a problem, depending on the type of registration failure.

The best way to debug a module that fails to register is by opening REGSVR32.EXE in Dependency Walker rather than your DLL. Then choose to start profiling (F7). In the profiling dialog, enter the full path to your DLL in the "Program arguments" field. For "Starting directory", you may wish to enter the directory that the DLL resides in. Check the options you wish to use and press Ok. This will run REGSVR32.EXE and attempt to register your DLL. By actually running REGSVR32.EXE, you can see more types of runtime errors.

Q.

My application runs better when being profiled by Dependency Walker than when I run it by itself. Why is this?

A.

I've had several reports of applications that normally crash, will not crash when being profiled under Dependency Walker. Dependency Walker acts as a debugger when you are profiling your application. This in itself, makes your program run differently.

First, there is the overhead of Dependency Walker that slows the execution of your application down. If your application is crashing due to some race condition, this slow down alone might be enough to avoid the race condition. If this is the case, it is a design issue of the application and you are just getting lucky when it doesn't crash.

Second, normally when threads block on critical sections, events, semaphores, mutexes, etc., they unblock on a first-in-first-out (FIFO) basis. This is not guaranteed by the OS, but is usually the case. When being run under a debugger, FIFO queues are sometimes randomized, so threads may block and resume in a different order than they would when not running under a debugger. This might be relieving a race condition or altering the execution enough to make things work. Again, the application is just getting lucky when it doesn't crash.

Finally, applications running under the debugger automatically get a system debug heap. All memory functions are handled slightly different. Allocations are padded with guard bytes to check to see if you are writing outside of a region you have allocated (buffer overrun/underrun). Allocations might also be laid out differently in memory than when not under the debugger. So, if you are writing past the end of a buffer under the debugger, you might be trashing guard bytes, freed memory, or just something not very critical. However, when not running under the debugger, you might be trashing something critical (like a pointer), and your app crashes.

For the debug heap, you can turn this off in Dependency Walker and see if your application crashes when being profiled. If it does then, then you probably suffer a buffer overrun, stray/bad/freed pointer, etc. To do this, start a command prompt. Type "SET _NO_DEBUG_HEAP=1". Then start Dependency Walker from that command line. This should disable the debug heap for that instance of Dependency Walker. Note, this only works on Windows XP and beyond.



<p>Q.</p>	<p>How do I view the parameter and return types of a function?</p>
<p>A.</p>	<p>For most functions, this information is simply not present in the module. The Windows' module file format only provides a single text string to identify each function. There is no structured way to list the number of parameters, the parameter types, or the return type. However, some languages do something called function "decoration" or "mangling", which is the process of encoding information into the text string. For example, a function like int Foo(int, int) encoded with simple decoration might be exported as _Foo@8. The 8 refers to the number of bytes used by the parameters. If C++ decoration is used, the function would be exported as ?Foo@@YGHHH@Z, which can be directly decoded back to the function's original prototype: int Foo(int, int). Dependency Walker supports C++ undecoration by using the Undecorate C++ Functions Command.</p>
<p>Q.</p>	<p>Why are my function names exported differently then I declare them?</p>
<p>A.</p>	<p>Many compilers "decorate" function names by default. Unless you give the compiler specific instructions on how to export functions, a function like int Foo(int, int) may end up getting exported as _Foo@8, or even ?Foo@@YGHHH@Z if C++ decoration is used. Languages like C++ allow function overloading, which is the ability to declare multiple functions with the same name, but with different parameters. Because of this, each function must have a unique signature string since exporting just the name would cause a name conflict. To disable C++ decoration, you can use the extern "C" notation when declaring your functions in a C++ source file. To prevent decoration altogether, you can add a DEF file to your C/C++ project and declare the actual function names you want exported.</p>
<p>Q.</p>	<p>My application seems to run just fine during profiling, however, I see errors in the log view and red or yellow icons in the other views. Is this normal?</p>
<p>A.</p>	<p>It is fairly normal to see errors or warnings during profiling. One common error seen is when one module tries to dynamically load another module (using one of the LoadLibrary functions), but the module is not found. Dependency Walker makes a note of this failure, but if the application is prepared for the failure, then this is not a problem. Another common error is when a module tries to dynamically locate a function (using GetProcAddress) in a module. Again, this is not a problem if the application is prepared for the failure. You may also see first-chance exceptions occur in the log view. If the application handles the exceptions and they don't turn into second-chance exceptions, then this is not a problem. All these cases are normal, and can usually be ignored. However, if the application you are profiling crashes or fails to run properly, then the errors may provide some insight as to what caused the problem. See the How to Interpret Warnings and Errors in Dependency Walker section for more details.</p>
<p>Q.</p>	<p>Wow, my application depends on <u>all</u> those files? Which ones do I need to redistribute with my application?</p>
<p>A.</p>	<p>For starters, there are certain modules you should never redistribute with your application, such as kernel32.dll, user32.dll, and gdi32.dll. To see which files you are allowed to redistribute, you can look for a file named REDIST.TXT on your development computer. This file is included with development suites like Microsoft Visual C++ and Visual Basic. You can also look up "redistributable files" and "redist.txt" in the MSDN index for more information on what files to redistribute, how to redistribute them, how to check file versions, etc. Another site</p>



	worth mentioning is the Microsoft DLL Help Database (http://support.microsoft.com/dllhelp). This site has detailed version histories of DLLs, and lists what products were shipped with each version.
Q.	What does "Shared module not hooked" mean, and why are some module's DllMain calls never being logged?
A.	Dependency Walker hooks modules as they load in order to track calls to functions like DllMain, LoadLibrary, and GetProcAddress. Any module loaded above address 0x80000000 (usually system modules) on Windows 95/98/Me is shared system-wide and cannot be hooked. The result is that Dependency Walker cannot log information about function calls in those modules. Windows NT/2000/XP/2003/Vista/+ does not have this limitation. See Using Application Profiling to Detect Dynamic Dependencies for more information.
Q.	Why do some modules show up more than once under a single parent module?
A.	Dependency Walker may show a module more than once to inform you that it is a dependency for more than one reason. It is possible for a module to show up as an implicitly linked dependency, a forwarded dependency, and a dynamic dependency, all under a single parent module. See the Module Dependency Tree View for more details. In reality, only one copy of the module resides in memory during run-time.
Q.	Is there a command line version of Dependency Walker?
A.	Dependency Walker can be run as a graphical application or as a console application. When the console mode option is used, Dependency Walker can process a module, save the results, and exit without any graphical interface or user prompting. See the Command Line Options section for more information.
Q.	Will Dependency Walker work with COM, Visual Basic, or .NET modules?
A.	Yes. Dependency Walker will work with any 32-bit or 64-bit Windows module, regardless of what language was used to develop it. However, many languages have their own way to specify dependency relationships between modules. For example, COM modules may have embedded type libraries and registration information in the registry, and .NET modules may use .NET assemblies. These techniques are all implemented as layers above the core Windows API. In the end, these layers still need to call down to the core Windows functions like LoadLibrary and GetProcAddress to do the actual work. It is at this core level that Dependency Walker understands what is going on. So, while Dependency Walker may not understand all the language specific complexities of your application, it will still be able to track all module activity at a core Windows API level.
Q.	Will Dependency Walker work with 64-bit modules?
A.	Yes. Dependency Walker will work with any 32-bit or 64-bit Windows module. There are 32-bit and 64-bit versions Dependency Walker. All versions are capable of opening 32-bit and 64-bit modules. However, there are major advantages to using the 32-bit Dependency

Dependency Walker Frequently Asked Questions (FAQ)



	<p>Walker to process 32-bit modules and the 64-bit Dependency Walker to process 64-bit modules. This is especially true when running on a 64-bit version of Windows, which allows execution of both 32-bit and 64-bit programs. The 32-bit subsystem on 64-bit Windows (known as "WOW64") has its own private registry, "AppPaths", "KnownDlls", system folders, and manifest processing. Only the 32-bit version of Dependency Walker can access this 32-bit environment, which is needed to accurately process a 32-bit module. Likewise, only the 64-bit version of Dependency Walker can fully access the 64-bit environment, so it should always be used for processing 64-bit modules.</p>
Q.	Why is the "Start Profiling" button and menu item disabled?
A.	<p>The profiling option works by actually executing your application and watching it to see what it loads. In order for this to be possible, you need to have opened an executable (usually has an EXE extension) rather than a DLL. If you want to profile a DLL, you will need to open some executable that loads the DLL (see the FAQ about using REGSVR32.EXE to load DLLs). The profiling feature also requires that the executable you have loaded is for the same CPU architecture as the version of Dependency Walker you are currently running. For example, you need the 32-bit x86 version of Dependency Walker to profile a 32-bit x86 executable, and the 64-bit x64 version of Dependency Walker to profile a 64-bit x64 executable.</p>
Q.	Will Dependency Walker work with Windows CE modules?
A.	<p>Yes. Windows CE modules use the same module format (known as the "Portable Executable" format) that is used for modules written for Windows 95, Windows 98, Windows Me, Windows NT, Windows 2000, Windows XP, Windows 2003, Windows Vista, and beyond. There is no version of Dependency Walker that actually runs on Windows CE, but you can open Windows CE modules with Dependency Walker on a standard Windows computer. However, Dependency Walker automatically tries to locate dependent modules using the default Windows module search path. For Windows CE modules, this can cause errors since non-CE modules may be found in the default search path. To fix this, you can use Dependency Walker's "Configure Module Search Order" dialog to remove all standard paths and then add a private folder of your own that contains only CE modules. If you frequently find yourself doing this, you can save your custom search order to a file and then later pass the file to Dependency Walker using the "/d:your_file.dwp" command line option (see Command Line Options for more details).</p>
Q.	Will Dependency Walker work with 16-bit modules?
A.	<p>No. Dependency Walker only supports 32-bit and 64-bit Windows modules. It never has and never will support 16-bit.</p>
Q.	What do all the version numbers mean?
A.	<p>See the Overview of Module Version Numbers section for the details.</p>

Dependency Walker Frequently Asked Questions (FAQ)



Q.	Can I print out the results of a session?
A.	No, but you can save the results to several different text formats which can be viewed or printed from a text viewer program like Notepad.
Q.	How can I send the results of a session to someone?
A.	Dependency Walker supports several ways to capture the data in a session. All the views support simple copying from them using the Copy Command. Dependency Walker also supports several methods of saving the entire session to a file. There are various text formats that can be easily printed or emailed to someone for viewing. You can also save the results to a Dependency Walker Image (DWI) file, which can be loaded by Dependency Walker on another computer to see the captured results from your computer. For more information on saving the session to a file, see the Save Command and File Save Dialog section.
Q.	What do all the icons mean?
A.	Each view in Dependency Walker has detailed help describing what the icons mean for that view. See the Module Session Window section for a list of views.
Q.	Can I search for a function by name or ordinal?
A.	All the list views in Dependency Walker can be sorted and searched. Any text you type while in a list view will search for that text in the column that the list is currently sorted by. For example, if the export function list is sorted by function names and you type "Get", the first function that starts with "Get" will be highlighted. This will work for any column in any list. For more details, see the help sections for the actual list views.
Q.	Dependency Walker's open dialog is not showing a file that I want to open. How can I fix this?
A.	By default, Windows "hides" certain system files (like DLLs) from the user. To change this setting, open "My Computer" and select "Options" from the menu. Depending on what version of Windows you are using, this should be off of the "View" or "Tools" menu, and may be called "Folder Options" or just "Options". In the dialog that appears, choose the "View" tab. You should see an option that reads either "Show all files" or "Show hidden files and folders". Make sure this option is selected. You will also see a check-box that reads "Hide MS-DOS file extensions for file types that are registered" or "Hide file extensions for known file types". You will want to uncheck this box. Once done, press "Ok" in that dialog. Dependency Walker should now show all system files in its open dialog.
Q.	How do I uninstall Dependency Walker?
A.	Dependency Walker does not have a setup or uninstall program. It was designed to simply run when you want it, and delete if you don't need it anymore. If you have told Dependency



	<p>Walker to handle certain file extensions, you will probably want to remove those associations before deleting the program. This can be done by using the Handled File Extensions command. The files to delete when Dependency Walker is no longer needed are depends.exe, depends.dll, and depends.chm.</p>
Q.	Why are some modules looking for a function named "IsTNT" in KERNEL32.DLL?
A.	TNT is a 32-bit emulation layer written by Phar Lap. There are still some modules in use that have pieces of code that check to see if they are running on TNT by calling GetProcAddress("IsTNT") for KERNEL32.DLL. This warning can be ignored.
Q.	Why are some modules trying to load a module named "AUX"?
A.	This is usually related to modules trying to load the AUX audio driver. Since AUX is a reserved DOS name, the load fails. This warning is harmless and can be ignored.
Q.	MFC42.DLL is trying to load MFC42LOC.DLL, but it is not found. [or] COMCTL32.DLL is trying to load CMCTLENU.DLL, but it is not found. Why is this?
A.	Both MFC42LOC.DLL and CMCTLENU.DLL are language specific resource DLLs that may not be needed on your system. Many modules on Windows store all their language specific messages in external DLLs (one per language). At run-time, the module loads the language DLL for the current language of the operating system. The names of the modules usually end in "ENU" for United States English, "ESP" for Spanish, "JPN" for Japanese, etc. The "LOC" ending that MFC uses stands for "localized". When MFC is installed, it copies the correct language DLL to your system and renames it to MFC42LOC.DLL. So, why the missing module? Well, most modules protect themselves from failure by storing one default language in the main DLL itself. If the language specific resource DLL fails to load, then the module defaults to using the local resources in itself. In most cases, these default resources are the same resources as would be in the ENU version of the resource DLL. For this reason, there does not need to be an ENU version of the resource DLL, and therefore it fails to find one at runtime. This is normal.