Why Use Dependency Walker?

Have you ever...

...wondered why an application or module was failing to load?

...wondered what minimum set of files are required to run a particular application or load a particular DLL?

...wondered why a certain module was being loaded with a particular application?

...wanted to know what functions are exposed by a particular module, and which ones are actually being called by other modules?

...wanted to know the parameter and return types of exported C++ functions?

...wanted to remove all dependencies for a given module?

...wanted to know the complete path of all the modules being loaded for a particular application?

...wanted to know all the base addresses of each module being loaded for a particular application? What about versions? Or maybe CPU types?

...received one of the following errors...

> The dynamic link library BAR.DLL could not be found in the specified path...
>
> The procedure entry point FOO could not be located in the dynamic link library BAR.DLL.
>
> The application or DLL BAR.DLL is not a valid Windows image.
>
> The application failed to initialize properly.
>
> Initialization of the dynamic link library BAR.DLL failed. The process is terminating abnormally.
>
> The image file BAR.EXE is valid, but is for a machine type other than the current machine.

Program too big to fit in memory.

Using Dependency Walker for Troubleshooting Modules

Dependency Walker recursively scans all dependent modules required by a particular application. During this scan it performs the following tasks:

Detects missing files. These are files that are required as a dependency to another module. A symptom of this problem is the "The dynamic link library BAR.DLL could not be found in the specified path..." error.

Detects invalid Files. This includes files that are not Win32 or Win64 compliant and files that are corrupt. A symptom of this problem is the "The application or DLL BAR.EXE is not a valid Windows image" error.

Detects import/export mismatches. Verifies that all functions imported by a module are actually exported from the dependent modules. All unresolved import functions are flagged with an error. A symptom of this problem is the "The procedure entry point FOO could not be located in the dynamic link library BAR.DLL" error.

Detects circular dependency errors. This is a very rare error, but can occur with forwarded functions.

Detects mismatched CPU types of modules. This occurs if a module built for one CPU tries to load a module built for a different CPU.

Detects checksum inconsistencies by verifying module checksums to see if any modules have been modified after they were built.

Detects module collisions by highlighting any modules that fail to load at their preferred base address.

Detects module initialization failures by tracking calls to module entrypoints and looking for errors.

Dependency Walker can also perform a run-time profile of your application to detect dynamically loaded modules and module initialization failures. The same error checking from above applies to dynamically loaded modules as well.

Using Dependency Walker for General Information about Modules

Dependency Walker is more than just a troubleshooting utility. It also provides a great deal of valuable information about the module layout of a particular application and details on each module. Dependency Walker provides the following information:

A complete module dependency tree diagram of all the modules required by a particular application.

A list of all functions exported from each module. These lists include functions exported by name, functions exported by ordinal, and functions that are actually forwarded to other modules. Named C++ functions can be shown in their native decorated format, or can be expanded into human readable function prototypes including return types and parameters types.

A list of functions that are actually called in each module by other modules. These lists can help developers understand why a particular module is being linked with an application, and also provides information on how to remove unneeded modules from being dependencies.

A list of the minimum set of files that are required in order for a module to load and run. This list can be very useful when copying files to another computer or creating setup scripts.

For each individual module found, the following information is provided...

Full path to the module file.

Date and time of the module file.

Date and time the module was actually built.

Size of the module file.

Attributes of the module file.

The module checksum from when the module was built.

The actual module checksum.

Type of CPU that the module was built for.

Type of subsystem that the module was built to run in.

Type of debugging symbols that are associated with the module.

The preferred base load address of the module.

The actual base load address of the module.

The virtual size of the module.

The load order of the module with respect to other modules.

The file version found in the module's version resource.

The product version found in the module's version resource.

The image version found in the module's file header.

The version of the linker that was used to create the module file.

The version of the OS that the module file was built to run on.

The version of the subsystem that the module file was built to run in.

A possible error message if any error occurred while processing the file.

Command Line Options and Return Values

DEPENDS.EXE	[/?] [/c] [/a:#] [/f:#] [/u:#] [/ps:#] [/pp:#] [/po:#] [/ph:#] [/pl:#] [/pg:#] [/pt:#] [/pn:#] [/pe:#] [/pm:#] [/pf:#] [/pi:#] [/pc:#] [/pa:#] [/pd:dir] [/pb] [/sm:#] [/si:#] [/se:#] [/sf:#] [/od:path] [/ot:path] [/of:path] [/oc:path] [/d:path] [path [args...]]
/?	**Help** - Displays this page.
/c	**Console mode** - Dependency Walker will process the other command line options and exit without displaying its graphical interface. You must specify a module or Dependency Walker Image (DWI) file to open when using this option.
/a:#	**Auto Expand** - Use /a:0 to start Dependency Walker with the Auto Expand setting initially turned off, or /a:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used.
/f:#	**View full paths** - Use /f:0 to start Dependency Walker with the View Full Paths setting initially turned off, or /f:1 to start

	with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used.
/u:#	**Undecorate C++ functions** - Use /u:0 to start Dependency Walker with the Undecorate C++ Functions setting initially turned off, or /u:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used.
/ps:#	**Profiling option: Simulate ShellExecute by inserting any App Paths directories into the PATH environment variable** - Use /ps:0 to start Dependency Walker with this setting initially turned off, or /ps:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used.
/pp:#	**Profiling option: Log DllMain calls for process attach and process detach messages** - Use /pp:0 to start Dependency Walker with this setting initially turned off, or /pp:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used.
/po:#	**Profiling option: Log DllMain calls for all other messages, including thread attach and thread detach** - Use /po:0 to start Dependency Walker with this setting initially turned off, or /po:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used.
/ph:#	**Profiling option: Hook the process to gather more detailed dependency information** - Use /ph:0 to start Dependency Walker with this setting initially turned off, or /ph:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used.
/pl:#	**Profiling option: Log LoadLibrary function calls** - Use /pl:0 to start Dependency Walker with this setting initially turned off, or /pl:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used. If this option is turned on, then the "Hook the process to gather more detailed

	dependency information" option will also be turned on.
/pg:#	**<u>P</u>rofiling option: Log <u>G</u>etProcAddress function calls** - Use /pg:0 to start Dependency Walker with this setting initially turned off, or /pg:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used. If this option is turned on, then the "Hook the process to gather more detailed dependency information" option will also be turned on.
/pt:#	**<u>P</u>rofiling option: Log <u>t</u>hread information** - Use /pt:0 to start Dependency Walker with this setting initially turned off, or /pt:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used.
/pn:#	**<u>P</u>rofiling option: Use simple thread <u>n</u>umbers instead of actual thread IDs** - Use /pn:0 to start Dependency Walker with this setting initially turned off, or /pn:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used. If this option is turned on, then the "Log thread information" option will also be turned on.
/pe:#	**<u>P</u>rofiling option: Log first chance <u>e</u>xceptions** - Use /pe:0 to start Dependency Walker with this setting initially turned off, or /pe:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used.
/pm:#	**<u>P</u>rofiling option: Log debug output <u>m</u>essages** - Use /pm:0 to start Dependency Walker with this setting initially turned off, or /pm:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used.
/pf:#	**<u>P</u>rofiling option: Use <u>f</u>ull paths when logging file names** - Use /pf:0 to start Dependency Walker with this setting initially turned off, or /pf:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used.
/pi:#	**<u>P</u>rofiling option: Log a t<u>i</u>me stamp with each line of log** - Use /pi:0 to start Dependency Walker with this setting initially

	turned off, or /pi:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used.
/pc:#	**Profiling option: Automatically open and profile child processes** - Use /pc:0 to start Dependency Walker with this setting initially turned off, or /pc:1 to start with it turned on. If this option is not specified, then the setting from the last time you ran Dependency Walker will be used. This option is ignored when running in console mode.
/pa:#	**Profiling option: Turn all profiling options on or off** - Use /pa:0 to initially turn all profiling options off, or /pa:1 to initially turn them all on. This option can be used before other profiling options. For example, /pa:1 /pf:0 will turn on all options except for the "Use full paths when logging file names" option.
/pd:dir	**Profiling option: Starting directory** - Specifies the starting directory to use when profiling the module. This option requires that you specify a module to open.
/pb	**Profiling option: Automatically begin profiling after the module has been loaded** - This option requires that you specify a module to open. If an output option (/od, /ot, /of, or /oc) is specified, Dependency Walker will wait until the profiling fully completes before saving the results.
/sm:#	**Sort column for module list view** - This option controls the initial sort column that Dependency Walker will use when sorting the items in the Module List View. If this option is not specified, then the value from the last time you ran Dependency Walker will be used. The values allowed are: 1. Icon 2. Module Name or Path 3. File Time Stamp 4. Link Time Stamp 5. File Size 6. File Attributes 7. Link Checksum 8. Real Checksum 9. CPU Type 10. Subsystem Type 11. Symbol Types

	12. Preferred Base Address 13. Actual Base Address 14. Virtual Size 15. Load Order 16. File Version 17. Product Version 18. Image Version 19. Linker Version 20. OS Version 21. Subsystem Version
/si:#	**Sort column for parent import function list view** - This option controls the initial sort column that Dependency Walker will use when sorting the items in the Parent Import Function List View. If neither this option or the /sf option is specified, then the value from the last time you ran Dependency Walker will be used. The values allowed are: 1. Icon 2. Ordinal Value 3. Hint Value 4. Function Name 5. Entry Point Address
/se:#	**Sort column for export function list views** - This option controls the initial sort column that Dependency Walker will use when sorting the items in the Export Function List View. If neither this option or the /sf option is specified, then the value from the last time you ran Dependency Walker will be used. The values allowed are: 1. Icon 2. Ordinal Value 3. Hint Value 4. Function Name 5. Entry Point Address
/sf:#	**Sort column for both function list views** - This option controls the initial sort column that Dependency Walker will use when sorting the items in both the Parent Import Function List View and the Export Function List View. If no sort column option is specified for a particular column, then the value(s) from the last time you ran Dependency Walker will be used. The values allowed are:

	1. Icon 2. Ordinal Value 3. Hint Value 4. Function Name 5. Entry Point Address
/od:path	**Output file in Dependency Walker Image (DWI) format** - This option requires that you specify a module or Dependency Walker Image (DWI) file to open. Once the module has been processed, the results will be written to the specified file in the Dependency Walker Image (DWI) format.
/ot:path	**Output file in text format** - This option requires that you specify a module or Dependency Walker Image (DWI) file to open. Once the module has been processed, the results will be written to the specified file in text format.
/of:path	**Output file in text format with import / export function lists** - This option requires that you specify a module or Dependency Walker Image (DWI) file to open. Once the module has been processed, the results will be written to the specified file in text format, including the import and export function lists.
/oc:path	**Output file in Comma Separated Value (CSV) format** - This option requires that you specify a module or Dependency Walker Image (DWI) file to open. Once the module has been processed, the results will be written to the specified file in a Comma Separated Value (CSV) format.
/d:path	**Dependency Walker Path (DWP) file to load** - This options allows you to specify a Dependency Walker Path (DWP) File to load and use as the initial search path when searching for modules. DWP files can be created using the Configure Module Search Order command in Dependency Walker.
path	**Path to a module or Dependency Walker Image (DWI) file to load** - For this option, you can specify a file name, a relative path, or a full path to a file to load. The file must be a 32-bit or 64-bit Windows module or a Dependency Walker Image (DWI) file. This path must come after any options intended for Dependency Walker since all options that follow this path are assumed to be program arguments for use when profiling the module.

args...	**Program arguments** - Specifies the command line arguments to use when profiling the module specified by the **path** option. Dependency Walker considers any text following the **path** option as being program arguments. For this reason, any options intended for Dependency Walker must be specified before the **path** option. If the file specified by the **path** option is really a Dependency Walker Image (DWI) file, then the args are ignored.

General Rules about Command Line Options

Options are case insensitive. For example, "/c" and "/C" are equivalent.

Options may start with a slash or a dash. For example, "/c" and "-c" are equivalent.

The colons (:) shown in the options above are optional. They may be removed or replaced with spaces. For example, "/f:0", "/f 0", and "/f0" are equivalent.

All profiling options are cumulative from left to right. For example, /pa:1 /pm:0 will turn on all the profiling options, then turn off the "Log debug output messages" option, but /pm:0 /pa:1 will simply turn on all profiling options.

Program options intended for Dependency Walker must come before the module path. All options after the module path will be passed to the module as its command line when profiled.

If you wish to specify text that has spaces, that text should be placed in quotes. For example:

```
depends /pb /oc "c:\output files\foo bar.csv" "c:\input files\foo bar.exe" 1 2 3 "this is a test"
```

Multiple options can be grouped together. You may even append options to other options that require numerical values. The only options that cannot be appended to are options that require a path or text values (-pd, -od, -ot, -of, -oc, and -d). For example:

```
    depends -c -f:0 -u:1 -pa:1 -pf:0 -pe:0 -pb -sm:12 -sf:4 -d:search.dwp -oc:result.csv -od:result.dwi foo.exe
```

Could be shortened to:

```
depends -cf0u1pa1pf0pe0pbsm12sf4dsearch.dwp -ocresult.csv -
odresult.dwi foo.exe bar
```

All options can be specified with or without the "Console Mode" option (/c).

More than one output file type option can be specified.

Return Values

When Dependency Walker exits, it returns a set of bit flags that are OR'ed together. There are three groups of error flags - module warnings, module errors, and processing errors. The error flags have been arranged in a way that makes it easy to detect the severity of a problem.

If the return value is greater than or equal to 0x00010000, then there was a processing error with Dependency Walker and no work was done. Otherwise, if the return value is greater than or equal to 0x00000100, then the operating system will not be able to load the module due to some module or dependency error. Otherwise, if the return value is greater than or equal to 0x00000001, then the module has no load-time dependency problems and will most likely have no problems loading, but may have runtime problems.

Module Warnings - Application should load, but might fail during runtime.

0x00000001	At least one dynamic dependency module was not found.
0x00000002	At least one delay-load dependency module was not found.
0x00000004	At least one module could not dynamically locate a function in another module using the GetProcAddress function call.
0x00000008	At least one module has an unresolved import due to a missing export function in a delay-load dependent module.
0x00000010	At least one module was corrupted or unrecognizable to Dependency Walker, but still appeared to be a Windows module.
0x00000020	At least one module failed to load during profiling. This usually occurs when a module returns 0 from its DllMain function or generates an unhandled exception while processing the DLL_PROCESS_ATTACH message.

Module Errors - Application will fail to load by the operating system.

0x00000100	At least one file was not a 32-bit or 64-bit Windows module.
0x00000200	At least one required implicit or forwarded dependency was not found.
0x00000400	At least one module has an unresolved import due to a missing export function in a dependent module.
0x00000800	Modules with different CPU types were found.
0x00001000	A circular dependency was detected.
0x00002000	There was an error in a Side-by-Side configuration file.

Processing Errors - All or some modules could not be processed.

0x00010000	There was an error with at least one command line option.
0x00020000	The file you specified to load could not be found.
0x00040000	At least one file could not be opened for reading.
0x00080000	The format of the Dependency Walker Image (DWI) file was unrecognized.
0x00100000	There was an error while trying to profile the application.
0x00200000	There was an error writing the results to an output file.
0x00400000	Dependency Walker ran out of memory.
0x00800000	Dependency Walker encountered an internal program error.

Overview of Module Version Numbers

There are four version fields that every Windows module is guaranteed to have. They are all two-part version numbers (#.#). They include:

Image Version	This value is set by the developer of the module by using the VERSION statement in their DEF file or by using the /VERSION linker option. It usually represents the version of the module or product that the module is part of, but can contain any value since it is up to the developer to set it. If the developer does not specify a version, then this value will default to

	0.0. This value may be used as a last resort when comparing two modules to check which module is newer.
OS Version	This value represents which version of the operating system the module was designed to run on. Certain functions may behave differently depending on this value in order to remain compatible with applications built for a particular operating system version.
Subsystem Version	This value represents which subsystem version the module was designed to run on. Most modules use the default value, but developers can override the default by using the /SUBSYSTEM linker option if they wish to target a particular subsystem version other than the default. Certain subsystem functions may behave differently depending on this value in order to remain compatible with applications built for a particular subsystem version.
Linker Version	This value represents the version of the linker that was used to build the module. It can be used to determine if a specific linker feature was available at the time the module was built. For example, delay-load dependencies is a new feature introduced with version 6.0 of the linker, so if this value is less than 6.0, the module shouldn't have any delay-load dependencies.

In addition to the four standard version values, developers can add four more optional version values by including a VERSION_INFO resource as part of their resource file. This resource structure has two four-part numeric fields (#.#.#.#) and two text fields. They include:

File Version Value	This field is known as the "FILEVERSION" field in the VERSION_INFO resource structure. This numerical value usually represents the version of the module itself, but can contain any value since it is up to the developer to set it. This is the value that most programs use when comparing two modules to check which module is newer.
Product Version Value	This field is known as the "PRODUCTVERSION" field in the VERSION_INFO resource structure. This numerical value usually represents the version of the product that this module is part of, but can

	contain any value since it is up to the developer to set it. For example, "Acme Tools version 3.0" is a set of ten utilities, including "Acme Virus Checker version 1.5". The virus checker executable might have a file version of 1.5.0.0 and a product version of 3.0.0.0
File Version Text	This field is known as the "FileVersion" field in the VERSION_INFO resource structure. This text string usually represents the version of the module itself, but can contain any text string since it is up to the developer to set it.
Product Version Text	This field is known as the "ProductVersion" field in the VERSION_INFO resource structure. This text string usually represents the version of the product that this module is part of, but can contain any text string since it is up to the developer to set it.

Dependency Walker shows the true FILEVERSION and PRODUCTVERSION version values and not the text string versions. Other applications, like the Windows Properties dialog, show the text string values since that is what the developer of the module wants the average non-technical user to see. For example, you may see only "2.0" in the Windows Properties dialog for a module when its real version is 2.0.5.2034. If you want to know the true version of a file, you should use Dependency Walker and not the Windows Properties dialog.

A great web site for looking up version numbers of modules is the Microsoft DLL Help Database (http://support.microsoft.com/servicedesks/FileVersion/dllinfo.asp). This site has detailed version histories of DLLs and lists what products were shipped with each version. This database can be helpful in tracking down version problems.

Types of Dependencies Handled By Dependency Walker

There are several ways a module can be a dependent of another module:

1. **Implicit Dependency** (also known as a load-time dependency or sometimes incorrectly referred to as static dependency): Module A is implicitly linked with a LIB file for Module B at compile/link time, and Module A's source code actually calls one or more functions in Module B. Module B is a load time dependency of Module A and will be loaded into memory regardless if Module

A actually makes a call to Module B at run-time. Module B will be listed in Module A's import table.

2. **Delay-load Dependency**: Module A is delay-load linked with a LIB file for Module B at compile/link time, and Module A's source code actually calls one or more functions in Module B. Module B is a dynamic dependency and will only be loaded if Module A actually makes a call to Module B at run-time. Module B will be listed in Module A's delay-load import table.
3. **Forward Dependency**: Module A is linked with a LIB file for Module B at compile/link time, and Module A's source code actually calls one or more functions in Module B. One of the functions called in Module B is actually a forwarded function call to Module C. Module B and Module C are both dependencies of Module A, but only Module B will be listed in Module A's import table.
4. **Explicit Dependency** (also known as a dynamic or run-time dependency): Module A is not linked with Module B at compile/link time. At runtime, Module A dynamically loads Module B via a LoadLibrary type function. Module B becomes a run time dependency of Module A, but will not be listed in any of Module A's tables. This type of dependency is common with OCXs, COM objects, and Visual Basic applications.
5. **System Hook Dependency** (also known as an injected dependency): This type of dependency occurs when another application hooks a specific event (like a mouse event) in a process. When that process produces that event, the OS can inject a module into the process to handle the event. The module that is injected into the process is not really a dependent of any other module, but does resides in that process' address space.

Dependency Walker fully supports modules loaded by all of the above techniques. Case 1, 2, and 3 can easily be detected by just opening a module in Dependency Walker. Case 4 and 5 require runtime profiling, a new feature in Dependency Walker 2.0. For more information on profiling, see the Using Application Profiling to Detect Dynamic Dependencies section.

Using Application Profiling to Detect Dynamic Dependencies

Dependency Walker version 2.0 adds application profiling, a technique used to watch a running application to see what modules it loads. This allows Dependency Walker to detect dynamically loaded modules that are not necessarily reported in any on the import tables of other modules. Dependency Walker's profiler can also detect when a module fails to initialize, which often results in the "The application failed to initialize properly" error.

When a module is first opened by Dependency Walker, it is immediately scanned for all implicit, delay-load, and forwarded dependencies (for more information on dependency types, see the Types of Dependencies Handled By Dependency Walker section). Once all the modules have been scanned, the results are displayed. In addition to these known dependencies, modules are free to load other modules at run-time without any prior warning to the operating system. These types of dependencies are known as dynamic or explicit dependencies. There is really no way to detect dynamic dependencies without actually running the application and watching it to see what modules it loads at run-time. This is exactly what Dependency Walker's application profiling does.

For profiling to work, the module you open in Dependency Walker has to be an executable file (usually ends with .EXE) that is designed to run on the system you are working with. If not, the Start Profiling menu option and toolbar button will not be enabled. When you choose to profile an application, your application should begin to run. As your application runs, Dependency Walker will gather information and log it to the Log View, as well as update the other views.

It is the job of the user to "exercise" the application to ensure that all dynamic dependencies are found. Usually dynamic dependencies are only loaded when needed. For example, modules related to printing might only be loaded if the application actually prints. In a case like this, if the application does not perform a print while being profiled, then Dependency Walker will not detect those modules related to printing. Other modules might only get loaded if an error occurs in the application. Scenarios like these might be hard to produce. Because of this, **It is impossible to guarantee that all dynamic dependencies are found**, but the more an application is exercised, the better the odds are of finding them.

Dependency Walker's application profiler tracks every module that gets loaded and attempts to determine which module actually requested the file to be loaded. This allows dynamically loaded modules to be inserted into the Module Dependency Tree View as a child of the module that actually loaded the module.

The profiler works by hooking particular function calls in the remote process being profiled. On Windows 95, Windows 98, and Windows Me, only non-system modules can be hooked. The result is that when a system module dynamically loads another module, the profiler cannot tell who the parent module is for the dynamically loaded module. Parentless modules like these will be added to the root of the Module Dependency Tree View. All modules that are loaded due to a system-wide hook will also be added to the root of the Module Dependency Tree View since these types of modules are loaded directly by the OS and have no parent module. Even though Dependency Walker may not be able to detect the parent of a dynamic dependency, it does guarantee that all modules that get loaded by the application will be detected.

One final benefit of the profiler is that it can correct the paths of any modules that may have been incorrectly determined during the initial implicit module scan.

When you first open a module in Dependency Walker, it recursively scans all the import and export tables of modules to build the initial module hierarchy. Only file names are stored in these tables, so dependency walker uses the rules you have set up in the Module Search Order Dialog to determine the full path to each module. During profiling, Dependency Walker examines the real path of each module as they load and compares them to the modules in the tree. If a module loads from a different path than Dependency Walker expected it to load from, then it will update the module hierarchy and other views to reflect the change.

How to Interpret Warnings and Errors in Dependency Walker

Dependency Walker may generate many warnings and errors for an application. Some errors may cause an application to fail, while others are harmless and can be ignored. Most failures fit into one of two categories: load-time failures or run-time failures.

A load-time failure means that an application or module didn't even have a chance to run. In more technical terms, this usually means that the entry-point to a module was never called since the operating system couldn't load all the required modules. This can occur if an implicit or forward dependency could not be found or was missing a needed function (for more information on dependency types, see the Types of Dependencies Handled By Dependency Walker section). You will also encounter a load-time failure if the application attempts to load a corrupt or non-Windows module, a module for a different CPU type then you are using, or a 16-bit module into a 32-bit application. Here are some common load-time error messages:

> The dynamic link library BAR.DLL could not be found in the specified path...
>
> The procedure entry point FOO could not be located in the dynamic link library BAR.DLL.
>
> The application or DLL BAR.DLL is not a valid Windows image.
>
> The application failed to initialize properly.
>
> Initialization of the dynamic link library BAR.DLL failed. The process is terminating abnormally.
>
> The image file BAR.EXE is valid, but is for a machine type other than the current machine.

Most load-time problems can be immediately detected by Dependency Walker. When you first open a module in Dependency Walker, it scans that module for all implicit, forward, and delay-load dependencies. Implicit and forward dependencies are required by the operating system in order for the application to run. If any implicit or forward dependencies are missing or have errors, then it is likely that the application will encounter a load-time failure if run. Delay-load dependencies are not required by the operating system at load-time, so errors or warning with delay load dependencies may or may not cause problems.

Run-time dependencies are modules that an application loads after it has initialized and begun to run. This is usually achieved by calling one of the LoadLibrary type functions. Once a module has been loaded, an application can call the GetProcAddress function to locate a specific function in the newly loaded module. Dependency Walker can track all these calls and reports any failures. However, if the application is prepared to handle the failure, then the warning can be ignored.

There are many reasons for using run-time dependencies. First, they can increase load-time performance since an application can delay the loading of certain modules that may not be needed until later. For example, if an application uses a DLL related to printing, that DLL might not get loaded unless you actually print something from the application. Second, they can be used in cases where a module, or a function within a module, may not exist. For example, an application might need to call a Windows NT specific function when running on Windows NT, but the module or function does not exist on Windows 9x. If the application were to implicitly link to the module that the function lives in, then a load-time failure would occur on Windows 9x since the operating system would not be able to locate the function at load-time. By making it a run-time dependency, the application can check to see if the function exists and only call it if it does.

There are two types of run-time dependencies: explicit dependencies (often referred to as dynamic dependencies) and delay-load dependencies. Explicit dependencies can be loaded at anytime during the life of the application with no prior notice. Because of this, the only way to determine what explicit dependencies an application will use is to run the application and watch it to see what it loads (for more information on profiling, see the Using Application Profiling to Detect Dynamic Dependencies section). With explicit dependencies, the application directly calls LoadLibrary and GetProcAddress to do the work.

Delay-load dependencies are actually implemented as explicit dependencies, but a helper library and the linker do most of the work. Most all Windows modules have an "import table" stored in them. This table is built by the linker and used by the operating system to determine the implicit and forward dependencies of a given module. Any module or function in this list that cannot be found will cause the module to fail. If you tell the linker to make a module a delay-load dependency, then instead of storing that module's information in the main import table, it stores it in a separate delay-load import table. At run-time, if a module calls into a delay-load dependency module, the call is trapped by the helper library. This library then uses LoadLibrary to load the module and GetProcAddress to query all the functions referenced in the module. Once this is complete, the call is passed along to the real function and execution resumes without the module that made the call even knowing what just happen. All future calls from that specific module to the delay-loaded module will be made directly into the already loaded module instead of being trapped by the helper library.

The delay-load helper library has a mechanism for notifying the caller if there is a failure. Like failures with explicit dependencies, if the application is prepared for the failure, then this should not be a problem.

To summarize, implicit and forward dependencies are required dependencies that need to exist and have no errors or warnings. Explicit and delay-load dependencies may not need to exist and may not need to export all the functions that the parent module wishes to import from them. However, if an application is not prepared to handle a missing explicit or delay-load module, or a missing function within an explicit or delay-load module, then this can result in a run-time failure of the application. Dependency Walker cannot predict if an application plans to handle failures, so it just warns you of all potential problems. If you find an application runs smoothly, then you can probably ignore most all warnings. However, if your application were to fail, then the warnings may provide some insight as to what caused the failure.

There is one other type of warning generated by Dependency Walker while profiling that is worth mentioning. This is related to first and second exceptions. When an exception (like an access violation) occurs in an application, the application is given a chance to handle the exception. These are known as first chance exceptions. If the application handles the exception, then there should be no problem and the exception can probably be ignored. If the application does not handle the first chance exception, then it turns into a second chance exception, which are usually fatal to the application. When a second chance exception occurs, the operating system usually puts up a dialog telling you that the application has crashed and needs to exit.

Dependency Walker always logs second chance exceptions and can optionally log first chance exceptions. Many applications routinely generate first chance exceptions and handle them. This is not a sign of a bad application since there are many legitimate reasons to generate first chance exceptions and handle them.

Dependency Walker Path (DWP) Files

Dependency Walker Path (DWP) files are used to define how Dependency Walker locates modules on your system. By default, Dependency Walker is set up to simulate the search algorithm that the operating system uses to locate modules. However, you can override this default and set up your own custom search criteria. See the Module Search Order Dialog section for more information.

DWP files are usually created by configuring a search order in the Module Search Order Dialog, and then choosing save from that dialog to save the search order to a DWP file. This DWP file can then be loaded at a later time from the Module Search Order Dialog or from the Command Line.

DWP files can also be created and edited by hand. DWP files are simply text files that contain a list of search groups. The following is a list of supported keywords:

SxS	Side-by-Side components
KnownDLLs	The system's "KnownDLLs" list
AppDir	The application directory
32BitSysDir	The 32-bit system directory
16BitSysDir	The 16-bit system directory (Windows NT/2000/XP/2003/Vista/+ only)
OSDir	The system's root OS directory
AppPath	The application's registered "App Paths" directories
SysPath	The system's "PATH" environment variable directories
UserDir	A user defined directory

Each keyword must be on a line by itself. All keywords are case insensitive. Except for the UserDir keyword, no keyword can be specified more than once. The UserDir keyword is a special keyword that also requires a directory path. The syntax for it is:

```
UserDir c:\path\to\some\directory\
```

You may use system variables in the path as well. For example:

```
UserDir %build_directory%\%target_cpu%\debug\
```

All spaces and empty lines in the DWP file are ignored, except for spaces that are part of a directory path. No quotes should be used with any of the keywords or paths. You may add comments to the file by starting a line with a colon (:), semicolon (;), forward slash (/), single quote ('), or pound (#).