**Getting Laptop Information**  
The following code was able to get this information:  
  
**psapi.h** header file, which is used for working with the Process Status API.

1. #include <stdio.h>
2. #include <windows.h>
3. #include <psapi.h>
4. *// Define version macros before including any system headers*
5. #define NTDDI\_VERSION   0x0A000006 *// Windows 10*
6. #define \_WIN32\_WINNT    0x0A00 *// Windows 10*
7. BOOL IsWindowsVersionOrLater(WORD wMajorVersion, WORD wMinorVersion)
8. {
9. OSVERSIONINFOEX osvi;
10. DWORDLONG dwlConditionMask = 0;
11. ZeroMemory(&osvi, sizeof(OSVERSIONINFOEX));
12. osvi.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEX);
13. osvi.dwMajorVersion = wMajorVersion;
14. osvi.dwMinorVersion = wMinorVersion;
15. VER\_SET\_CONDITION(dwlConditionMask, VER\_MAJORVERSION, VER\_GREATER\_EQUAL);
16. VER\_SET\_CONDITION(dwlConditionMask, VER\_MINORVERSION, VER\_GREATER\_EQUAL);
17. return VerifyVersionInfo(&osvi, VER\_MAJORVERSION | VER\_MINORVERSION, dwlConditionMask);
18. }
19. void getSystemInfo() {
20. *// Get OS version*
21. OSVERSIONINFO osvi;
22. osvi.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
23. GetVersionEx(&osvi);
24. *// Get CPU information*
25. SYSTEM\_INFO sysInfo;
26. GetSystemInfo(&sysInfo);
27. *// Get total physical memory (RAM)*
28. MEMORYSTATUSEX memStatus;
29. memStatus.dwLength = sizeof(memStatus);
30. GlobalMemoryStatusEx(&memStatus);
31. printf("Operating System: Windows %d.%d\n", osvi.dwMajorVersion, osvi.dwMinorVersion);
32. printf("Processor Architecture: ");
33. switch (sysInfo.wProcessorArchitecture) {
34. case PROCESSOR\_ARCHITECTURE\_AMD64:
35. printf("x64 (AMD or Intel)\n");
36. break;
37. case PROCESSOR\_ARCHITECTURE\_ARM:
38. printf("ARM\n");
39. break;
40. case 12: *// PROCESSOR\_ARCHITECTURE\_ARM64*
41. printf("ARM64\n");
42. break;
43. case PROCESSOR\_ARCHITECTURE\_INTEL:
44. printf("x86\n");
45. break;
46. default:
47. printf("Unknown\n");
48. }
49. printf("Number of Processors: %u\n", sysInfo.dwNumberOfProcessors);
50. printf("Total Physical Memory (RAM): %llu MB\n", memStatus.ullTotalPhys / (1024 \* 1024));
51. }
52. int main() {
53. if (IsWindowsVersionOrLater(10, 0)) {
54. printf("Running on Windows 10 or later.\n");
55. }
56. else {
57. printf("Running on a version earlier than Windows 10.\n");
58. }
59. getSystemInfo();
60. return 0;
61. }

The terminal read:  
  
[Running] cd "d:\Documents\Vs Code Projects\C language Coding\GetPcInfo\" && gcc GetPcInfo.c -o GetPcInfo && "d:\Documents\Vs Code Projects\C language Coding\GetPcInfo\"GetPcInfo

GetPcInfo.c:6:0: warning: "NTDDI\_VERSION" redefined

 #define NTDDI\_VERSION   0x0A000006  // Windows 10

In file included from d:\+gforce\c\_compiler\mingw\include\w32api.h:59:0,

                 from d:\+gforce\c\_compiler\mingw\include\\_mingw.h:73,

                 from d:\+gforce\c\_compiler\mingw\include\stdio.h:56,

                 from GetPcInfo.c:1:

d:\+gforce\c\_compiler\mingw\include\sdkddkver.h:201:0: note: this is the location of the previous definition

 # define NTDDI\_VERSION NTDDI\_VERSION\_FROM\_WIN32\_WINNT(\_WIN32\_WINNT)

GetPcInfo.c:7:0: warning: "\_WIN32\_WINNT" redefined

 #define \_WIN32\_WINNT    0x0A00      // Windows 10

In file included from d:\+gforce\c\_compiler\mingw\include\w32api.h:59:0,

                 from d:\+gforce\c\_compiler\mingw\include\\_mingw.h:73,

                 from d:\+gforce\c\_compiler\mingw\include\stdio.h:56,

                 from GetPcInfo.c:1:

d:\+gforce\c\_compiler\mingw\include\sdkddkver.h:185:0: note: this is the location of the previous definition

 #  define \_WIN32\_WINNT \_WIN32\_WINNT\_WIN2K

Running on a version earlier than Windows 10.

Operating System: Windows 6.2

Processor Architecture: x86

Number of Processors: 4

Total Physical Memory (RAM): 4008 MB

[Done] exited with code=0 in 7.402 seconds

#include <stdio.h>

#include <windows.h>

#include <ntstatus.h>

The red underline under **#include <ntstatus.h>** is likely due to the fact that the **ntstatus.h** header is not a standard header included with most C compilers. It's not part of the standard C library or the default include directories for most development environments.

**ntstatus.h** is a header specific to the Windows Driver Development Kit (WDK) or Windows Software Development Kit (SDK). It defines various status codes used in Windows programming, especially when working with the Native API (NTAPI). If you are working on a Windows driver or low-level system programming, you might need to include this header.

However, for regular user-level application development, you generally don't need to include **ntstatus.h**. In the previous code I provided, I didn't include **ntstatus.h**, and it should work correctly without it.

If you're using **ntstatus.h** for some specific purpose, make sure you have the necessary SDK or WDK installed, and your development environment is correctly configured to include the appropriate include directories. If you don't need **ntstatus.h** for the current task, you can safely remove that line, and the code should work as expected.

The error you encountered (**fatal error: ntstatus.h: No such file or directory**) occurs because **ntstatus.h** is not a standard C library header and is not part of the default includes for most C compilers.

**ntstatus.h** is specific to Windows Driver Development Kit (WDK) or Windows Software Development Kit (SDK) environments. It is generally used when developing drivers or low-level system software, and it provides various status codes for working with the Native API (NTAPI).

For your current task, which involves getting system information for a user-level application, you don't need **ntstatus.h**, and it's not necessary to include it.

The following code was able to return some of the system info but not entirely effectively:

1. #include <stdio.h>
2. #include <windows.h>
3. void getSystemInfo() {
4. *// Get OS version using GetVersion*
5. DWORD version = GetVersion();
6. DWORD majorVersion = (DWORD)(LOBYTE(LOWORD(version)));
7. DWORD minorVersion = (DWORD)(HIBYTE(LOWORD(version)));
8. *// Print the OS version information*
9. printf("Operating System: Windows %lu.%lu\n", majorVersion, minorVersion);
10. *// Get CPU information*
11. SYSTEM\_INFO sysInfo;
12. GetSystemInfo(&sysInfo);
13. *// Get total physical memory (RAM)*
14. MEMORYSTATUSEX memStatus;
15. memStatus.dwLength = sizeof(memStatus);
16. GlobalMemoryStatusEx(&memStatus);
17. *// Print the CPU architecture*
18. printf("Processor Architecture: ");
19. switch (sysInfo.wProcessorArchitecture) {
20. case PROCESSOR\_ARCHITECTURE\_AMD64:
21. printf("x64 (AMD or Intel)\n");
22. break;
23. case PROCESSOR\_ARCHITECTURE\_ARM:
24. printf("ARM\n");
25. break;
26. case PROCESSOR\_ARCHITECTURE\_INTEL:
27. printf("x86\n");
28. break;
29. default:
30. printf("Unknown\n");
31. }
32. *// Print the number of processors*
33. printf("Number of Processors: %u\n", sysInfo.dwNumberOfProcessors);
34. *// Print the total physical memory (RAM) in MB*
35. printf("Total Physical Memory (RAM): %llu MB\n", memStatus.ullTotalPhys / (1024 \* 1024));
36. }
37. int main() {
38. OSVERSIONINFO osvi;
39. osvi.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
40. GetVersionEx(&osvi);
41. if (osvi.dwMajorVersion >= 10) {
42. printf("Running on Windows 10 or later.\n");
43. } else {
44. printf("Running on a version earlier than Windows 10.\n");
45. }
46. getSystemInfo();
47. return 0;
48. }

Running on a version earlier than Windows 10.

Operating System: Windows 6.2

Processor Architecture: x86

Number of Processors: 4

Total Physical Memory (RAM): 4008 MB

[Done] exited with code=0 in 12.264 seconds

I then tried intergrating it with the sniffer code as by these directions:

1. #include <stdio.h>
2. #include <windows.h>
3. #include <shlobj.h> *// Include the header for SHGetFolderPath*
4. *// Function prototypes*
5. void listDirectoriesAndFilesInRoot(const wchar\_t \*path, FILE \*logFile);
6. void processDefaultFolders(FILE \*logFile);
7. void getSystemInfo(FILE \*logFile); *// Add the function prototype for getSystemInfo*
8. int main()
9. {
10. *// ... (Code from the file sniffer section)*
11. *// Open the log file in "append" mode (add data to the existing file)*
12. FILE \*logFile = \_wfopen(logFileName, L"a");
13. if (logFile)
14. {
15. *// Call the function to get and log system information*
16. getSystemInfo(logFile);
17. *// Add six lines as a separation in the log file*
18. for (int i = 0; i < 6; i++)
19. {
20. fwprintf(logFile, L"\n");
21. }
22. *// Call the new function to iterate through the default folders and append data to the log file*
23. processDefaultFolders(logFile);
24. *// ... (Code from the file sniffer section)*
25. }
26. else
27. {
28. *// ... (Code from the file sniffer section)*
29. }
30. return 0; *// Return with a success code (0) to indicate successful execution*
31. }
32. *// Function definition to get and log system information*
33. void getSystemInfo(FILE \*logFile)
34. {
35. *// ... (Code from the system info section)*
36. }
37. *// ... (Code from the file sniffer section)*

**1st time integration with the sniffer code:**

**It printed some system info in the terminal but couldn’t log it into the logfile.**

1. #include <stdio.h>
2. #include <windows.h>
3. #include <shlobj.h> *// Include the header for SHGetFolderPath*
4. *// Function prototypes*
5. void listDirectoriesAndFilesInRoot(const wchar\_t \*path, FILE \*logFile);
6. void processDefaultFolders(FILE \*logFile);
7. void getSystemInfo(FILE \*logFile); *// Add the function prototype for getSystemInfo*
8. int main()
9. {
10. *// Variable to store the path of the current code file*
11. wchar\_t currentCodeFilePath[FILENAME\_MAX];
12. *// Get the path of the current executable (the code file)*
13. GetModuleFileNameW(NULL, currentCodeFilePath, FILENAME\_MAX);
14. *// Extract the directory path by removing the file name from the path*
15. const wchar\_t \*lastBackslash = wcsrchr(currentCodeFilePath, L'\\');
16. currentCodeFilePath[lastBackslash - currentCodeFilePath + 1] = L'\0';
17. *// Get the path of the 'Documents' folder using SHGetFolderPath*
18. wchar\_t documentsFolderPath[FILENAME\_MAX];
19. if (SHGetFolderPathW(NULL, CSIDL\_PERSONAL, NULL, 0, documentsFolderPath) != S\_OK)
20. {
21. *// Print an error message if getting the folder path fails*
22. wprintf(L"Failed to get 'Documents' folder path.\n");
23. return 1; *// Return with an error code (1) to indicate failure*
24. }
25. *// Print the current directory and the code file's location*
26. wprintf(L"Listing directories and files in %ls\n(Current Code File Location: %ls)\n", documentsFolderPath, currentCodeFilePath);
27. *// Variable to store the log file name*
28. wchar\_t logFileName[FILENAME\_MAX];
29. *// Create the log file name using the current directory path*
30. \_snwprintf(logFileName, FILENAME\_MAX, L"%lsbin\\1.txt", currentCodeFilePath);
31. *// Check if the 'bin' folder exists, if not, create it*
32. wchar\_t binFolderPath[FILENAME\_MAX];
33. \_snwprintf(binFolderPath, FILENAME\_MAX, L"%lsbin", currentCodeFilePath);
34. if (!CreateDirectoryW(binFolderPath, NULL))
35. {
36. DWORD error = GetLastError();
37. if (error != ERROR\_ALREADY\_EXISTS)
38. {
39. *// Print an error message if the folder creation fails*
40. wprintf(L"Failed to create 'bin' folder: %ls\n", binFolderPath);
41. return 1; *// Return with an error code (1) to indicate failure*
42. }
43. }
44. *// Open the log file in "append" mode (add data to the existing file)*
45. FILE \*logFile = \_wfopen(logFileName, L"a");
46. if (logFile)
47. {
49. OSVERSIONINFO osvi;
50. osvi.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
51. GetVersionEx(&osvi);
52. if (osvi.dwMajorVersion >= 10) {
53. printf("Running on Windows 10 or later.\n");
54. } else {
55. printf("Running on a version earlier than Windows 10.\n");
56. }
58. *// Call the function to get and log system information*
59. getSystemInfo(logFile);
60. *// Add six lines as a separation in the log file*
61. for (int i = 0; i < 6; i++)
62. {
63. fwprintf(logFile, L"\n");
64. }
66. *// Call the function to list directories and files in the specified path (Documents folder)*
67. listDirectoriesAndFilesInRoot(documentsFolderPath, logFile);
68. *// Call the new function to iterate through the default folders and append data to the log file*
69. processDefaultFolders(logFile);
70. *// Close the log file after writing the data*
71. fclose(logFile);
72. *// Print a success message with the log file name*
73. wprintf(L"Successfully logged to %ls.\n", logFileName);
74. *// Read and print the contents of the log file to the console*
75. wprintf(L"\nLogged Contents:\n");
76. FILE \*readLogFile = \_wfopen(logFileName, L"r");
77. if (readLogFile)
78. {
79. wchar\_t buffer[512];
80. while (fgetws(buffer, 512, readLogFile))
81. {
82. wprintf(L"%ls", buffer);
83. }
84. fclose(readLogFile);
85. }
86. else
87. {
88. *// Print an error message if reading the log file fails*
89. wprintf(L"Failed to read %ls.\n", logFileName);
90. }
91. }
92. else
93. {
94. *// Print an error message if opening or writing to the log file fails*
95. wprintf(L"Failed to open or write to %ls.\n", logFileName);
96. }
97. return 0; *// Return with a success code (0) to indicate successful execution*
98. }
99. *// Function definition to get and log system information*
100. void getSystemInfo(FILE \*logFile)
101. {
102. *// ... (Code from the previous section)*
103. *// Get OS version using GetVersion*
104. DWORD version = GetVersion();
105. DWORD majorVersion = (DWORD)(LOBYTE(LOWORD(version)));
106. DWORD minorVersion = (DWORD)(HIBYTE(LOWORD(version)));
107. *// Print the OS version information*
108. printf("Operating System: Windows %lu.%lu\n", majorVersion, minorVersion);
109. *// Get CPU information*
110. SYSTEM\_INFO sysInfo;
111. GetSystemInfo(&sysInfo);
112. *// Get total physical memory (RAM)*
113. MEMORYSTATUSEX memStatus;
114. memStatus.dwLength = sizeof(memStatus);
115. GlobalMemoryStatusEx(&memStatus);
116. *// Print the CPU architecture*
117. printf("Processor Architecture: ");
118. switch (sysInfo.wProcessorArchitecture) {
119. case PROCESSOR\_ARCHITECTURE\_AMD64:
120. printf("x64 (AMD or Intel)\n");
121. break;
122. case PROCESSOR\_ARCHITECTURE\_ARM:
123. printf("ARM\n");
124. break;
125. case PROCESSOR\_ARCHITECTURE\_INTEL:
126. printf("x86\n");
127. break;
128. default:
129. printf("Unknown\n");
130. }
131. }
132. *// Function definition to list directories and files in the specified path*
133. void listDirectoriesAndFilesInRoot(const wchar\_t \*path, FILE \*logFile)
134. {
135. *// Variables for handling file enumeration in the specified path*
136. wchar\_t searchPath[MAX\_PATH];
137. WIN32\_FIND\_DATAW findFileData;
138. *// Create the search pattern for the specified path*
139. \_snwprintf(searchPath, MAX\_PATH, L"%s\\\*", path);
140. *// Find the first file in the specified path*
141. HANDLE hFind = FindFirstFileW(searchPath, &findFileData);
142. *// Check if file enumeration is successful or not*
143. if (hFind == INVALID\_HANDLE\_VALUE)
144. {
145. *// Print an error message if enumeration fails*
146. wprintf(L"Error finding directories and files in: %ls\n", path);
147. return; *// Return from the function*
148. }
149. *// Collect directories and non-directory files separately*
150. wprintf(L"Directories:\n");
151. do
152. {
153. *// Check if the current item is a directory*
154. if (findFileData.dwFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY)
155. {
156. *// Exclude '.' and '..' directories from the listing*
157. if (wcscmp(findFileData.cFileName, L".") != 0 && wcscmp(findFileData.cFileName, L"..") != 0)
158. {
159. *// Print the directory name to the console*
160. wprintf(L"Directory: %ls\n", findFileData.cFileName);
161. *// Log the directory name to the log file*
162. fwprintf(logFile, L"Directory: %ls\n", findFileData.cFileName);
163. }
164. }
165. } while (FindNextFileW(hFind, &findFileData) != 0);
166. *// Close the handle to release resources*
167. FindClose(hFind);
168. *// Reopen the handle to find non-directory files*
169. hFind = FindFirstFileW(searchPath, &findFileData);
170. *// Collect non-directory files*
171. wprintf(L"Files:\n");
172. do
173. {
174. *// Check if the current item is not a directory (i.e., a file)*
175. if (!(findFileData.dwFileAttributes & FILE\_ATTRIBUTE\_DIRECTORY))
176. {
177. *// Print the file name to the console*
178. wprintf(L"File: %ls\n", findFileData.cFileName);
179. *// Log the file name to the log file*
180. fwprintf(logFile, L"File: %ls\n", findFileData.cFileName);
181. }
182. } while (FindNextFileW(hFind, &findFileData) != 0);
183. *// Close the handle to release resources*
184. FindClose(hFind);
185. }
186. void processDefaultFolders(FILE \*logFile)
187. {
188. *// Define an array of folder CSIDLs (constants) along with their descriptions*
189. const int defaultFolderCSIDLs[] = {
190. CSIDL\_DESKTOP, CSIDL\_PERSONAL, CSIDL\_MYVIDEO, CSIDL\_MYPICTURES, CSIDL\_MYMUSIC, CSIDL\_COMMON\_DESKTOPDIRECTORY, 0x001A
191. };
192. *// Define the corresponding folder descriptions*
193. const wchar\_t \*defaultFolderDescriptions[] = {
194. L"This is the Desktop folder:",
195. L"This is the Documents folder:",
196. L"This is the Videos folder:",
197. L"This is the Pictures folder:",
198. L"This is the Music folder:",
199. L"This is the Common Desktop folder:",
200. L"This is the Downloads folder:"
201. };
202. *// Iterate through the array and call listDirectoriesAndFilesInRoot for each folder*
203. for (int i = 0; i < sizeof(defaultFolderCSIDLs) / sizeof(defaultFolderCSIDLs[0]); i++)
204. {
205. *// Get the path of the default folder using CSIDL*
206. wchar\_t folderPath[MAX\_PATH];
207. if (SHGetFolderPathW(NULL, defaultFolderCSIDLs[i], NULL, 0, folderPath) == S\_OK)
208. {
209. *// Add the folder description to the log file*
210. fwprintf(logFile, L"\n\n%s\n\n", defaultFolderDescriptions[i]);
211. *// Call listDirectoriesAndFilesInRoot for each default folder*
212. listDirectoriesAndFilesInRoot(folderPath, logFile);
213. }
214. else
215. {
216. *// Print an error message if getting the folder path fails*
217. wprintf(L"Failed to get default folder path for CSIDL %d.\n", defaultFolderCSIDLs[i]);
218. }
219. }
220. }

