

# Undocumented CreateProcess

[Home](#) > [Tutorials](#)

james - Sat, 08/30/2008 - 10:27

This tutorial forms a part of a new series which will concentrate on some of the non-GUI related issues in Windows programming. The subject of this tutorial will be the CreateProcess win32 API call. The tutorial is split into several sections, each one describing a neat fact about CreateProcess which you can use to your advantage. What I will describe cannot be found in Microsoft documentation, but has been discovered by many people over the years through a lot of experimentation. All of the information collected here was found in various sources - especially older publications such as "Windows Developer Journal" from the mid-90s and also old USENET postings.

Before I start properly let's review what CreateProcess does, and how to use it in your code. If you are familiar with CreateProcess then please skip down to the first [undocumented tip](#).

```
BOOL CreateProcess(
    LPCTSTR          lpApplicationName,    // name of executable module
    LPCTSTR          lpCommandLine,       // command line string
    LPSECURITY_ATTRIBUTES lpProcessAttributes, // SD
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD
    BOOL             fInheritHandles,      // handle inheritance option
    DWORD            dwCreationFlags,      // creation flags
    LPVOID           lpEnvironment,        // new environment block
    LPCTSTR          lpCurrentDirectory,   // current directory name
    LPSTARTUPINFO    lpStartupInfo,       // startup information
    LPPROCESS_INFORMATION lpProcessInformation // process information
);
```

This function definition (above) was taken directly out of MSDN. The function is a bit unwieldy and difficult to understand at first but process creation is a complicated business. Fortunately most of the parameters in CreateProcess can be omitted and the standard way of creating a new process looks like this:

```
STARTUPINFO    si = { sizeof(si) };
PROCESS_INFORMATION pi;
char           szExe[] = "cmd.exe";

if(CreateProcess(0, szExe, 0, 0, FALSE, 0, 0, 0, &si, &pi))
{
    // optionally wait for process to finish
    WaitForSingleObject(pi.hProcess, INFINITE);

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

The example above simply starts a new instance of cmd.exe and lets it run. However, there are a lot of options available in the call to CreateProcess which allow you to control how the program is started. Some of these options are specified directly as parameters, but most of the options are specified inside the **STARTUPINFO** structure which is passed as the **9th** parameter:

```
typedef struct
{
    DWORD    cb;
    LPCTSTR  lpReserved;
    LPCTSTR  lpDesktop;
    LPCTSTR  lpTitle;
    DWORD    dwX;
    DWORD    dwY;
    DWORD    dwXSize;
    DWORD    dwYSize;
    DWORD    dwXCountChars;
    DWORD    dwYCountChar;
    DWORD    dwFillAttribute;
    DWORD    dwFlags;
    WORD     wShowWindow;
    WORD     cbReserved2;
    LPBYTE   lpReserved2;
    HANDLE   hStdInput;
    HANDLE   hStdOutput;
    HANDLE   hStdError;
} STARTUPINFO;
```

The STARTUPINFO structure is also documented in MSDN, but some interesting information has been omitted by Microsoft which we will start to explore right now. The whole thrust of this article will centre around this STARTUPINFO structure, and the three "reserved" members of the STARTUPINFO structure - that is, **lpReserved**, **lpReserved2** and **cbReserved2**.

Later on in this article I will explain what these three member variables are really used for, but first of all we will look at the dwFlags member variable and what it does. A quick reminder from MSDN is appropriate here:

Name	Value	Meaning
STARTF_USESHOWWINDOW	0x01	If this value is not specified, the wShowWindow member is ignored.

STARTF_USESIZE	0x02	If this value is not specified, the dwXSize and dwYSize members are ignored.
STARTF_USEPOSITION	0x04	If this value is not specified, the dwX and dwY members are ignored.
STARTF_USECOUNTCHARS	0x08	If this value is not specified, the dwXCountChars and dwYCountChars members are ignored.
STARTF_USEFILLATTRIBUTE	0x10	If this value is not specified, the dwFillAttribute member is ignored.
STARTF_RUNFULLSCREEN	0x20	Indicates that the process should be run in full-screen mode, rather than in windowed mode.
STARTF_FORCEONFEEDBACK	0x40	Indicates that the cursor is in feedback mode for two seconds after CreateProcess is called.
STARTF_FORCEOFFFEEDBACK	0x80	Indicates that the feedback cursor is forced off while the process is starting. The normal cursor is displayed.
STARTF_USESTDHANDLES	0x100	Sets the standard input, standard output, and standard error handles for the process to the handles specified in the hStdInput, hStdOutput, and hStdError members of the STARTUPINFO structure.

These nine values (or bit-flags) can be specified on their own or together by ORING them. The table above is pretty boring because apart from the STARTF\_USESTDHANDLES flag none of them are particularly useful. However, look at the range of values that the bit-flags use: the numbers go from 0x01 to 0x100 - this is only 9 bit-positions. There are 32bits in a single DWORD, so that leaves 23 possible bit-flags which haven't been defined yet.

That's more than enough background to get us started, so let's move onto more interesting things.

## Detect if an executable was started from a Short-Cut

OK, the first trick I'm going to show you is how any win32 application can detect if it was started from a shortcut (i.e. by double-clicking a shortcut link), or directly via Windows explorer or the Run dialog. This trick is so simple it's amazing Microsoft hasn't documented it:

There is an undocumented STARTUPINFO flag which I have called **STARTF\_TITLESHORTCUT**. This bit-flag has the value 0x800. Windows sets this flag when an executable has been started via a short-cut. So it is possible for any program to check if it was started this way by examining it's own STARTUPINFO structure to see if this flag was specified:

```
// Returns TRUE if started through a shortcut, FALSE if not.
// Also returns the name of the shortcut file (if any)
BOOL GetShortcutName(TCHAR *szLinkName, UINT nLinkNameSize)
{
    STARTUPINFO si = { sizeof(si) };

    GetStartupInfo(&si);

    if(si.dwFlags & STARTF_TITLESHORTCUT)
    {
        lstrcpyn(szLinkName, si.lpTitle, nLinkNameSize);
        return TRUE;
    }

    return FALSE;
}
```

It is a simple matter to check for this undocumented flag, but another important piece of information should be explained. When the STARTF\_TITLESHORTCUT flag is set, the **lpTitle** member of STARTUPINFO points to a string which contains the full path to the shortcut file used to start the current executable. So imagine that there is a short-cut on your desktop which launches notepad.exe. When notepad.exe starts, it's **STARTUPINFO::lpTitle** contains the following text:

```
C:\Documents and Settings\James\Desktop\Notepad.lnk
```

Pretty cool huh? Well I hope I've whet the appetite because we can move onto the next undocumented nugget!

## Specify which monitor a process should start on

The next undocumented tip uses another unknown STARTUPINFO flag. This flag has the value 0x400, and I have given it the name **STARTF\_MONITOR**.

When this flag is specified in the dwFlags member, the STARTUPINFO's **hStdOutput** member is used to specify a handle to a monitor, on which to start the new process. This monitor handle can be obtained by any of the multiple-monitor display functions (i.e. EnumDisplayMonitors, MonitorFromPoint, MonitorFromWindow etc).

```
BOOL CALLBACK EnumMonitorProc(HMONITOR hMonitor, HDC hdc, LPRECT rect, LPARAM data)
{
    STARTUPINFO      si = { sizeof(si) };
    PROCESS_INFORMATION pi;
    char             szExe[] = "sol.exe";

    // specify which monitor to use
    si.dwFlags      = STARTF_MONITOR;
    si.hStdOutput   = hMonitor;

    // start solitaire on specified monitor
    if(CreateProcess(0, szExe, 0, 0, FALSE, 0, 0, 0, &si, &pi))
    {
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
    }
}

// call EnumMonitorProc (above) for every installed monitor on the system
void Demo()
{
    EnumDisplayMonitors(NULL, NULL, EnumMonitorProc, 0);
}
```

The example above uses the EnumDisplayMonitors API call as a method of obtaining a monitor handle (one for each monitor installed). The EnumMonitorProc function simply starts a new instance of Solitaire on each monitor in turn.

There are certain restrictions which should be noted here. You may be asking yourself why the above example works, when you know full well that the hStdOutput member should be used for a standard-output pipe. The answer to this question is simple - when the undocumented STARTF\_MONITOR flag is specified, the STARTF\_USESTDHANDLES flag is ignored. This means that these two flags cannot be used together, and the hStdInput, hStdOutput and hStdError handles act differently to how they are described in the documentation.

The next restriction is kind of obvious too: when you start a new process with CreateProcess, it has no concept of monitors, windows, or any other GUI related entities. A windows program actually has to physically create it's windows in order to display it's GUI. This is where the restriction for the STARTF\_MONITOR flag comes into play. When a process calls CreateWindow, it can specify exactly where the window will appear by using the x, y, width and height parameters. This means that it is not possible for one program to specify where a second program's windows should appear unless this second program creates it's windows in a certain way.

```
hWnd = CreateWindow("ClassName",
    "Title Text",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT,           // X coordinate
    CW_USEDEFAULT,         // Y coordinate
    width,                  // width
    height,                 // Height
    0, 0, 0, 0);

ShowWindow(hWnd, SW_SHOWDEFAULT);
```

It is only when a program creates a window using the **CW\_USEDEFAULT** values for the coordinates, does the monitor specified by CreateProcess get used. The CreateWindow API call must obviously look inside the current process's STARTUPINFO structure to see what monitor it should create the window on. The Solitaire card-game was used in the example because it creates it's main window in this manner. I couldn't use Notepad as a demonstration because it didn't work - notepad seems to use actual coordinates to display it's main window rather than the CW\_USEDEFAULT values.

Note that the **ShellExecuteEx** API call makes use of this CreateProcess feature in order to implement it's own monitor settings.

## Starting screen-savers

In old versions of Microsoft's platform SDK documentation there was a flag called **STARTF\_SCREENSAVER** - with the hex value 0x80000000. This flag is not documented anymore. When this flag is specified in CreateProcess' STARTUPINFO structure, the new process is started with NORMAL\_PRIORITY. But, when this new process first calls GetMessage, it's priority is automatically lowered to that of IDLE\_PRIORITY. This functionality seems vaguely useful for a screen-saver, and completely useless for any other situation. Presumably it is intended to allow a screen-saver's startup code to get up and running quickly, and then let the screensaver run "normally" without using too much CPU.

It appears that only the WinLogon process (winlogon.exe) is allowed to use STARTF\_SCREENSAVER when activating the current screensaver, and this flag is next to useless for any other scenario.

## Legacy Program Manager functionality

Do you remember Program Manager from the Windows 3.1 days? Program Manager still exists today even in Windows XP - just goto a command-prompt and type "progman" - the familiar program manager shell will pop up. Even in the days of Windows 3.1 (home and NT editions) the CreateProcess API contained undocumented functionality. I'm going to share this information with you now - even though this information is next to useless, it is still interesting to go over it.

If you look back at the STARTUPINFO structure definition, you can see that it has a member called **lpReserved**. This member has actually been in use for a long time now, but only by the Program Manager when it started executables.

The **STARTUPINFO::lpReserved** member is a pointer to a string buffer in the following format:

```
"dde.#,hotkey.#,ntvdm.#"
```

Whenever a new process is started by Program Manager, it can call GetStartupInfo to see how it was started. The resulting lpReserved member points to a string which contains three comma-separated fields, with the hash-signs (#) representing hexadecimal numbers.

The "dde." item specifies the DDE identifier which the child process can use to communicate with Program Manager. When the child process sends progman a WM\_DDE\_REQUEST message with this id, progman responds with a WM\_DDE\_DATA message. This message contains, amongst other things, the progman description, progman icon index and progman working directory for the child process.

More information on this progman DDE interface can be found in the Microsoft Knowledge Base article 105446.

The "hotkey." item specifies the Program Manager hotkey sequence which was used to activate the process. This is a 16bit hex number, with the low byte representing the ASCII code of the hotkey, and the high byte representing the HOTKEYF\_XXX values used by the WM\_SETHOTKEY message. I have no idea why it is useful for a child process to know this information.

The "ntvdm." item is used to inform the NTVDM process about a program's properties in Program Manager. This is a single hex field which is made up of a combination of bit-flags. The value 1 indicates that there is a current directory, a value of 2 indicates that the item has a hotkey, and a value of 4 indicates that a title has been assigned to the program.

That just about covers the **lpReserved** member. It really isn't a useful feature any more, and even though it is possible to use lpReserved in a call to CreateProcess, no child process is ever going to make use of the information. You can see for yourself how Program Manager uses lpReserved by debugging a child process which has been executed via progman.exe.

*Of course, if you have two applications and want to pass a text string from one app to the second (other than through the command-line) the lpReserved field could be useful.*

## Legacy ShellExecuteEx functionality

The ShellExecuteEx API has been around since Windows NT 3.1 (but not "normal" 3.1). This API call takes a single parameter - a pointer to a SHELLEXECUTEINFO structure. You can look in MSDN to see what this structure looks like (it's pretty boring). There are one or two interesting fields in this structure which are not at all well documented:

The first is the hMonitor member. ShellExecuteEx uses this value to control which monitor a new process should appear on. This is possible because ShellExecuteEx calls CreateProcess internally, and uses the same undocumented feature we covered earlier on (the STARTF\_MONITOR flag).

The next field of interest is the `hIcon` member. MSDN documents this as being a handle to an icon for the new process, but I believe it might be an icon-index rather than an icon. I haven't been able to make use of this feature, and as far as I can tell only console applications running under Windows NT 3.1/3.5 made use of this feature. However, `ShellExecuteEx` calls `CreateProcess` with another undocumented setting. Again this is a new `STARTUPINFO` flag (I call it `STARTF_ICON`). Strangely this flag has the same value as the `STARTF_MONITOR` flag (0x400), and like before the `STARTUPINFO::hStdOutput` member is used to store the icon handle/index/whatever it is. You can observe this quite simply by writing a small program that calls `ShellExecuteEx` which specifies an icon for the new process, and watch the call to `CreateProcess` in a debugger. (The new program always ignores the icon though).

The last field of interest is the `dwHotKey` value. This is supposed to specify a hot-key for a child process, so that you can activate the application at any time by using the hotkey sequence at the keyboard - however, I have been unable to make this work. Again, `ShellExecuteEx` uses another undocumented `STARTUPINFO` flag. In this case the flag appears in `winbase.h` - it is called `STARTF_USEHOTKEY` and has the value 0x200. When this flag is specified in the `STARTUPINFO::dwFlags` member, the `hStdInput` member is supposed to be used as the hotkey value instead of the standard-input pipe. (See `WM_SETHOTKEY` for more information).

Both the icon and hotkey features of `CreateProcess` are a little strange, firstly because the functionality appears to be duplicated by the `lpReserved` parameter, and secondly the functionality is not present in modern versions of Windows. If anyone knows any more details about this subject I'd be very glad to hear them!

## Pass arbitrary data to a child process!

The last undocumented trick is quite different to the previous ones so I thought I'd save it until last. The `STARTUPINFO` structure contains two members, **`lpReserved2`** and **`cbReserved2`**:

```
WORD    cbReserved2;
LPBYTE  lpReserved2;
```

These two members provide a mechanism for passing arbitrary amounts of data from one process to another, without having to call `VirtualAllocEx / WriteProcessMemory`. The `cbReserved2` member is a 16bit integer and specifies the size of the buffer pointed to by `lpReserved2`. This means that `lpReserved2` can be as big as 65535 bytes.

The example below demonstrates how to pass a buffer from one process to another. When process-B is executed by process-A, it will display a message box saying "Hello from Process A!":

### Process A:

```
int main()
{
    STARTUPINFO      si = { sizeof(si) };
    PROCESS_INFORMATION pi;
    char             buf[4444] = "Hello from Process A!";

    // setup the buffer to pass to child process
    si.lpReserved2 = buf;
    si.cbReserved2 = sizeof(buf);

    // create a new process with this data
    if(CreateProcess(0, szExe, 0, 0, 0, 0, 0, 0, &si, &pi))
    {
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
    }
}
```

### Process B:

```
int mainCRTStartup()
{
    STARTUPINFO si = { sizeof(si) };

    GetStartupInfo(&si);

    // display what process A sent us
    MessageBox(NULL, si.lpReserved2, "Process B", MB_OK);
}
```

So far so good - we have a nice method for passing arbitrary binary data between applications, without having to use the command line. There is a problem though. In the example above process-B must be compiled with absolutely no C-runtime support. The reason is quite complicated but I will explain it now:

The Microsoft C runtime (including Visual Studio.NET) uses the `lpReserved2` feature in its implementation of the C functions: `exec`, `system` and `spawn`. When a new process is executed using these routines, the C-runtime has to be able to give the child process copies of its open file handles (opened using `fopen/open` rather than `CreateFile`).

`lpReserved2` is used as a mechanism to pass this file-handles information between programs compiled using MSVC. Before the `exec/spawn` runtime functions inevitably call `CreateProcess`, the `lpReserved2` buffer is constructed using the following format:

```
DWORD count;
BYTE  flags[count];
HANDLE handles[count];

// in memory these are layed out sequentially:
[ count ][ flags... ][ handles... ]
```

The first field in the `lpReserved2` buffer is a 32bit count of the number of file-handles being passed. Next comes a `BYTE`-array of flags, one for each file-handle. These flags represent the file attributes that were used when the file was opened using `fopen/open` - i.e. read-only, write-append, text-mode, binary-mode etc. Immediately following the flags array is the `HANDLE`-array containing each open file. This array contains the actual file handle identifiers, again there are `handle_count` items in the array.

Any amount of data can follow this structure, up to a maximum of 65536 bytes. The `cbReserved2` member must be set to the length (in bytes) of this structure.

The actual steps need to construct the `lpReserved2` are as follows:

1. Any currently open "runtime" file-handles are enumerated.
2. Each underlying win32 HANDLE is marked as inheritable.
3. The number of file-handles being passed between processes is stored as the first 32bit integer in `lpReserved2`.
4. The attributes of each open file-handle are stored sequentially in the flags BYTE-array.
5. The file-handle integers are stored sequentially in the handles 32bit int-array.
6. Finally `CreateProcess` is called, with the `blInheritHandles` parameter set to `TRUE`.

It is at this stage that the C-runtime becomes important. When the child process executes, as part of its initialization before `main()` is called, the I/O runtime support needs to be initialized. During this initialization the C-runtime calls `GetStartupInfo` and checks to see if an `lpReserved2` buffer has been specified. If `lpReserved2` is not NULL (i.e. it points to valid memory) then the C-runtime parses the contents of the buffer and extracts the file-handles contained within - this is so the file-handles will be available to the new process.

1. Call `GetStartupInfo` and check if `lpReserved2` points to valid data.
2. Extract the first 32bit integer - this is the number of file-handles in the buffer.
3. Initialize the current process's I/O state with this number of open files.
4. Loop over the `flags[]` array in `lpReserved2`.
5. Loop over the `handles[]` array, populating the open-file table.

The problem might now be apparent to the more astute readers. Any program compiled using Microsoft's C-runtime will check its `lpReserved2` when it first starts - it should come as no surprise that this probably represents 90% of the C/C++ Windows programs in existence. The `lpReserved2` member may also be used by other compiler vendors for the same purpose.

Should you wish to use `lpReserved2` in your own programs (using `CreateProcess` instead of `spawn/exec`) you will need to be careful because the `lpReserved2` buffer must be properly constructed. Failure to do so will result in the child processes crashing, or at the very least becoming unstable - the reason being that the child process is expecting to find `lpReserved2` in a particular format.

Getting around this problem is simple. Setting the first 4 bytes in `lpReserved2` to zeros (nulls) indicates that the handle-arrays are empty, and any c-runtime startup code will simply skip this phase. Your arbitrary binary data can come immediately after this zero-marker. The code now looks like this:

#### Process A:

```
int main()
{
    STARTUPINFO      si = { sizeof(si) };
    PROCESS_INFORMATION pi;
    char             buf[4444];

    // construct the lpReserved2 buffer
    *(DWORD *)buf = 0;
    lstrcpy(buf+sizeof(DWORD), "Hello from Process A!");

    // setup the buffer to pass to child process
    si.lpReserved2 = buf;
    si.cbReserved2 = sizeof(buf);

    // create a new process with this data
    if(CreateProcess(0, szExe, 0, 0, 0, 0, 0, 0, &si, &pi))
    {
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
    }
}
```

#### Process B:

```
int main()
{
    STARTUPINFO si = { sizeof(si) };

    GetStartupInfo(&si);

    // display what process A sent us
    if(si.lpReserved2 != NULL)
        MessageBox(NULL, si.lpReserved2 + sizeof(DWORD), "Process B", MB_OK);
}
```

The example above can now be compiled without having to worry about C-runtime considerations.

**Note:** Apparently this method does not work under 64bit Windows Vista.

## CreateProcess Summary

This was more of a programming article than a tutorial. Hopefully you have either found something useful, or have discovered something about `CreateProcess` which you didn't know before. Just to recap here is a summary of the undocumented `CreateProcess` features:

Undocumented `STARTUPINFO` flags:

Name	Value	Meaning
<code>STARTF_USEHOTKEY</code>	0x200	Use the hot-key DWORD specified in the <code>hStdInput</code> member.
<code>STARTF_MONITOR</code>	0x400	Use the <code>HMONITOR</code> handle specified in the <code>hStdOutput</code> member.

STARTF_ICON	0x400	Use the HICON specified in the hStdOutput member (incompatible with STARTF_MONITOR).
STARTF_TITLESORTCUT	0x800	Program was started through a shortcut. The lpTitle contains the shortcut path.
STARTF_SCREENSAVER	0x80000000	Start the program with NORMAL_PRIORITY, then drop to IDLE_PRIORITY.

**lpReserved** - pointer to a string containing the text "dde.#,hotkey.#,ntvdm.#". This only occurs when a process is started through the legacy Program Manager shell. Use this member to pass your own string buffer to a child process.

**lpReserved2** and **cbReserved2** - pointer to a buffer containing arbitrary binary data to pass to the child process. To be safe ensure that the first four bytes of lpReserved2 are always zero. Has issues under 64bit Vista.

I'd really like to hear any comments you have on this article, or any more information which I might have presented incorrectly or omitted entirely.

Attachment	Size
<a href="#">undoc01.zip</a>	1.2 KB

[Login](#) or [register](#) to post comments

[System](#)

