ArgX Documentation

Release 0.2.0

Alastair J. Houghton

Jan 01, 2020

Contents:

CHAPTER 1

Introduction

Windows' handling of command line arguments historically has been poor; even in 2019, Windows still passes arguments as a single command line string, which makes processing arguments error-prone and also results in unreasonable limits on command line lengths.

ArgX aims to fix this situation in a backwards-compatible manner, such that programs that use ArgX will be able to pass arbitrary numbers of arguments cleanly among themselves without worrying about how various metacharacters might be interpreted by subprocesses.

1.1 Background

1.1.1 Argument passing in C

The C standard specifies that programs start running in the `main` function, which takes two parameters, `argc`, the number of arguments, and `argv`, an array of pointers to those arguments.

It doesn't specify exactly how that information gets into your program, however, and historically DOS and DOS-like systems (including Windows) have chosen a different approach to their POSIX brethren.

1.1.2 Typical POSIX approach

On POSIX systems, the shell is responsible for parsing the command line into separate arguments. The upshot of this is that the arguments arrive in subprocesses already separated, and with any quoting or escaping supported by the shell already processed.

The shell is also responsible for globbing (that is, expanding any patterns that the user has entered on the command line, e.g. `*.txt`).

The advantage of this is that the processing of command lines is determined entirely by whichever shell the user is using. The shell is free to do what it pleases, and subprocesses for the most part do not care how it goes about its business.

1.1.3 Typical DOS/DOS-like approach

On DOS and similar systems, the shell is *not* responsible for command line parsing, and instead passes the entire command line, as a string, to the subprocess. This means that subprocesses must take responsibility for quoting and escaping, as well as globbing.

The upshot of this is that support for quoting, escaping and globbing tends to be rather patchy and ad-hoc. Some programs support these features, and often not in every location/situation in the command line.

1.1.4 What does Windows do?

Well, it is perhaps no surprise that Windows takes the DOS-like approach, with one concession, namely that there is a Windows API, CommandLineToArgvW, that will take a flat command line string and parse it into an argument array for you according to some rather odd and somewhat counterintuitive rules.

Since that API was only added in Windows 2000, and since up to that point it was up to the C library's start-up code to do any necessary parsing, there's a good chance that whatever program you're looking at doesn't use the API and also that it parses its command line in some other manner than the one Microsoft clearly expects.

Windows places the command line into a UNICODE_STRING in the RTL_USER_PROCESS_PARAMETERS structure that is pointed to by the Process Environment Block (or PEB for short) that it creates in the new process's address space. This is the origin of the 32,767 character length limit; the UNICODE_STRING structure uses a `USHORT` for its length (in UTF-16 code units). So you're limited to 32,768 of them, including the terminating NUL, no matter what you do.

(You might also see some people mention a limit of 8,192 characters; this is a limit built into the command processor, `CMD.EXE`.)

1.2 ArgX Specification

This is **version 1.0** of this specification.

Processes that support the ArgX protocol are expected to adhere to this specification. They *do not* have to use the ArgX library code from this project to do so.

1.2.1 1. Supporting ArgX in a client

To indicate that your program supports receiving arguments using the ArgX mechanism, it must place a section within its executable image with the name "ArgX". This section must contain the following data:

Offset	Size	Name	Meaning
0	4	`dwMagic`	Magic number (see below)
4	4	`dwArgc`	Argument count
8	4/8	`pszArgv`	Argument vector pointer

The magic number must consist of the bytes:

```
61 78 00 00    a x . .
```

to indicate support for this version of the ArgX specification. The two zero bytes are reserved to indicate future, incompatible, versions of this specification.

Other members of the structure should be set to zero.

When an ArgX-supporting parent process starts a subprocess that itself supports ArgX, the parent will initialise the `dwArgc` and `pszArgv` members, and will change `dwMagic` to hold the following bytes:

```
41 58 00 00    A X . .
```

The child process, by checking the value of the `dwMagic` field, can test whether or not its parent has provided an argument vector using the ArgX mechanism, in which case it *should* use the contents of the argument vector in preference to the flat command line string supplied by the operating system.

1.2.2 2. Supporting ArgX in a parent process

A parent process that supports ArgX can only use the ArgX protocol if it can locate the "ArgX" section in the subprocess's runtime image. It must take care when doing this, and *must* fall back to using the flat command line mechanism if there is any doubt about the subprocess it is starting. In particular, it must check:

- That the subprocess has an ArgX section.
- That the ArgX section is at least large enough to hold the ArgX data mentioned above, noting that for 64-bit processes the `pszArgv` pointer will be eight bytes rather than the four bytes it would be for a 32-bit process.
- That the dwMagic field has been initialised appropriately.
- That the ArgX section is readable and writable, but not executable.

Parent processes using the ArgX mechanism *should* pass an equivalent flat command line, formatted in such a way as to generate the same argument vector if passed to the Windows CommandLineToArgvW API.

If passing a command line that will not fit in the flat command line, a parent process *must* indicate a failure if the subprocess does not support ArgX; if passing to a process that *does* support ArgX, it *should* set the flat command line to:

```
<argv[0]> --ArgX
```

If this is not possible because the first argument is itself too long, it is permissible to pass `NULL` to the CreateProcess API instead of the flat command line. ArgX-supporting subprocesses should not see the flat command line string in most cases anyway.

The parent process is responsible for allocating space in its child process's address space for the argument vector and for the strings to which that vector points. It is also responsible for updating the `dwMagic` field to indicate to the child that ArgX is in use.

A parent process must ensure that the child process does not execute code until the ArgX procedure has been completed. That is, the child process should be able to test whether ArgX is in use the moment it starts up; there *must not* be a race between the parent and child.

Parent processes *should* check for the existence of the ArgX process creation APIs in `kernel32.dll` before performing any processing themselves. This is to allow Microsoft to take over implementation of the ArgX protocol in future, should it so wish.

1.2.3 Change log

Date	Version	Author	Changes
1 Jan 2020	1.0	ajh	Created specification.

1.3 Using this library

The ArgX library code is intended to be easy to use; your program can include the `ArgX.h` header file, and link against either `ArgX32.lib` or `ArgX64.lib` as appropriate. You will also need to link some system libraries; presently the set required is `kernel32.lib`, `shell32.lib`, `shlwapi.lib` and `advapi32.lib`.

You can then make use of the ArgX API functions defined in the header file.

As an alternative, you can copy the relevant source files (in the `src` folder) directly into your own project. All the code here is subject to the MIT License, so this is quite permissible.

Please do not extend or alter the ArgX code in such a way that it deviates from the specification.

1.4 Reference

1.4.1 ArgxCreateProcess function

Creates a new process, running in the security context of the calling process.

This is equivalent to the Windows CreateProcess API.

Syntax

```
BOOL ArgxCreateProcess(
        LPCTSTR               lpApplicationName,
        LPCTSTR*              lpArgv,
        DWORD                 dwArgc,
        LPSECURITY_ATTRIBUTES lpProcessAttributes,
        LPSECURITY_ATTRIBUTES lpThreadAttributes,
        BOOL                  bInheritHandles,
        DWORD                 dwCreationFlags,
        LPVOID                lpEnvironment,
        LPCTSTR               lpCurrentDirectory,
        LPSTARTUPINFO         lpStartupInfo,
        LPPROCESS_INFORMATION lpProcessInformation
     );
```

Parameters

`lpApplicationName` The path to the executable to start. This string is not subject to any path searching, though it may be a relative path. There is no default extension for this parameter.

If this parameter is `NULL`, the executable path will be taken from the first argument, `lpArgv[0]`.

`lpArgv` The argument vector. If `lpApplicationName` is `NULL`, the first element of the argument vector will be used to locate the desired executable.

In that case:

- If `lpArgv[0]` contains path delimiters, it will be treated as a literal path and used directly; otherwise,
- If `lpArgv[0]` does not have an extension, the extension ".exe" will be appended automatically.
- The function will then try to find the executable by looking in the following places:
    1. The directory from which the calling application loaded.

2. The current directory.
3. The Windows System32 directory (as returned by the GetSystemDirectory API).
4. The 16-bit Windows System directory, if present.
5. The Windows directory (as returned by the GetWindowsDirectory API).
6. The directories listed in the `PATH` environment variable.

This function will *not* modify any of the strings in `lpArgv`. Also note that this function does not suffer from the security hole in the CreateProcess API caused by that function's attempt to parse filenames and directory names containing spaces.

`lpArgv` *must not* be `NULL`.

`dwArgc` The number of elements in `lpArgv`, which must be at least one.

`lpProcessAttributes` Describes the desired SECURITY_ATTRIBUTES for the new process. May be `NULL`.

`lpThreadAttributes` Describes the desired SECURITY_ATTRIBUTES for the primary thread of the new process. May be `NULL`.

`bInheritHandles` If `TRUE`, inheritable handles will be inherited by the subprocess. Importantly, inherited handles have the same access rights that they have in the parent process, so this needs to be used with care.

`dwCreationFlags` Flags that control priority class and creation behaviour. See Process Creation Flags for more information.

`lpEnvironment` Points to the environment block for the new process, or `NULL` to inherit the environment of the parent.

`lpCurrentDirectory` If `NULL`, the new process will start with the same current directory as the parent process. Otherwise, must contain the full path to the desired current directory.

`lpStartupInfo` Points to a STARTUPINFO or STARTUPINFOEX structure. May be `NULL`.

`lpProcessInformation` Points to a PROCESS_INFORMATION structure that will be filled in with handles to the process and its primary thread. Note that these handles *must be closed* when no longer needed.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, it will return zero (i.e. `FALSE`), with extended error information supplied via GetLastError.

See also

CreateProcess

1.4.2 ArgxCreateProcessAsUser function

Creates a new process, running in the security context of the user represented by the given token

This is equivalent to the Windows CreateProcessAsUser API.

Syntax

```
BOOL ArgxCreateProcessAsUser(
        HANDLE                 hToken,
        LPCTSTR                lpApplicationName,
        LPCTSTR*               lpArgv,
        DWORD                  dwArgc,
        LPSECURITY_ATTRIBUTES lpProcessAttributes,
        LPSECURITY_ATTRIBUTES lpThreadAttributes,
        BOOL                   bInheritHandles,
        DWORD                  dwCreationFlags,
        LPVOID                 lpEnvironment,
        LPCTSTR                lpCurrentDirectory,
        LPSTARTUPINFO          lpStartupInfo,
        LPPROCESS_INFORMATION lpProcessInformation
     );
```

Parameters

hToken A primary token representing the user in whose security context the new process should start. This handle must have `TOKEN_QUERY`, `TOKEN_DUPLICATE` and `TOKEN_ASSIGN_PRIMARY` access rights (see Access Rights for Access-Token Objects). The user identified by the token must have read and execute access to the application that is being started.

You can obtain such a token by calling the LogonUser or DuplicateTokenEx APIs.

lpApplicationName The path to the executable to start. This string is not subject to any path searching, though it may be a relative path. There is no default extension for this parameter.

If this parameter is `NULL`, the executable path will be taken from the first argument, `lpArgv[0]`.

lpArgv The argument vector. If `lpApplicationName` is `NULL`, the first element of the argument vector will be used to locate the desired executable.

In that case:

- If `lpArgv[0]` contains path delimiters, it will be treated as a literal path and used directly; otherwise,
- If `lpArgv[0]` does not have an extension, the extension ".exe" will be appended automatically.
- The function will then try to find the executable by looking in the following places:
  1. The directory from which the calling application loaded.
  2. The current directory.
  3. The Windows System32 directory (as returned by the GetSystemDirectory API).
  4. The 16-bit Windows System directory, if present.
  5. The Windows directory (as returned by the GetWindowsDirectory API).
  6. The directories listed in the `PATH` environment variable.

This function will *not* modify any of the strings in `lpArgv`. Also note that this function does not suffer from the security hole in the CreateProcessAsUser API caused by that function's attempt to parse filenames and directory names containing spaces.

`lpArgv` *must not* be `NULL`.

dwArgc The number of elements in `lpArgv`, which must be at least one.

lpProcessAttributes Describes the desired SECURITY_ATTRIBUTES for the new process. May be `NULL`.

lpThreadAttributes Describes the desired SECURITY_ATTRIBUTES for the primary thread of the new process. May be `NULL`.

bInheritHandles If `TRUE`, inheritable handles will be inherited by the subprocess. Importantly, inherited handles have the same access rights that they have in the parent process, so this needs to be used with care.

dwCreationFlags Flags that control priority class and creation behaviour. See Process Creation Flags for more information.

lpEnvironment Points to the environment block for the new process, or `NULL` to inherit the environment of the parent.

lpCurrentDirectory If `NULL`, the new process will start with the same current directory as the parent process. Otherwise, must contain the full path to the desired current directory.

lpStartupInfo Points to a STARTUPINFO or STARTUPINFOEX structure. May be `NULL`.

lpProcessInformation Points to a PROCESS_INFORMATION structure that will be filled in with handles to the process and its primary thread. Note that these handles *must be closed* when no longer needed.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, it will return zero (i.e. `FALSE`), with extended error information supplied via GetLastError.

See also

CreateProcessAsUser

1.4.3 ArgxGetArguments function

Retrieves the calling process's command line arguments in the form of an argument vector. If the ArgX protocol is not in use, this function will parse the flat command line using the CommandLineToArgvW API and return the results.

Syntax

```
BOOL ArgxGetArguments(
        PDWORD     pdwArgc,
        LPCTSTR** plpArgv,
        BOOL*      pbUserArgX
    );
```

Parameters

pdwArgc A pointer to a `DWORD` that will be initialised with a count of the number of arguments in the argument vector. May not be `NULL`.

plpArgv Points to a variable that will receive the argument vector pointer. May not be `NULL`.

pbUserArgX Points to a `BOOL` variable that will be set to `TRUE` if the ArgX mechanism was used to obtain the arguments and `FALSE` otherwise. If not required, may be `NULL`.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, it will return zero, with extended error information supplied via GetLastError.

See also

CommandLineToArgvW

1.4.4 ArgxFindExecutable function

Given the first element of an argument vector, attempts to locate an executable according to the rules specified in the *ArgxCreateProcess* documentation.

Syntax

```
LPTSTR ArgxFindExecutable(LPCTSTR lpszArgv0);
```

Parameters

`lpszArgv0` The name of the executable to find. This is processed as follows:

- If `lpszArgv0` contains path delimiters, it will be treated as a literal path and used directly; otherwise,
- If `lpszArgv0` does not have an extension, the extension ".exe" will be appended automatically.
- The function will then try to find the executable by looking in the following places:
  1. The directory from which the calling application loaded.
  2. The current directory.
  3. The Windows System32 directory (as returned by the GetSystemDirectory API).
  4. The 16-bit Windows System directory, if present.
  5. The Windows directory (as returned by the GetWindowsDirectory API).
  6. The directories listed in the `PATH` environment variable.

Return value

If the function succeeds, it returns a string containing the path to the executable found according to the rules above. This string should be released when no longer required, using the LocalFree function.

If the function fails to find a match, it returns `NULL`.

See also

ArgxCreateProcess

1.4.5 ArgxIsSupportedByExecutable function

Tests if the specified executable supports ArgX, without actually starting it.

Syntax

```
BOOL ArgxIsSuportedByExecutable(LPCTSTR lpszExecutablePath);
```

Parameters

`lpszExecutablePath` The path to the executable to test. This should be a valid path to the executable file; no searching takes place, and no default extension is appended.

Return value

If the executable specified by `lpszExecutablePath` supports ArgX protocol, the return value is non-zero; otherwise, the return value is zero.

1.4.6 ARGX_SECTION_DATA structure

A structure corresponding to the ArgX section in the *ArgX Specification*.

Syntax

```
#define ARGX_MAGIC      ((DWORD)0x00005841)
#define ARGX_MAGIC_INIT ((DWORD)0x00007861)

typedef struct {
  DWORD   dwMagic;
  DWORD   dwArgc;
  LPWSTR *pszArgv;
} ARGX_SECTION_DATA;
```

Members

`dwMagic` In the executable image, should be set to `ARGX_MAGIC_INIT`; if ArgX protocol is in use, this will be updated to `ARGX_MAGIC`.

`dwArgc` A count of the arguments in the argument vector. In the executable image, should be set to zero.

`pszArgv` A pointer to the argument vector. In the executable image, should be set to zero.

Remarks

You most likely do not need to use this structure directly; instead, any call to *ArgxGetArguments* will automatically result in an appropriately initialised copy of this structure ending up in the "ArgX" section in your executable image. This is done by the code at the top of `src/ArgxGetArguments.cpp`:

```
#pragma section("ArgX", read, write)
namespace {
  __declspec(allocate("ArgX")) ARGX_SECTION_DATA argxData = { ARGX_MAGIC_INIT, 0, 0 };
}
```

See also

ArgxGetArguments

CHAPTER 2

Indices and tables

- genindex
- modindex
- search