PyWinCFFI Documentation

Release 0.5.0

Oliver Palmer

Nov 18, 2017

Contents

`pywincffi` is a wrapper around some Windows API functions using Python and the cffi library. This project was originally created to assist the Twisted project in moving away from its dependency on `pywin32`. Contributions to expand on the APIs which pywincffi offers are always welcome however.

The core objectives and design principles behind this project are:

- It should be easier to to use Windows API functions both in terms of implementation and distribution.
- Python 2.7 and 3.x should be supported from a single code base and not require a consumer of pywincffi to worry about how they use the library.
- Type conversion, error checking and other 'C like' code should be the responsibility of the library where possible.
- APIs provided by pywincffi should mirror their Windows counterparts as closely as possible so the MSDN documentation can be more easily used as reference.
- Documentation and error messages should be descriptive, consistent, complete and accessible. Examples should be provided for more complex use cases.
- For contributors, it should be possible to develop and test regardless of what platform the contributor is coming from.

See also:

PyWinCFFI's README

CHAPTER 1

Main Index

1.1 Changelog

This document contains information on pywincff's release history. Later versions are shown first.

1.1.1 Versions

0.5.0

Notable enhancements and changes are:

- **Python 2.6 support has been dropped**. Many projects have already moved on from Python 2.6 including Twisted which this project was initially intended to support. Additionally many libraries or tools that pywincffi no longer have direct support for Python 2.6 or simply break in later versions. This leads to having to maintain and support older libraries in ths build which is going to become increasing difficult. Pull requests to support Python 2.6 will be evaluated on a case-by-case basis.
- Various improvements to the tests and build including replacement of nosetests with pytest, transition from pep8 to pycodestyle and upgrading tools and libraries to more modern versions.
- #123 - Implemented `pywincffi.kernel32.event.SetEvent()`
- #124 - Implemented *pywincffi.kernel32.file.GetTempPath()*
- #129 - Implemented *pywincffi.kernel32.console.SetConsoleTextAttribute()*. Also implemented *pywincffi.kernel32.console.GetConsoleScreenBufferInfo()* and *pywincffi.kernel32.console.CreateConsoleScreenBuffer()* to properly test the new functionality.
- Changed ordering of arguments for the following functions as they did not match the underlying C function signatures:
  - #130 - *pywincffi.kernel32.events.CreateEvent()*
  - #131 - *pywincffi.kernel32.pipe.CreatePipe()*

  - #133 - *pywincffi.kernel32.process.CreateProcess()*

- #134 / #137 - Use str.format(), and unicode string literals in a few places.
- #138 - Improvements to *pywincffi.exceptions.WindowsAPIError* to better facilitate debugging.
- #140 - Added constant `ERROR_BAD_EXE_FORMAT`, a required constant by Twisted.

0.4.0

Notable enhancements and changes are:

- Addition of *pywincffi.kernel32.process.CreateProcess()*, *pywincffi.kernel32.overlapped.GetOverlappedResult()* and several structures. Implemented for #69.
- Reworked the test setup steps so they're more consistent.
- Added a cleanup step to the tests to track down cases that were not resetting or testing the Windows API error code.
- Cleaned up the setUp step in the base test case.
- Added error constant `ERROR_INVALID_HANDLE`.
- `pywincffi.kernel32.pid_exists()` will no longer result in the Windows API error code being set to a non-zero value after exiting the function.
- General code cleanup in a few of the core modules.
- Removed an installation dependency: enum34

0.3.1

Notable enhancements and changes are:

- #81 - `pywincffi.user32.synchronization.WSAEventSelect()` and `pywincffi.user32.synchronization.WSAEnumNetworkEvents()`
- Removal of the `pywincffi.core.config` module in #107. The module was mostly unused internally and was not being used as part of the public APIs either.
- Improvements to the *pywincffi.core.dist* module in #106. This change allows pywincffi to add constants, functions, etc to the loaded library when *pywincffi.core.dist.load()* is called. Before certain constants, such as `ERROR_INVALID_HANDLE`, had to be imported from other modules rather than used directly from the library object.

0.3.0

Notable enhancements and changes are:

- Added the `pywincffi.kernel32.CreateToolhelp32Snapshot()` function in #101.
- Fixed a bug where `pywincffi.checks.input_check()` might raise `ffi.error` in #73
- Several enhancements bringing #69 closer to closure.
- **Addition several functions for #69:**
  - #70 - *pywincffi.kernel32.events.CreateEvent()* and *pywincffi.kernel32.events.OpenEvent()*
  - #75 - *pywincffi.kernel32.events.ResetEvent()*

  - #76 - *pywincffi.kernel32.process.TerminateProcess()*
  - #78 - *pywincffi.kernel32.handle.DuplicateHandle()*
  - #79 - pywincffi.kernel32.process.ClearCommError()
  - #80 - *pywincffi.user32.synchronization.MsgWaitForMultipleObjects()*
- Added Python 3.5 support to the build. No bug fixes or code changes where required, just a minor test modification.
- All exposed APIs updated to use the new Windows equivalent Python types in *pywincffi.wintypes*.
- All exposed APIs now explicitly require either text or binary data.
- Added FOREGROUND_RED, FOREGROUND_GREEN and FOREGROUND_BLUE constants in #95.
- Improved documentation for *pywincffi.exceptions.InputError* and added the ability to generate custom error messages.

0.2.0

This release contains several enhancements, bug fixes and other changes. You can see all of the major issues by viewing the milestone on GitHub: https://github.com/opalmer/pywincffi/issues?q=milestone:0.2.0.

Notable enhancements and changes are:

- Improved error handling which brings more consistent error messages with better information.
- Several new Windows API function implementations including FlushFileBuffers, CreateFile, LockFileEx, UnlockFileEx, MoveFileEx, GetProcessId, and GetCurrentProcess.
- New wrapper function pid_exists().
- Refactored kernel32 module structure.
- Several bug fixes to existing tests and functions.
- Updated developer documentation to better cover code reviews, style, functions, etc.
- Fixed broken urls in *PyCharm Remote Interpreter* section of vagrant documentation for developers.
- Added *pywincffi.kernel32.handle.GetHandleInformation()* and *pywincffi.kernel32.handle.SetHandleInformation()* in #66 - Thanks exvito!

0.1.2

Contains a fix to ensure that the proper version of `cffi` is installed. See https://github.com/opalmer/pywincffi/pull/45 for more detailed information. This release also includes a fix to the internal release tool.

0.1.1

The first public release of pywincffi. The GitHub release contains the full list of issues, changes and pull requests. The primary purpose of this release was to end up with the tools and code necessary to begin integrating pywincffi into Twisted.

0.1.0

This was an internal test release. No data was published to PyPi or GitHub.

- genindex
- modindex
- search

CHAPTER 2

Python Package

2.1 pywincffi

2.1.1 pywincffi package

Subpackages

pywincffi.core package

Submodules

pywincffi.core.checks module

Checks

Provides functions that are responsible for internal type checks.

`pywincffi.core.checks.`**`error_check`**(*function*, *code=None*, *expected=None*)

Checks the results of a return code against an expected result. If a code is not provided we'll use `ffi.getwinerror()` to retrieve the code.

Parameters

- **`function`** (*`str`*) – The Windows API function being called.
- **`code`** (*`int`*) – An explicit code to compare against.
- **`expected`** (*`int`*) – The code we expect to have as a result of a successful call. This can also be passed `pywincffi.core.checks.NON_ZERO` if `code` can be anything but zero.

Raises ***`pywincffi.exceptions.WindowsAPIError`*** – Raised if we receive an unexpected result from a Windows API call

pywincffi.core.checks.**input_check**(*name*, *value*, *allowed_types=None*, *allowed_values=None*)

A small wrapper around `isinstance()`. This is mainly meant to be used inside of other functions to pre-validate input rater than using assertions. It's better to fail early with bad input so more reasonable error message can be provided instead of from somewhere deep in cffi or Windows.

Parameters

- **name** (*str*) – The name of the input being checked. This is provided so error messages make more sense and can be attributed to specific input arguments.
- **value** – The value we're performing the type check on.
- **allowed_types** – The allowed type or types for `value`.
- **allowed_values** (*tuple*) – A tuple of allowed values. When provided `value` must be in this tuple otherwise `InputError` will be raised.

Raises

- ***pywincffi.exceptions.InputError*** – Raised if `value` is not an instance of `allowed_types`
- **TypeError** – Raised if `allowed_values` is provided and not a tuple.

pywincffi.core.dist module

Distribution

Module responsible for building the pywincffi distribution in `setup.py`. This module is meant to serve two purposes. The first is to serve as the main means of loading the pywincffi library:

```
>>> from pywincffi.core import dist
>>> ffi, lib = dist.load()
```

The second is to facilitate a means of building a static library. This is used by the setup.py during the install process to build and install pywincffi as well as a wheel for distribution.

pywincffi.core.dist.**load**()

The main function used by pywincffi to load an instance of `FFI` and the underlying library.

pywincffi.core.logger module

Logger

This module contains pywincffi's logger and functions to retrieve new child loggers.

pywincffi.core.logger.**get_logger**(*name*)

Returns an instance of `logging.Logger` as a child of pywincffi's main logger.

Parameters **name** (*str*) – The name of the child logger to return. For example, if you provide *foo* for the name the resulting name will be *pywincffi.foo*.

Raises **ValueError** – Raised if `name` starts with a dot.

Return type `logging.Logger`

pywincffi.core.typesbase module

Types Base

Provides the base types on top of which user visible types will be built.

class pywincffi.core.typesbase.**CFFICDataWrapper**(*cdecl*, *ffi*)
Bases: object

Base class for exposing Python types and interfaces to pywincffi users:

- Wraps a CFFI cdata object in self._cdata.
- Delegates attribute getting/setting to self._cdata, supporting structs.
- Delegates item getting/setting to self._cdata, supporting arrays.

Attribute access is not delegated to the wrapped object if the class itself contains such an attribute and that attribute is a descriptor; this is in place to support @property in sub-classes.

Parameters

- **cdecl** (*str*) – C type specification as used in ff.new(cdecl)
- **ffi** (*cffi.api.FFI*) – FFI instance used to create wrapped cdata object.

Module contents

Core Sub-Package

An internal package used by pywincffi for loading the underlying _pywincffi module, handling configuration data, logging and other common tasks. This package also contains the C source and header files.

pywincffi.dev package

Submodules

pywincffi.dev.lint module

Lint Utilities

Provides some help to pylint so static analysis can be made aware of some constants and functions that we define in headers.

pywincffi.dev.lint.**constants_in_file**(*path*)
Returns a set of constants in the given file path

pywincffi.dev.lint.**functions_in_file**(*path*)
Returns a set of functions defined in the given file path

pywincffi.dev.lint.**register**(_)
An entrypoint that pylint uses to search for and register plugins with the given linter

pywincffi.dev.lint.**transform**(*cls*, *constants=None*, *functions=None*)
Transforms class objects from pylint so they're aware of extra attributes that are not present when being statically analyzed.

pywincffi.dev.release module

pywincffi.dev.testutil module

Test Utility

This module is used by the unittests.

class pywincffi.dev.testutil.**LibraryWrapper**(*library*, *attributes*)
Bases: object

Used by `TestCase.mock_library()` to replace specific attributes on a compiled library.

class pywincffi.dev.testutil.**SharedState**
Bases: object

Contains some state data which is shared across multiple *TestCase* instances. This is kept outside of the test case class itself so it can't be inadvertently modified by a test or fixture.

HAS_INTERNET = None

ffi = None

kernel32 = None

ws2_32 = None

class pywincffi.dev.testutil.**TestCase**(*methodName='runTest'*)
Bases: unittest.case.TestCase

A base class for all test cases. By default the core test case just provides some extra functionality.

GetLastError()
Returns a tuple containing output from the Windows GetLastError function and the associated error message. The error message will be None if GetLastError() returns 0.

HAS_INTERNET = None

INTERNET_HOSTS = ('github.com', 'readthedocs.org', 'example.com')

INTERNET_PORT = 80

REQUIRES_INTERNET = False

SetLastError(*errno*)
Wrapper for SetLastError()

WSAGetLastError()
Returns a tuple containing output from the Windows WSAGetLastError function and the associated error message. The error message will be None if WSAGetLastError() returns 0.

WSASetLastError(*errno*)
Wrapper for WSASetLastError()

assert_last_error(*errno*)
This function will assert that the last unhandled error was `errno`. After the check the last error will be reset to zero.

Parameters **errno** (*int*) – The expected value from GetLastError.

create_python_process(*command*)
Creates a Python process that run `command`

ffi = None

kernel32 = None

maybe_assert_last_error(*errno*)
This function is similar to *assert_last_error()* except it won't fail if the current error number is already 0.

random_string(*length*)
Returns a random string as long as `length`. The first character will always be a letter. All other characters will be A-F, A-F or 0-9.

setUp()

classmethod setUpClass()

unhandled_error_check()
A cleanup step which ensures that there are not any uncaught API errors left over. Unhandled errors could be a sign of an unhandled testing artifact, improper API usage or other problem. In any case, unhandled errors are often a source of test flake.

ws2_32 = None

pywincffi.dev.testutil.**mock_library**(***attributes*)
Used to replace an attribute the library that `dist.load()` returns. Useful for replacing part of the compiled library as part of the test.

Module contents

Development Sub-Package

This package is used for development, testing and release purposes. It does not contain core functionality of pywincffi and is unused by *pywincffi.core*, *pywincffi.kernel32* and other similar modules.

pywincffi.kernel32 package

Submodules

pywincffi.kernel32.comms module

Communications

A module containing Windows functions related to communications.

pywincffi.kernel32.comms.**ClearCommError**(*hFile*)
Retrieves information about a communications error and reports the current status of a communications device.

See also:

https://msdn.microsoft.com/en-us/aa363180

Parameters **hFile** (*pywincffi.wintypes.HANDLE*) – A handle to the communications device, typically created by `CreateFile()`

Return type tuple

Returns

Returns a two element tuple containing the `lpErrors` and `lpStat` result objects.

- `lpErrors` - Contains the mast indicating the type of error
- **`lpStat` - A `COMSTAT` structure which contains the device's** information.

pywincffi.kernel32.console module

Console

A module containing functions for interacting with a Windows console.

pywincffi.kernel32.console.**CreateConsoleScreenBuffer**(*dwDesiredAccess*, *dwShareMode*, *lpSecurityAttributes=None*, *dwFlags=None*)

Creates a console screen buffer.

See also:

https://docs.microsoft.com/en-us/windows/console/createconsolescreenbuffer

Parameters

- **`dwDesiredAccess`** (*`int or None`*) – The access to the console screen buffer. If *None* is provided then the Windows APIs will use a default security descriptor.
- **`dwShareMode`** (*`int or None`*) – Controls the options for sharing the resulting handle. If *None* or 0 then the resulting buffer cannot be shared.
- **`lpSecurityAttributes`** (*`pywincffi.wintypes.SECURITY_ATTRIBUTES`*) – Extra security attributes that determine if the resulting handle can be inherited. If *None* is provided, which is the default, then the handle cannot be inherited.
- **`dwFlags`** (*`int`*) – The type of console buffer to create. The flag is superficial because it only accepts None or `CONSOLE_TEXTMODE_BUFFER` as inputs. If no value is provided, which is the default, then `CONSOLE_TEXTMODE_BUFFER` is automatically used.

Return type `pywincffi.wintypes.HANDLE`\`

Returns Returns the handle created by the underlying C function. `pywincffi.kernel32.CloseHandle()` should be called on the handle when you are done with it.

pywincffi.kernel32.console.**GetConsoleScreenBufferInfo**(*hConsoleOutput*)

Retrieves information about the specified console screen buffer.

See also:

https://docs.microsoft.com/en-us/windows/console/getconsolescreenbufferinfo

Parameters **`hConsoleOutput`** (*`pywincffi.wintypes.HANDLE`*) – A handle to the console screen buffer. The handle must have the `GENERIC_READ` access right.

Returns Returns a ffi data structure with attributes corresponding to the fields on the `PCONSOLE_SCREEN_BUFFER_INFO` struct.

pywincffi.kernel32.console.**SetConsoleTextAttribute**(*hConsoleOutput*, *wAttributes*)

Sets the attributes of characters written to a console buffer.

See also:

https://docs.microsoft.com/en-us/windows/console/setconsoletextattribute

Parameters

- **hConsoleOutput** (*pywincffi.wintypes.HANDLE*) – A handle to the console screen buffer. The handle must have the GENERIC_READ access right.
- **wAttributes** (*int*) – The character attribute(s) to set.

pywincffi.kernel32.events module

Events

A module containing Windows functions for working with events.

pywincffi.kernel32.events.**CreateEvent**(*lpEventAttributes=None*, *bManualReset=True*, *bInitialState=False*, *lpName=None*)

Creates or opens an named or unnamed event object.

See also:

https://msdn.microsoft.com/en-us/library/ms682396

:keyword pywincffi.wintypes.SECURITY_ATTRIBUTES lpEventAttributes: If not provided then, by default, the handle cannot be inherited by a subprocess.

Parameters

- **bManualReset** (*bool*) – If True then this function will create a manual reset event which must be manually reset with *ResetEvent()*. Refer to the msdn documentation for full information.

  Default: True
- **bInitialState** (*bool*) – If True the initial state will be 'signaled'.

  Default: False
- **lpName** (*str*) – Type is unicode on Python 2, str on Python 3. The optional case-sensitive name of the event. If not provided then the event will be created without an explicit name.

Returns Returns a pywincffi.wintypes.HANDLE to the event. If an event by the given name already exists then it will be returned instead of creating a new event.

pywincffi.kernel32.events.**OpenEvent**(*dwDesiredAccess*, *bInheritHandle*, *lpName*)

Opens an existing named event.

See also:

https://msdn.microsoft.com/en-us/library/ms684305

Parameters

- **dwDesiredAccess** (*int*) – The access desired for the event object.
- **bInheritHandle** (*bool*) –
- **lpName** (*str*) – Type is unicode on Python 2, str on Python 3.

Returns Returns a pywincffi.wintypes.HANDLE to the event.

pywincffi.kernel32.events.**ResetEvent**(*hEvent*)
Sets the specified event object to the nonsignaled state.

See also:

https://msdn.microsoft.com/en-us/library/ms684305

Parameters **hEvent** (*pywincffi.wintypes.HANDLE*) – A handle to the event object to be reset. The handle must have the EVENT_MODIFY_STATE access right.

pywincffi.kernel32.events.**SetEvent**(*hEvent*)
Sets the specified event object to the signaled state.

See also:

https://msdn.microsoft.com/en-us/library/ms686211

Parameters **hEvent** (*pywincffi.wintypes.HANDLE*) – A handle to the event object. The handle must have the EVENT_MODIFY_STATE access right.

pywincffi.kernel32.file module

Files

A module containing common Windows file functions for working with files.

pywincffi.kernel32.file.**CreateFile**(*lpFileName*, *dwDesiredAccess*, *dwShareMode=None*, *lpSecurityAttributes=None*, *dwCreationDisposition=None*, *dwFlagsAndAttributes=None*, *hTemplateFile=None*)
Creates or opens a file or other I/O device. Default values are provided for some of the default arguments for CreateFile() so its behavior is close to Pythons `open()` function.

See also:

https://msdn.microsoft.com/en-us/library/aa363858 https://msdn.microsoft.com/en-us/library/gg258116

Parameters

- **lpFileName** (*str*) – Type is `unicode` on Python 2, `str` on Python 3. The path to the file or device being created or opened.
- **dwDesiredAccess** (*int*) – The requested access to the file or device. Microsoft's documentation has extensive notes on this parameter in the seealso links above.
- **dwShareMode** (*int*) – Access and sharing rights to the handle being created. If not provided with an explicit value, FILE_SHARE_READ will be used which will other open operations or process to continue to read from the file.
- **lpSecurityAttributes** (*pywincffi.wintypes.SECURITY_ATTRIBUTES*) – See Microsoft's documentation for more detailed information.
- **dwCreationDisposition** (*int*) – Action to take when the file or device does not exist. If not provided with an explicit value, CREATE_ALWAYS will be used which means existing files will be overwritten.
- **dwFlagsAndAttributes** (*int*) – The file or device attributes and flags. If not provided an explict value, FILE_ATTRIBUTE_NORMAL will be used giving the handle essentially no special attributes.

- **hTemplateFile** (*pywincffi.wintypes.HANDLE*) – A handle to a template file with the GENERIC_READ access right. See Microsoft's documentation for more information. If not provided an explicit value, NULL will be used instead.

Returns The file pywincffi.wintypes.HANDLE created by CreateFile.

pywincffi.kernel32.file.**FlushFileBuffers**(*hFile*)
Flushes the buffer of the specified file to disk.

See also:

https://msdn.microsoft.com/en-us/library/aa364439

Parameters **hFile** (*pywincffi.wintypes.HANDLE*) – The handle to flush to disk.

pywincffi.kernel32.file.**GetTempPath**()
Retrieves the path of the directory designated for temporary files.

See also:

https://msdn.microsoft.com/en-us/aa364992

Returns Returns a string containing the value produced by the underlying C function.

pywincffi.kernel32.file.**LockFileEx**(*hFile*, *dwFlags*, *nNumberOfBytesToLockLow*, *nNumberOfBytesToLockHigh*, *lpOverlapped=None*)
Locks hFile for exclusive access by the calling process.

See also:

https://msdn.microsoft.com/en-us/library/aa365203

Parameters

- **hFile** (*pywincffi.wintypes.HANDLE*) – The handle to the file to lock. This handle must have been created with either the GENERIC_READ or GENERIC_WRITE right.
- **dwFlags** (*int*) – One or more of the following flags:
  - LOCKFILE_EXCLUSIVE_LOCK - Request an exclusive lock.
  - LOCKFILE_FAIL_IMMEDIATELY - Return immediately if the lock could not be acquired. Otherwise *LockFileEx()* will wait.
- **nNumberOfBytesToLockLow** (*int*) – The start of the byte range to lock.
- **nNumberOfBytesToLockHigh** (*int*) – The end of the byte range to lock.
- **lpOverlapped** (*pywincffi.wintypes.OVERLAPPED*) – The underlying Windows API requires lpOverlapped, which acts both an input argument and may contain results after calling. If None is provided, a throw-away zero-filled instance will be created to support such call. See Microsoft's documentation for intended usage.

pywincffi.kernel32.file.**MoveFileEx**(*lpExistingFileName*, *lpNewFileName*, *dwFlags=None*)
Moves an existing file or directory, including its children, see the MSDN documentation for full options.

See also:

https://msdn.microsoft.com/en-us/library/aa365240

Parameters

- **lpExistingFileName** (*str*) – Type is `unicode` on Python 2, `str` on Python 3. Name of the file or directory to perform the operation on.
- **lpNewFileName** (*str*) – Type is `unicode` on Python 2, `str` on Python 3. Optional new name of the path or directory. This value may be `None`.
- **dwFlags** (*int*) – Parameters which control the operation of *MoveFileEx()*. See the MSDN documentation for full details. By default `MOVEFILE_REPLACE_EXISTING | MOVEFILE_WRITE_THROUGH` is used.

pywincffi.kernel32.file.**ReadFile**(*hFile*, *nNumberOfBytesToRead*, *lpOverlapped=None*)

Read the specified number of bytes from `hFile`.

See also:

https://msdn.microsoft.com/en-us/library/aa365467

Parameters

- **hFile** (*pywincffi.wintypes.HANDLE*) – The handle to read from.
- **nNumberOfBytesToRead** (*int*) – The number of bytes to read from `hFile`
- **lpOverlapped** (*pywincffi.wintypes.OVERLAPPED*) – See Microsoft's documentation for intended usage and below for an example.

```
>>> from pywincffi.core import dist
>>> from pywincffi.kernel32 import ReadFile, CreateEvent
>>> from pywincffi.wintypes import OVERLAPPED
>>> hEvent = CreateEvent(...)
>>> lpOverlapped = OVERLAPPED()
>>> lpOverlapped.hEvent = hEvent
>>> read_data = ReadFile(  # read 12 bytes from hFile
...     hFile, 12, lpOverlapped=lpOverlapped)
```

Returns Returns the binary data read from `hFile` Type is `str` on Python 2, `bytes` on Python 3.

pywincffi.kernel32.file.**UnlockFileEx**(*hFile*, *nNumberOfBytesToUnlockLow*, *nNumberOfBytesToUnlockHigh*, *lpOverlapped=None*)

Unlocks a region in the specified file.

See also:

https://msdn.microsoft.com/en-us/library/aa365716

Parameters

- **hFile** (*pywincffi.wintypes.HANDLE*) – The handle to the file to unlock. This handle must have been created with either the `GENERIC_READ` or `GENERIC_WRITE` right.
- **nNumberOfBytesToUnlockLow** (*int*) – The start of the byte range to unlock.
- **nNumberOfBytesToUnlockHigh** (*int*) – The end of the byte range to unlock.
- **lpOverlapped** (*pywincffi.wintypes.OVERLAPPED*) – The underlying Windows API requires lpOverlapped, which acts both an input argument and may contain results after calling. If None is provided, a throw-away zero-filled instance will be created to support such call. See Microsoft's documentation for intended usage.

pywincffi.kernel32.file.**WriteFile**(*hFile*, *lpBuffer*, *nNumberOfBytesToWrite=None*, *lpOverlapped=None*)

Writes data to `hFile` which may be an I/O device for file.

See also:

https://msdn.microsoft.com/en-us/library/aa365747

Parameters

- **hFile** (*pywincffi.wintypes.HANDLE*) – The handle to write to.
- **lpBuffer** (*str/bytes*) – Type is `str` on Python 2, `bytes` on Python 3. The data to be written to the file or device.
- **nNumberOfBytesToWrite** (*int*) – The number of bytes to be written. Defaults to len(lpBuffer).
- **lpOverlapped** (*pywincffi.wintypes.OVERLAPPED*) – See Microsoft's documentation for intended usage and below for an example.

```
>>> from pywincffi.core import dist
>>> from pywincffi.kernel32 import WriteFile, CreateEvent
>>> from pywincffi.wintypes import OVERLAPPED
>>> hEvent = CreateEvent(...)
>>> lpOverlapped = OVERLAPPED()
>>> lpOverlapped.hEvent = hEvent
>>> bytes_written = WriteFile(
...     hFile, "Hello world", lpOverlapped=lpOverlapped)
```

Returns Returns the number of bytes written.

pywincffi.kernel32.handle module

Handles

A module containing general functions for working with handle objects. The functions provided here are part of the `kernel32` library.

`pywincffi.kernel32.handle.`**`CloseHandle`**(*hObject*)

Closes an open object handle.

See also:

https://msdn.microsoft.com/en-us/library/ms724211

Parameters **hObject** (*pywincffi.wintypes.HANDLE or pywincffi.wintypes.SOCKET*) – The handle object to close.

`pywincffi.kernel32.handle.`**`DuplicateHandle`**(*hSourceProcessHandle*, *hSourceHandle*, *hTargetProcessHandle*, *dwDesiredAccess*, *bInheritHandle*, *dwOptions*)

Duplicates an object handle.

See also:

https://msdn.microsoft.com/en-us/ms724251

Parameters

- **hSourceProcessHandle** (*pywincffi.wintypes.HANDLE*) – A handle to the process which owns the handle to be duplicated.
- **hSourceHandle** (*pywincffi.wintypes.HANDLE*) – The handle to be duplicated.

- **hTargetProcessHandle** (*pywincffi.wintypes.HANDLE*) – A handle to the process which should receive the duplicated handle.
- **dwDesiredAccess** (*int*) – The access requested for the new handle.
- **bInheritHandle** (*bool*) – True if the handle should be inheritable by new processes.
- **dwOptions** (*int*) – Options which control how the handle is duplicated. Valid values are any of the below (or a combination of):
  - **DUPLICATE_CLOSE_SOURCE - Closes the source handle, even** if there's an error.
  - **DUPLICATE_SAME_ACCESS - Ignores the dwDesiredAccess** parameter duplicates with the same access as the original handle.

Return type pywincffi.wintypes.HANDLE

Returns Returns the duplicated handle.

pywincffi.kernel32.handle.**GetHandleInformation**(*hObject*)

Returns properties of an object handle.

See also:

https://msdn.microsoft.com/en-us/library/ms724329

Parameters **hObject** (*pywincffi.wintypes.HANDLE*) – A handle to an object whose information is to be retrieved.

Return type int

Returns Returns the set of bit flags that specify properties of `hObject`.

pywincffi.kernel32.handle.**GetStdHandle**(*nStdHandle*)

Retrieves a handle to the specified standard device (standard input, standard output, or standard error).

See also:

https://msdn.microsoft.com/en-us/library/ms683231

Parameters **nStdHandle** (*int*) – The standard device to retrieve.

Return type pywincffi.wintypes.HANDLE

Returns Returns a handle to the standard device retrieved.

pywincffi.kernel32.handle.**SetHandleInformation**(*hObject*, *dwMask*, *dwFlags*)

Sets properties of an object handle.

See also:

https://msdn.microsoft.com/en-us/ms724935

Parameters

- **hObject** (*pywincffi.wintypes.HANDLE*) – A handle to an object whose information is to be set.
- **dwMask** (*int*) – A mask that specifies the bit flags to be changed.
- **dwFlags** (*int*) – Set of bit flags that specifies properties of `hObject`.

pywincffi.kernel32.overlapped module

Overlapped

A module containing Windows functions for working with OVERLAPPED objects.

pywincffi.kernel32.overlapped.**GetOverlappedResult**(*hFile*, *lpOverlapped*, *bWait*)

Retrieves the results of an overlapped operation on the specified file, named pipe, or communications device. To specify a timeout interval or wait on an alertable thread, use GetOverlappedResultEx.

See also:

https://msdn.microsoft.com/en-us/library/ms683209

Parameters

- **hFile** (*pywincffi.wintypes.HANDLE*) – A handle to the file, named pipe, or communications device. This is the same handle that was specified when the overlapped operation was started by a call to the ReadFile, WriteFile, ConnectNamedPipe, TransactNamedPipe, DeviceIoControl, or WaitCommEvent function.
- **lpOverlapped** (*pywincffi.wintypes.OVERLAPPED*) – The an OVERLAPPED object that was specified when the overlapped operation was started
- **bWait** (*bool*) – If this parameter is TRUE, and the Internal member of the lpOverlapped structure is STATUS_PENDING, the function does not return until the operation has been completed. If this parameter is FALSE and the operation is still pending, the function returns FALSE and the GetLastError function returns ERROR_IO_INCOMPLETE

Returns The number of bytes that were actually transferred by a read or write operation. For a TransactNamedPipe operation, this is the number of bytes that were read from the pipe. For a DeviceIoControl operation, this is the number of bytes of output data returned by the device driver. For a ConnectNamedPipe or WaitCommEvent operation, this value is undefined.

pywincffi.kernel32.pipe module

Pipe

A module for working with pipe objects in Windows.

pywincffi.kernel32.pipe.**CreatePipe**(*lpPipeAttributes=None*, *nSize=0*)

Creates an anonymous pipe and returns the read and write handles.

See also:

https://msdn.microsoft.com/en-us/library/aa365152 https://msdn.microsoft.com/en-us/library/aa379560

```
>>> from pywincffi.core import dist
>>> from pywincffi.kernel32 import CreatePipe
>>> from pywincffi.wintypes import SECURITY_ATTRIBUTES
>>> lpPipeAttributes = SECURITY_ATTRIBUTES()
>>> lpPipeAttributes.bInheritHandle = True
>>> reader, writer = CreatePipe(lpPipeAttributes=lpPipeAttributes)
```

Parameters

- **lpPipeAttributes** (*pywincffi.wintypes.SECURITY_ATTRIBUTES*) – The security attributes to apply to the handle. By default NULL will be passed in, meaning the handle we create cannot be inherited. For more detailed information see the links below.
- **nSize** (*int*) – The size of the buffer in bytes. Passing in 0, which is the default will cause the system to use the default buffer size.

Returns Returns a tuple of pywincffi.wintype.HANDLE containing the reader and writer ends of the pipe that was created. The user of this function is responsible for calling CloseHandle at some point.

pywincffi.kernel32.pipe.**PeekNamedPipe**(*hNamedPipe*, *nBufferSize*)

Copies data from a pipe into a buffer without removing it from the pipe.

See also:

https://msdn.microsoft.com/en-us/library/aa365779

Parameters

- **hNamedPipe** (*pywincffi.wintypes.HANDLE*) – The handele to the pipe object we want to peek into.
- **nBufferSize** (*int*) – The number of bytes to 'peek' into the pipe.

Return type *PeekNamedPipeResult*

Returns Returns an instance of *PeekNamedPipeResult* which contains the buffer read, number of bytes read and the result.

class pywincffi.kernel32.pipe.**PeekNamedPipeResult**(*lpBuffer*, *lpBytesRead*, *lpTotalBytesAvail*, *lpBytesLeftThisMessage*)

Bases: tuple

lpBuffer
Alias for field number 0

lpBytesLeftThisMessage
Alias for field number 3

lpBytesRead
Alias for field number 1

lpTotalBytesAvail
Alias for field number 2

pywincffi.kernel32.pipe.**SetNamedPipeHandleState**(*hNamedPipe*, *lpMode=None*, *lpMaxCollectionCount=None*, *lpCollectDataTimeout=None*)

Sets the read and blocking mode of the specified hNamedPipe.

See also:

https://msdn.microsoft.com/en-us/library/aa365787

Parameters

- **hNamedPipe** (*pywincffi.wintypes.HANDLE*) – A handle to the named pipe instance.
- **lpMode** (*int*) – The new pipe mode which is a combination of read mode:
  - PIPE_READMODE_BYTE

– PIPE_READMODE_MESSAGE

And a wait-mode flag:

– PIPE_WAIT

– PIPE_NOWAIT

- **lpMaxCollectionCount** (*int*) – The maximum number of bytes collected.
- **lpCollectDataTimeout** (*int*) – The maximum time, in milliseconds, that can pass before a remote named pipe transfers information

pywincffi.kernel32.process module

Process

Provides functions, constants and utilities that wrap the Windows functions associated with process management and interaction. This module also provides several constants as well, see Microsoft's documentation for the constant names and their purpose:

- **Process Security and Access Rights** - https://msdn.microsoft.com/en-us/library/windows/desktop/ms684880

Note: Not all constants may be defined

pywincffi.kernel32.process.**CreateProcess**(*lpApplicationName=None*, *lpCommandLine=None*, *lpProcessAttributes=None*, *lpThreadAttributes=None*, *bInheritHandles=True*, *dwCreationFlags=None*, *lpEnvironment=None*, *lpCurrentDirectory=None*, *lpStartupInfo=None*)

Creates a new process and its primary thread. The process will be created in the same security context as the original process.

See also:

https://msdn.microsoft.com/en-us/library/ms682425

Parameters

- **lpStartupInfo** (*pywincffi.wintypes.STARTUPINFO*) – See Microsoft's documentation for additional information.

  Warning: The STARTUPINFOEX structure is not currently supported for this input.

- **lpCommandLine** (*str*) – The command line to be executed. The maximum length of this parameter is 32768. If no value is provided for lpApplicationName then the module name portion of lpCommandLine cannot exceed MAX_PATH.
- **lpApplicationName** (*str*) – The name of the module or application to be executed. This can be either the fully qualified path name or a partial name. The system path will not be searched. If no value is provided for this keyword then the input to lpCommandLine will be used by Windows instead.
- **lpProcessAttributes** (*pywincffi.wintypes.SECUREITY_ATTRIBUTES*) – Determines whether the returned handle to the new process object can be inherited by child processes. By default, the handle cannot be inherited.

- **lpThreadAttributes** (*pywincffi.wintypes.SECUREITY_ATTRIBUTES*) – Determines if the returned handle to the new thread object can be inherited by child processes. By default, the thread cannot be inherited.
- **bInheritHandles** (*bool*) – If True (the default) the handles inherited by the calling process are inherited by the new process.
- **dwCreationFlags** (*int*) – Controls the priority class and creation of the process. By default the process will flag will default to `NORMAL_PRIORITY_CLASS | CREATE_UNICODE_ENVIRONMENT`
- **lpEnvironment** (*dict*) – The environment for the new process. By default the the process will be created with the same environment as the parent process.

Note: All keys and values in the environment must be either unicode (Python 2) or strings (Python 3).

Note: This keyword will completely override the current if you wish to update the current environment instead then you will need to make and update a copy.

Warning: Excluding certain system environment variables such as `PATH` or `SYSTEMROOT` may result in crashes or unexpected behaviors depending on the program being run.

- **lpCurrentDirectory** (*str*) –

  The full path to the current directory for the process. If not provided then the process will have the same working directory as the parent process.

Raises ***InputError*** – Raised if `lpCommandLine` is too long or there are other input problems.

Return type *pywincffi.kernel32.process.CreateProcessResult*

Returns Returns a named tuple containing `lpCommandLine` and `lpProcessInformation`. The `lpProcessInformation` will be an instance of *pywincffi.wintypes.structures.PROCESS_INFORMATION*

class pywincffi.kernel32.process.**CreateProcessResult**(*lpCommandLine*, *lpProcessInformation*)

Bases: `tuple`

lpCommandLine
Alias for field number 0

lpProcessInformation
Alias for field number 1

pywincffi.kernel32.process.**CreateToolhelp32Snapshot**(*dwFlags*, *th32ProcessID*)

Takes a snapshot of the specified processes, as well as the heaps, modules, and threads used by these processes.

See also:

https://msdn.microsoft.com/en-us/ms682489

Parameters

- **dwFlags** (*int*) – The portions of the system to be included in the snapshot.
- **th32ProcessID** (*int*) – The process identifier of the process to be included in the snapshot.

Return type pywincffi.wintypes.HANDLE

Returns If the function succeeds, it returns an open handle to the specified snapshot.

pywincffi.kernel32.process.**GetCurrentProcess**()
Returns a handle to the current thread.

See also:

https://msdn.microsoft.com/en-us/library/ms683179

Note: Calling *pywincffi.kernel32.handle.CloseHandle()* on the handle produced by this function will produce an exception.

Returns The pywincffi.wintypes.HANDLE to the current process.

pywincffi.kernel32.process.**GetExitCodeProcess**(*hProcess*)
Retrieves the exit code of the given process handle. To retrieve a process handle use *OpenProcess()*.

Warning: You may want to use process_exit_code() instead of this function if you're just checking to see if a process has exited at all.

See also:

https://msdn.microsoft.com/en-us/library/ms683189

Parameters **hProcess** (*pywincffi.wintypes.HANDLE*) – The handle of the process to retrieve the exit code for.

Returns Returns the exit code of the requested process if one can be found.

pywincffi.kernel32.process.**GetProcessId**(*Process*)
Returns the pid of the process handle provided in Process.

See also:

https://msdn.microsoft.com/en-us/library/ms683215

Parameters **Process** (*pywincffi.wintypes.HANDLE*) – The handle of the process.

Returns Returns an integer which represents the pid of the given process handle.

pywincffi.kernel32.process.**OpenProcess**(*dwDesiredAccess*, *bInheritHandle*, *dwProcessId*)
Opens an existing local process object.

See also:

https://msdn.microsoft.com/en-us/library/ms684320

Parameters

- **dwDesiredAccess** (*int*) – The required access to the process object.

- **bInheritHandle** (*bool*) – Enables or disable handle inheritance for child processes.
- **dwProcessId** (*int*) – The id of the local process to be opened.

Returns Returns a `pywincffi.wintypes.HANDLE` to the opened process. This value can be used by other functions such as *`TerminateProcess()`*.

`pywincffi.kernel32.process.`**`TerminateProcess`**(*hProcess*, *uExitCode*)

Terminates the specified process and all of its threads.

See also:

https://msdn.microsoft.com/en-us/library/ms686714

Parameters

- **hProcess** (*pywincffi.wintypes.HANDLE*) – A handle to the process to be terminated.
- **uExitCode** (*int*) – The exit code of the processes and threads as a result of calling this function.

`pywincffi.kernel32.process.`**`module_name`**(*path*)

Returns the module name for the given `path`

```
>>> module_name(u"C:\Python27\python.exe -c 'print True'")
'C:\Python27\python.exe'
>>> module_name(u"C:\Program Files (x86)\Foo\program.exe -h")
'C:\Program'
>>> module_name(u"'C:\Program Files (x86)\Foo\program.exe' -h")
'C:\Program Files (x86)\Foo\program.exe'
```

This function is used internally by *`CreateProcess()`* to assist in validating input to the `lpCommandLine` argument. When calling *`CreateProcess()`* if `lpApplicationName` is not set then `lpCommandLine`'s module name cannot exceed `MAX_PATH`.

Raises **TypeError** – Raised if `path` is not a text type.

`pywincffi.kernel32.process.`**`pid_exists`**(*pid*, *wait=0*)

Returns True if there's a process associated with `pid`.

Parameters

- **pid** (*int*) – The id of the process to check for.
- **wait** (*int*) – An optional keyword that controls how long we tell `WaitForSingleObject()` to wait on the process.

Raises **ValidationError** – Raised if there's a problem with the value provided for `pid`.

pywincffi.kernel32.synchronization module

Synchronization

This module contains general functions for synchronizing objects and events. The functions provided in this module are parts of the `kernel32` library.

See also:

`pywincffi.user32.synchronization`

pywincffi.kernel32.synchronization.**WaitForSingleObject**(*hHandle*, *dwMilliseconds*)

Waits for the specified object to be in a signaled state or for `dwMiliseconds` to elapse.

See also:

https://msdn.microsoft.com/en-us/library/ms687032

Parameters

- **hHandle** (*pywincffi.wintypes.HANDLE*) – The handle to wait on.
- **dwMilliseconds** (*int*) – The time-out interval.

Module contents

Kernel32 Sub-Package

Provides functions, constants and utilities that wrap functions provided by `kernel32.dll`.

pywincffi.user32 package

Submodules

pywincffi.user32.synchronization module

Synchronization

This module contains general functions for synchronizing objects and events. The functions provided in this module are parts of the `user32` library.

See also:

pywincffi.kernel32.synchronization

pywincffi.user32.synchronization.**MsgWaitForMultipleObjects**(*pHandles*, *bWaitAll*, *dwMilliseconds*, *dwWakeMask*, *nCount=None*)

Waits until one or all of the specified objects are in a singled state or the timeout elapses.

See also:

https://msdn.microsoft.com/en-us/library/ms684242

Parameters

- **pHandles** (*list*) – A list or tuple of `pywincffi.wintypes.HANDLE` to wait on. See Microsoft's documentation for more information about the contents of this argument.
- **bWaitAll** (*bool*) – If True then this function will return when the states of all objects in `pHandles` are signaled.
- **dwMilliseconds** (*int*) – The timeout interval in milliseconds.
- **dwWakeMask** (*int*) – The input types for which an input event object handle will be added to the array of handles. See Microsoft's documentation for more detailed information.

- **nCount** (*int*) – The number of object handles in `pHandles`. By default this will be determined by checking the length of the input to `pHandles`.

Raises ***WindowsAPIError*** – Raised if the underlying Windows function returns `WAIT_FAILED`.

Return type int

Returns Returns the value of the event which caused the function to return. See Microsoft's documentation for full details on what this could be.

Module contents

User32 Sub-Package

Provides functions, constants and utilities that wrap functions provided by `user32.dll`.

pywincffi.wintypes package

Submodules

pywincffi.wintypes.functions module

Functions

This module provides various functions for converting and using Windows types.

`pywincffi.wintypes.functions.`**handle_from_file**(*file_*)
Converts a standard Python file object into a `HANDLE` object.

> **Warning:** This function is mainly intended for internal use. Passing in a file object with an invalid file descriptor may crash your interpreter.

Parameters **file** (`file`) – The Python file object to convert to a `HANDLE` object.

Raises ***InputError*** – Raised if `file_` does not appear to be a file object or is currently closed.

Return type `HANDLE`

`pywincffi.wintypes.functions.`**socket_from_object**(*sock*)
Converts a Python socket to a Windows SOCKET object.

> **Warning:** This function is mainly intended for internal use. Passing in an invalid object may result in a crash.

Parameters **sock** (*socket._socketobject*) – The Python socket to convert to `pywincffi.wintypes.SOCKET` object.

Return type `pywincffi.wintypes.SOCKET`

pywincffi.wintypes.functions.**wintype_to_cdata**(*wintype*)

Returns the underlying CFFI cdata object or ffi.NULL if wintype is None. Used internally in API wrappers to "convert" pywincffi's Python types to the required CFFI cdata objects when calling CFFI functions. Example:

```
>>> from pywincffi.core import dist
>>> from pywincffi.kernel32 import CreateEvent
>>> from pywincffi.wintypes import wintype_to_cdata
>>> ffi, lib = dist.load()
>>> # Get an event HANDLE, using the wrapper: it's a Python HANDLE object.
>>> hEvent = CreateEvent(bManualReset=False, bInitialState=False)
>>> # Call ResetEvent directly without going through the wrapper:
>>> hEvent_cdata = wintype_to_cdata(hEvent)
>>> result = lib.ResetEvent(hEvent_cdata)
```

Parameters **wintype** – A type derived from *pywincffi.core.typesbase.CFFICDataWrapper*

Returns The underlying CFFI <cdata> object, or ffi.NULL if wintype is None.

pywincffi.wintypes.objects module

Objects

Provides wrappers around core Windows objects such as file handles, sockets, etc.

class pywincffi.wintypes.objects.**HANDLE**(*data=None*)

Bases: *pywincffi.wintypes.objects.WrappedObject*

See also:

https://msdn.microsoft.com/en-us/library/aa383751

C_TYPE = 'HANDLE[1]'

class pywincffi.wintypes.objects.**SOCKET**(*data=None*)

Bases: *pywincffi.wintypes.objects.WrappedObject*

Handles interaction with a SOCKET object via its cdata

C_TYPE = 'SOCKET[1]'

class pywincffi.wintypes.objects.**WSAEVENT**(*data=None*)

Bases: *pywincffi.wintypes.objects.HANDLE*

Handles interaction with a WSAEVENT object via its cdata.

Note: This is functionally equivalent to a *HANDLE* object.

C_TYPE = 'WSAEVENT[1]'

class pywincffi.wintypes.objects.**WrappedObject**(*data=None*)

Bases: *pywincffi.core.typesbase.CFFICDataWrapper*

A wrapper used by other objects in this module to share common methods and conversion.

C_TYPE = None

pywincffi.wintypes.structures module

Structures

This module provides wrappers for structures produced or required by the Windows APIs.

class pywincffi.wintypes.structures.**FILETIME**
Bases: *pywincffi.core.typesbase.CFFICDataWrapper*

See also:

https://msdn.microsoft.com/en-us/library/ms724284

class pywincffi.wintypes.structures.**LPWSANETWORKEVENTS**
Bases: *pywincffi.core.typesbase.CFFICDataWrapper*

See also:

https://msdn.microsoft.com/en-us/ms741653

iErrorCode
An array of integers containing any associated error codes

class pywincffi.wintypes.structures.**OVERLAPPED**
Bases: *pywincffi.core.typesbase.CFFICDataWrapper*

See also:

https://msdn.microsoft.com/en-us/library/ms684342

hEvent

class pywincffi.wintypes.structures.**PROCESS_INFORMATION**
Bases: *pywincffi.core.typesbase.CFFICDataWrapper*

See also:

https://msdn.microsoft.com/en-us/library/ms684873

hProcess
Returns a *pywincffi.wintypes.objects.HANDLE* instance for the `hProcess` attribute.

hThread
Returns a *pywincffi.wintypes.objects.HANDLE* instance for the `hThread` attribute.

class pywincffi.wintypes.structures.**SECURITY_ATTRIBUTES**
Bases: *pywincffi.core.typesbase.CFFICDataWrapper*

See also:

https://msdn.microsoft.com/en-us/library/aa379560

nLength

class pywincffi.wintypes.structures.**STARTUPINFO**
Bases: *pywincffi.core.typesbase.CFFICDataWrapper*

See also:

https://msdn.microsoft.com/en-us/library/ms686331

hStdError
Returns a *pywincffi.wintypes.objects.HANDLE* instance for the `hStdError` attribute.

hStdInput
Returns a *pywincffi.wintypes.objects.HANDLE* instance for the `hStdInput` attribute.

hStdOutput
Returns a *pywincffi.wintypes.objects.HANDLE* instance for the `hStdOutput` attribute.

Module contents

Windows Types Package

Provides user accessible types corresponding to the respective Windows types used across the exposed APIs.

pywincffi.ws2_32 package

Submodules

pywincffi.ws2_32.events module

Events

A module containing Windows functions for working with events.

`pywincffi.ws2_32.events.`**`WSACreateEvent`**`()`
Creates a new event object.

See also:

https://msdn.microsoft.com/en-us/library/ms741561

Return type *pywincffi.wintypes.objects.WSAEVENT*

Returns Returns a handle to a new event object.

`pywincffi.ws2_32.events.`**`WSAEnumNetworkEvents`**`(`*socket*, *hEventObject=None*`)`
Discovers occurrences of network events on the indicated `socket`, clears internal events and optionally resets event objects.

See also:

https://msdn.microsoft.com/en-us/ms741572

Parameters

- **socket** (`pywincffi.wintypes.objects.SOCKET`) – The socket object to enumerate events for.
- **hEventObject** (`pywincffi.wintypes.objects.WSAEVENT`) – An optional handle identify an associated event object to be reset.

Return type *pywincffi.wintypes.structures.LPWSANETWORKEVENTS*

Returns

`pywincffi.ws2_32.events.`**`WSAEventSelect`**`(`*socket*, *hEventObject*, *lNetworkEvents*`)`
Specifies an event object to be associated with the specified set of FD_XXX network events.

See also:

https://msdn.microsoft.com/en-us/library/ms741576

Parameters

- **socket** (pywincffi.wintypes.objects.SOCKET) – The socket object to associate the selected network events with.
- **hEventObject** (pywincffi.wintypes.objects.WSAEVENT) – A handle which identifies the event object to be associated with the network events.
- **lNetworkEvents** (*int*) – A bitmask which specifies the combination of FD_XXX network events which the application has interest in.

pywincffi.ws2_32.events.**WSAGetLastError**()

Returns the last error status for a windows socket operation.

See also:

https://msdn.microsoft.com/en-us/library/ms741580

Module contents

Ws2_32 Sub-Package

Provides functions, constants and utilities that wrap functions provided by ws3_32.dll.

Submodules

pywincffi.exceptions module

Exceptions

Custom exceptions that pywincffi can throw.

exception pywincffi.exceptions.**ConfigurationError**

Bases: *pywincffi.exceptions.InternalError*

Raised when there was a problem with the configuration file

exception pywincffi.exceptions.**InputError**(*name*, *value*, *allowed_types=None*, *allowed_values=None*, *ffi=None*, *message=None*)

Bases: *pywincffi.exceptions.PyWinCFFIError*

A subclass of *PyWinCFFIError* that's raised when invalid input is provided to a function. Because we're passing inputs to C we have to be sure that the input(s) being provided are what we're expecting so we fail early and provide better error messages.

Parameters

- **name** (*str*) – The name of the parameter being checked.
- **value** – The value of the parameter being checked.
- **allowed_types** – The expected type(s). If provided then the exception's message will be tailored to provide information about value's type and the possible input types.
- **allowed_values** – The expected value(s). If provided then the exception's message will be tailored to provide information about what value(s) were allowed for value.

- **ffi** – If `value` is a C object, `ffi` is provided and `allowed_types` is provided as well then the provided `ffi` instance will be used to provide additional context.
- **message** (*str*) – A custom error message. This will override the default error messages which *InputError* would normally generate. This can be helpful if there is a problem with a given input parameter to a function but it's unrelated to the type of input.

exception pywincffi.exceptions.**InternalError**
Bases: *pywincffi.exceptions.PyWinCFFIError*

Raised if we encounter an internal error. Most likely this is an indication of a bug in pywincffi but it could also be a problem caused by an unexpected use case.

exception pywincffi.exceptions.**PyWinCFFIError**
Bases: exceptions.Exception

The base class for all custom exceptions that pywincffi can throw.

exception pywincffi.exceptions.**PyWinCFFINotImplementedError**
Bases: *pywincffi.exceptions.InternalError*

Raised if we encounter a situation where we can't figure out what to do. The message for this error should contain all the information necessary to implement a future work around.

exception pywincffi.exceptions.**ResourceNotFoundError**
Bases: *pywincffi.exceptions.InternalError*

Raised when we fail to locate a specific resource

exception pywincffi.exceptions.**WindowsAPIError**(*function*, *error*, *errno*, *return_code=None*, *expected_return_code=None*)
Bases: *pywincffi.exceptions.PyWinCFFIError*

A subclass of *PyWinCFFIError* that's raised when there was a problem calling a Windows API function.

Parameters

- **function** (*str*) – The Windows API function being called when the error was raised.
- **error** (*str*) – A string representation of the error message.
- **errno** (*int*) – An integer representing the error. This usually represents a constant which Windows has produced in response to a problem.
- **return_code** (*int*) – If the return value of a function has been checked the resulting code will be set as this value.
- **expected_return_code** (*int*) – The value we expected to receive for `code`.

Module contents

PyWinCFFI

The root of the pywincffi package. See the `README` and documentation for help and examples.

CHAPTER 3

Development

3.1 Development

The documents outlined here cover topics related to development of pywincffi. A high level overview of development can also be found in the README

3.1.1 Functions

This document provides detailed information on how to add new functions to pywincffi and should be treated as a general guide as the implementation may vary between functions.

Adding A New Windows Function

This section walks through adding a new Windows function, WriteFile, including some of the best practices on how to handle user input. As stated at the top of this documentation, these practices will likely vary some from function to function.

Function Definition

There are two parts to defining a new function. You must define the function in Python to wrap the underlying library function and you must define the function the C header so the function can be called in Python so consider this a guide book more than a set of rules.

C Header

The C header for function definitions is located in pywincffi/core/cdefs/headers/functions.h and is sometimes referred to the 'cdef'. When creating a new function you should essentially match what the msdn documentation defines. If you're implementing *WriteFile* for example you'd look at aa365747 and copy this into *functions.h* as:

```
BOOL WINAPI WriteFile(
   _In_        HANDLE       hFile,
   _In_        LPCVOID      lpBuffer,
   _In_        DWORD        nNumberOfBytesToWrite,
   _Out_opt_   LPDWORD      lpNumberOfBytesWritten,
   _Inout_opt_ LPOVERLAPPED lpOverlapped
);
```

It's important to note here that all inputs, output, optional arguments, etc are included in the header definition even if you don't plan on exposing them from the Python wrapper.

Location of C Definitions

Currently all C definitions reside in pywincffi/core/cdefs/headers/functions.h. Unlike the Python wrapper functions, which are discussed below, the C definition is not exposed to downstream consumers. The structure of the C definition files also does not impact how the wrapper functions are structured either since both pywincffi and the downstream consumers consume from *pywincffi.core.dist.load()*.

The C definition files could be better organized in the future if necessary. As it stands today this would only require minor changes to the *HEADER_FILES* and *SOURCE_FILES* globals in *pywincffi.core.dist*.

Python

Constructing The Wrapper

In order to make a Windows function available you need to write a 'wrapper' function. Technically speaking it's not a requirement in order to call the underlying C function however it makes the process of calling into a C function much easier for a consumer of pywincffi.

Getting back to the *WriteFile* example above and the aa365747 article from msdn, WriteFile has a few input and outputs

```
BOOL WINAPI WriteFile(
  _In_        HANDLE       hFile,                // input (required)
  _In_        LPCVOID      lpBuffer,             // input (required)
  _In_        DWORD        nNumberOfBytesToWrite,  // input (required)
  _Out_opt_   LPDWORD      lpNumberOfBytesWritten, // output (optional)
  _Inout_opt_ LPOVERLAPPED lpOverlapped          // input/output (optional)
);
```

When approaching a function like this, ask a few basic questions to compare the C implementation to Python:

- How do you write data to a file in Python?
- What arguments are required when you write data?
- What do you get out of the function(s) that can write data to a file?
- Are there functions in Python which are similar to the function being defined?

So in Python, the following input arguments are not normally required because Python typically handles them for you:

- **lpBuffer** - A buffer containing the data to write
- **nNumberOfBytesToWrite** - The number of bytes you intend to write

The only function which is similar to *WriteFile* is `os.write()` which takes a file descriptor and data to be written and returns the number of bytes written. So our implementation of *WriteFile* should be similar. In fact, it can look almost identical:

```
def WriteFile(hFile, lpBuffer): # -> bytes written
    pass
```

However since we're wrapping a Windows function and shouldn't artificially limit access to the underlying Windows API what should really be defined is:

```
def WriteFile(
    hFile, lpBuffer,
    nNumberOfBytesToWrite=None, lpOverlapped=None): # -> bytes written
    pass
```

Here's how the individual arguments would be handled inside of the function:

- **hFile** - A Windows handle must be created before being passed in. There is the `pywincffi.kernel32.handle_from_file()` function to help with going from a Python file object to Windows handle object.
- **lpBuffer** - String, bytes and unicode are converted to the appropriate C type before being passed to the C call.
- **nNumberOfBytesToWrite** - Can be determined from the size of lpBuffer or an integer can be provided.
- **lpOverlapped** - Optional according to msdn but someone can pass in their own overlapped structure if they wanted.

Location Of Wrapper Function

For the most part what module you decide to place *WriteFile* in is up to you however the module should be related to the function. *WriteFile* is meant to operate on files so it makes sense to include it in a *file* module. In Windows the *kernel32* library defines *WriteFile* so the subpackage the wrapper belongs to is also called *kernel32*:

```
pywincffi.kernel32.file.WriteFile <---- wrapper function
    ^          ^      ^
    |          |      |
  Root         |      |
 Package       |      |
        Subpackage/   |
        Windows Lib   |
                      |
               Object Type
                   or
             Operation Group
```

New functions which come from other Windows modules should add new top level subpackages.

Import Structure

In many Python programs, full import paths are often encouraged. So to import *WriteFile* one would do:

```
from pywincffi.kernel32.file import WriteFile
```

Internally within pywincffi, the above import path should be used. External consumers of pywincffi would import the function like this:

```
from pywincffi.kernel32 import WriteFile
```

So when you add a new function be sure to add it to the *__init__.py* for the subpackage. This ensures that if the import structure has to change within one of pywincffi's modules we're less likely to break downstream consumers.

Argument and Keyword Naming Conventions

If an argument or keyword is intended to be an analog for an argument to a Windows API call then it should follow the same naming convention as the documented function does. The *WaitForSingleObject* function for example takes two arguments according to the MSDN documentation which when translated to Python would look like this:

```
def WaitForSingleObject(hHandle, dwMilliseconds):
    pass
```

Any argument or keyword which is not directly related to an input to a Windows API should instead use the standard PEP8 naming conventions:

```
def WaitForSingleObject(hHandle, dwMilliseconds, other_keyword=None):
    pass
```

Internal Variables

Like arguments or keywords variables should be named either using *camelCase* if they're intended to map to a value passed into a Windows API call or using the *name_with_underscores* convention in other cases. Here's an example of the two:

```
def UnlockFileEx(...):

    # internal variables
    ffi, library = dist.load()

    # lpOverlapped is a Windows structure
    if lpOverlapped is None:
        lpOverlapped = ffi.new("OVERLAPPED[]", [{"hEvent": hFile}])
```

Documentation

This section covers the basics of documenting functions in pywincffi. The below mostly applies to how Windows functions should be documented but should generally apply elsewhere in the project too.

Basic Layout

The layout of the documentation string for each function should be consistent throughout the project. This generally makes it easier to understand but also harder to miss more critical information. Below is an annotated example of a fake Windows function:

```
def AWindowsFunction(...):
    """
    First few sentences should tell someone what AWindowsFunction
    does.  This can usually be pulled from the MSDN documentation but
    is usually shorter and more concise.
```

```
    .. seealso::

       <url pointing to the msdn reference for AWindowsFunction>
       <url pointing to a use case or other useful information>

    :param <python type> variable_name:
        Some information about what variable_name is.  Again, can be pulled
        from the msdn documentation but should be concise as someone can
        always go read the msdn documentation.  This information should
        always state key differences, if there are any, between what
        the C api call normally expects and what the wrapper does.

    <additional keyword or argument documentation>

    :raises SomeException:
        Information about under what condition(s) SomeException may be
        raised.  SomeException should be something that's raised directly
        by AWindowsFunction.


    :rtype: <The python type returned.  Required if different from the msdn docs>
    :returns:
        Some information about the return value.  This part of the
        documentation should be excluded if the function does not
        return anything.
    """
```

Arguments and Keywords

Position arguments should be documented using `:param <type> name:` while keywords should be documented using `:keyword <type> name:`. The `<type>` is referring to the Python type rather than the Windows type which the argument may be an analog for. Here's a simplified example:

```
def CreateFile(lpFileName, dwDesiredAccess, dwShareMode=None ...):
    """
    :param str lpFileName:

    :param int dwDesiredAccess:

    :keyword int dwShareMode:
    """
```

It's possible to allow an input argument to support multiple types as well:

```
def foobar(arg1):
    """
    :type arg1: int or str
    :param arg1:
    """
```

If the argument or keyword you are documenting requires some additional setup, such initializing a struct, it can be helpful to include a real example:

```
def CreatePipe(lpPipeAttribute=None):
    """
```

```
    ...

    :keyword struct lpPipeAttributes:
        The security attributes to apply to the handle. By default
        ``NULL`` will be passed in meaning then handle we create
        cannot be inherited.  Example struct:

        >>> from pywincffi.core import dist
        >>> ffi, library = dist.load()
        >>> lpPipeAttributes = ffi.new(
        ...     "SECURITY_ATTRIBUTES[1]", [{
        ...     "nLength": ffi.sizeof("SECURITY_ATTRIBUTES"),
        ...     "bInheritHandle": True,
        ...     "lpSecurityDescriptor": ffi.NULL
        ...     }]
        ... )
    """
```

External References

External references, such as those referencing the msdn documentation, are usually included within a `.. seealso::` block. For msdn documentation, this structure is usually preferable:

```
.. seealso::

   https://msdn.microsoft.com/en-us/library/<article_number>
```

Note: The documentation build, which is run for every commit, checks to ensure that the documents being referenced do in fact exist. If the url can't be reached the build will fail.

Handling Input

One of the main goals of pywincffi is to provide are more Python like interface for calling Windows APIs. To do this the pywincffi functions implement type checking, conversion and argument handling so less work is necessary on the consumer's part.

Type Checking

In order to provide better error messages and more consistent expectations of input arguments each function should perform type checking on each argument. Most type checks are run using the *pywincffi.core.checks.input_check()* function:

```
from six import integer_types
from pywincffi.core.checks import input_check

def Foobar(arg1, arg2):
    input_check("arg1", arg1, integer_types)
    input_check("arg1", arg2, allowed_values=(1, 2, 3))
```

If *pywincffi.core.checks.input_check()* does not do what you need or you have to perform multiple steps to validate an input argument you can raise the *pywincffi.exceptions.InputError* exception yourself.

Note: There are some enums to help with special cases (file handles, structure, etc) and more can be added. See pywincffi/core/checks.py

Type Conversion

The underlying library that pywincffi uses, cffi, can do most type conversions for you. While normally this will function as you'd expect it's better to be explicit and handle the conversion yourself so there are fewer surprises.

Here's an example of how an 'automatic' conversion would look:

```
library.LockFileEx(hFile, 0, 0, 0, 0, lpOverlapped)
```

The problem is it makes it easier to pass something into *LockFileEx* that cffi might not know how to convert. The error produced as a result may look strange to someone unfamiliar with cffi and it could be more difficult to debug as result.

To avoid this problem pywincffi should try to perform the cast manually before making calls to the underlying API call. This ensures that cffi shouldn't need to do the conversion itself and limits the chance of lower level errors propagating:

```
library.LockFileEx(
   hFile,
   ffi.cast("DWORD", 0),
   ffi.cast("DWORD", 0),
   ffi.cast("DWORD", 0),
   ffi.cast("DWORD", 0),
   lpOverlapped
)
```

Keywords

In C, there's not really an equivalent to a keyword in Python. However for many of the Windows API functions the msdn documentation may say something along the lines of *This parameter can be NULL*. For pywincffi, reasonable default values should be defined where possible so not every argument is always required.

As an example the *lpSecurityAttributes* argument for *CreateFile* can be *NULL* and would be handled like this:

```
def CreateFile(..., lpSecurityAttributes=None):
   ffi, library = dist.load()

   if lpSecurityAttributes is None:
      lpSecurityAttributes = ffi.NULL
```

Attention: Be sure that if a keyword is in fact required in some cases but not others that you raise InputError when the required keyword is not provided.

Handling Output

Many Windows functions have a return value and some return values will be stored in another variable rather returned directly from the API call. This section tries to detail a couple of different cases and how to handle them.

Windows API Error Checking

When calling a Windows function it's the responsibility of the wrapper function in pywincffi to check for errors using the *pywincffi.core.checks.error_check()* function:

```
from pywincffi.core.checks import Enums, error_check

def WriteFile(...):
   code = library.WriteFile(
        hFile, lpBuffer, nNumberOfBytesToWrite, bytes_written, lpOverlapped)
    error_check("WriteFile", code=code, expected=Enums.NON_ZERO)
```

This ensures that when an API does fail pywincffi will raise a consistent error with as much information as possible to help the consumer of the API determine what the problem is.

API Return Values

If a function returns a handle, structure, etc it's usually best to return this from the wrapper function too. Be sure the wrapper functions's documentation provides an example if accessing or using the data requires a couple of extra steps.

Windows Constants

When it comes to Windows constants code in Python you'll often seen one of two kinds of definitions:

```
FILE_ATTRIBUTE_ENCRYPTED = 0x4000  # matches the msdn reference
FILE_ATTRIBUTE_ENCRYPTED = 16384  # same as the above but turn into an int
```

While neither of these are incorrect there are a few problems with making constants this way:

- It's easy to insert a typo into a variable name or its value.
- You have to rely on code review to check for correctness.
- They're not true constants and could be modified at runtime.

So in pywincffi, we usually define constants in pywincffi/core/cdefs/headers/constants.h. At compile time any typos will result in build errors and the values are replaced when the library is compiled.

Adding New Constants

To add a new constant, simply define a line in pywincffi/core/cdefs/headers/constants.h:

```
#define FILE_ATTRIBUTE_ENCRYPTED ...
```

When should new constants be defined? It varies but it's good general practice to define all of the constants mentioned in the msdn documentation for the function you are working on. So for example if you're working on the `SetHandleInformation` function the documentation at ms724935 would have you define two constants as a result:

```
#define HANDLE_FLAG_INHERIT ...
#define HANDLE_FLAG_PROTECT_FROM_CLOSE ...
```

Using Existing Constants

When developing code for pywincffi, either within the library itself or the tests, constants should be used instead of default values. To access a defined constant you'll need to load the library:

```
from pywincffi.core import dist
_, library = dist.load()
library.FILE_ATTRIBUTE_ENCRYPTED
```

3.1.2 Coding Style and Conventions

This document covers some specific coding conventions and style choices for pywincffi that may not be covered by other development documentation.

Single vs. Double Quotes

Python has two kinds of quotes, `'` and `"`. The language itself does not treat the two any differently however other languages, like C, do. For the purposes of pywincffi all strings should be constructed with `"` unless there's a specific reason not to. This is mostly for internal consistency but also because "hello" is how you'd expect to see a string literal in C. Though Python is not C pywincffi can and does deal with C APIs so it's less of a cognitive jump to just stick with `"`.

Windows Constants

This project uses and reference a lot of constants in Windows. For consistency and readability we should always use Windows constants by name rather than hard coding numbers.

For example if you're writing a test or code like this:

```
SetHandleInformation(handle, 1, 0)
```

Then it would be preferable to write:

```
_, library = dist.load()
SetHandleInformation(handle, library.HANDLE_FLAG_INHERIT, 0)
```

3.1.3 Code Review

This document gives a basic overview of code reviews for the pywincffi projects. All code reviews are performed on GitHub by using pull requests. Information about pull requests and how to submit one can be found here:

> https://help.github.com/articles/using-pull-requests/

What branch should I use?

You should always base your code from the master branch unless you've been told otherwise. The master branch should be considered production ready and other branches are usually for testing and development.

What Will Be Reviewed

- If a new function is being added, review the function documentation and make sure the new code matches these expectations.
- For style issues, the default rule of thumb is to follow PEP8 unless it's something Windows specific. Then, the function documentation should be referenced.
- Does the new function include documentation? Are there comments for special cases?
- Tests - Generally speaking, most changes should include a combination of unit and integration like tests. Just calling the functions and ensuring they don't raise errors is sometimes ok but it's usually best to also ensure that the function works under 'real life' conditions. For example if you are testing file locking, try accessing the file in another process.

Pre-Merge Requirements

The following are required before a pull request can normally be merged:

- All conflicts with the target branch should be resolved.
- The unittests, linters and other checks run on AppVeyor must pass.
- There should not be any major drops in coverage. If there are it will be up to the reviewer(s) if the pull request should merge.
- A brief description of the changes should be included in `docs/changelog.rst` under the 'latest' version.
- Breaking changes should not occur on minor or micro versions unless the existing behavior can be preserved somehow.

3.1.4 Vagrant

Vagrant is used by the pywincffi project to facilitate testing and development on non-windows platforms. While the project does have continuous integration hooked up to commits and pull requests Vagrant can help with local development. The information below details the general steps needed to get Vagrant up and running.

Prerequisites

Before starting, you will need a pieces of software preinstalled on your system:

- Vagrant - The software used to launch and provision the virtual machine image.
- Packer - Used to build a virtual machine image, referred to as a box, which Vagrant can then use.

Building The Base Virtual Machine Image

In order to effectively run and test pywincffi you must have access to a Windows host, various versions of Python and a couple of different compilers. While you can rely on continuous integration to provide this it's faster to test locally usually.

The install process for the various dependencies besides the operating system will be covered in another section. This section will cover setting up the base machine image itself.

1. Use git to clone the packer templates:

```
git clone https://github.com/mwrock/packer-templates.git
```

This repository contains all the code necessary to build our base image. For some extra information on how this works you can see this article:

http://www.hurryupandwait.io/blog/creating-windows-base-images-for-virtualbox-and-hyper-v-using-packer-boxstarter-an

2. Run packer. This will generate the box image which vagrant will need to spin up a virtual machine.

```
cd packer-templates
packer build -force -only virtualbox-iso ./vbox-2012r2.json
```

The above will take a while to run. When complete you should end up with a file like `windows2012r2min-virtualbox.box` on disk.

3. Add the box image to vagrant:

```
vagrant box add windows2012r2min-virtualbox.box --name windows2012r2min
```

At this point, you should have everything you need to launch vagrant with a Windows image.

Note: The box that was generated is using an evaluation copy of Windows 2012 R2 Standard which expires in 180 days. You will either need to add a license for the operating system or repeat the steps outlined above again later on.

Running Vagrant

Vagrant is responsible for running the virtual machine as well as installing and downloading the necessary software for pywincffi. The process for launching vagrant is:

```
cd <path to clone of pywincffi>
vagrant up --provider virtualbox
```

This will start up the virtual machine, download the necessary software and get it installed on the system.

Important: At certain points during the install you will be required to perform some manual steps. This is because certain software, such as Visual Studio express editions, can't easily be installed in an unattended manner.

Rerunning The Provisioning Step

Sometimes you might need to execute the provisioning process again. This could be because one of the steps failed when running `vagrant up`, you've added a new step to the Vagrantfile or you've modified a step in `.ci/vagrant/`.

To reexecute the provisioning process on a running VM run:

```
vagrant provision
```

To restart the VM and execute the provisioning process run:

```
vagrant reload --provision
```

Installing Python Source Code

By default, going back over *rerunning the provisioning step* will install the source code for you. If you make changes however to the setup.py file or something seems broken you can force the provision process to run again and skip the OS steps:

```
vagrant provision --provision-with python,install
```

Adding SSH Authorized Keys

SSH for the Windows VM is setup to use key based authentication. To provide you own set of keys, create a file at `.ci/vagrant/files/authorized_keys` with your own public key(s).

pywincffi ships `.ci/vagrant/files/authorized_keys.template` which contains vagrant's public key. You're welcome to copy this over and add your own keys. By doing this, you'll be able to run `vagrant ssh` in addition to being able to use ssh directly with your own key.

In addition you can also use the `vagrant` password for either the vagrant account or the Administrator account to login manually if needed.

Testing PyWinCFFI

PyCharm Remote Interpreter

If you're using PyCharm you can take advantage of its remote interpreter feature. This will allow you to execute the tests as if Python is running locally even though it's in a virtual machine.

For more information on how to set this up, check out these guides direct from JetBrains:

- https://www.jetbrains.com/help/pycharm/2016.1/configuring-remote-python-interpreters.html
- https://www.jetbrains.com/help/pycharm/2016.1/tutorial-configuring-pycharm-to-work-on-the-vm.html

---

Note: Some of the features above may require the professional version of PyCharm.

Manually Testing Using Vagrant

Warning: This method of testing does not work currently. Please use one of these methods instead:

- *PyCharm Remote Interpreter*
- *Manually Using SSH and CYGWIN*

Issue: https://github.com/opalmer/pywincffi/issues/28

Before attempting to test be sure the core Python interpreters have been installed:

```
vagrant provision --provision-with python,install
```

If you add a new module or the tests seem to be failing due to recent project changes you can rerun the above steps.

Next, execute the tests:

```
vagrant provision --provision-with test
```

Manually Using SSH and CYGWIN

You can also manually test the project as well over ssh.

```
$ ssh -p 2244 vagrant@localhost
$ cd /cygdrive/c/code
$ ~/virtualenv/2.7.10-x86/Scripts/python.exe setup.py test
[ ... ]
----------------------------------------------------------------------
Ran 70 tests in 0.359s

OK
```

Python Module Index

p

Index

S

T

U

W