ADB Shell Documentation

Release 0.4.4

Jeff Irion

Oct 26, 2023

CONTENTS

CHAPTER

ONE

ADB_SHELL

1.1 adb_shell package

1.1.1 Subpackages

adb_shell.auth package

Submodules

adb_shell.auth.keygen module

This file implements encoding and decoding logic for Android's custom RSA public key binary format. Public keys are stored as a sequence of little-endian 32 bit words. Note that Android only supports little-endian processors, so we don't do any byte order conversions when parsing the binary struct.

Structure from: https://github.com/aosp-mirror/platform_system_core/blob/c55fab4a59cfa461857c6a61d8a0f1ae4591900c/libcrypto_utils/android_pubkey.c

```
typedef struct RSAPublicKey {
    // Modulus length. This must be ANDROID_PUBKEY_MODULUS_SIZE_WORDS
    uint32_t modulus_size_words;

    // Precomputed montgomery parameter: -1 / n[0] mod 2^32
    uint32_t n0inv;

    // RSA modulus as a little-endian array
    uint8_t modulus[ANDROID_PUBKEY_MODULUS_SIZE];

    // Montgomery parameter R^2 as a little-endian array of little-endian words
    uint8_t rr[ANDROID_PUBKEY_MODULUS_SIZE];

    // RSA modulus: 3 or 65537
    uint32_t exponent;
} RSAPublicKey;
```

Contents

adb_shell.auth.keygen.**ANDROID_PUBKEY_MODULUS_SIZE = 256**

Size of an RSA modulus such as an encrypted block or a signature.

adb_shell.auth.keygen.**ANDROID_PUBKEY_MODULUS_SIZE_WORDS = 64**

Size of the RSA modulus in words.

adb_shell.auth.keygen.**ANDROID_RSAPUBLICKEY_STRUCT = '<LL256s256sL'**

Python representation of "struct RSAPublicKey"

adb_shell.auth.keygen.**_to_bytes**(*n*, *length*, *endianess='big'*)

Partial python2 compatibility with int.to_bytes

https://stackoverflow.com/a/20793663

Parameters

- **n** (*TODO*) – TODO
- **length** (*TODO*) – TODO
- **endianess** (*str, TODO*) – TODO

Returns
TODO

Return type
TODO

adb_shell.auth.keygen.**decode_pubkey**(*public_key*)

Decode a public RSA key stored in Android's custom binary format.

Parameters
public_key (*TODO*) – TODO

adb_shell.auth.keygen.**decode_pubkey_file**(*public_key_path*)

TODO

Parameters
public_key_path (*str*) – TODO

adb_shell.auth.keygen.**encode_pubkey**(*private_key_path*)

Encodes a public RSA key into Android's custom binary format.

Parameters
private_key_path (*str*) – TODO

Returns
TODO

Return type
TODO

adb_shell.auth.keygen.**get_user_info**()

TODO

Returns
' <username>@<hostname>

Return type
str

adb_shell.auth.keygen.**keygen**(*filepath*)

Generate an ADB public/private key pair.

- The private key is stored in `filepath`.
- The public key is stored in `filepath + '.pub'`

(Existing files will be overwritten.)

Parameters
filepath (*str*) – File path to write the private/public keypair

adb_shell.auth.keygen.**write_public_keyfile**(*private_key_path*, *public_key_path*)

Write a public keyfile to `public_key_path` in Android's custom RSA public key format given a path to a private keyfile.

Parameters

- **private_key_path** (*TODO*) – TODO
- **public_key_path** (*TODO*) – TODO

adb_shell.auth.sign_cryptography module

ADB authentication using the `cryptography` package.

Contents

class adb_shell.auth.sign_cryptography.**CryptographySigner**(*rsa_key_path*)

Bases: `object`

AuthSigner using cryptography.io.

Parameters
rsa_key_path (*str*) – The path to the private key.

public_key

The contents of the public key file

Type
str

rsa_key

The loaded private key

Type
cryptography.hazmat.backends.openssl.rsa._RSAPrivateKey

GetPublicKey()

Returns the public key in PEM format without headers or newlines.

Returns
self.public_key – The contents of the public key file

Return type
str

Sign(*data*)

Signs given data using a private key.

Parameters
data (*TODO*) – TODO

Returns
The signed `data`

Return type
TODO

adb_shell.auth.sign_pycryptodome module

ADB authentication using `pycryptodome`.

Contents

class adb_shell.auth.sign_pycryptodome.**PycryptodomeAuthSigner**(*rsa_key_path=None*)

Bases: `object`

AuthSigner using the pycryptodome package.

Parameters
rsa_key_path (*str, None*) – The path to the private key

public_key

The contents of the public key file

Type
str

rsa_key

The contents of theprivate key

Type
Crypto.PublicKey.RSA.RsaKey

`GetPublicKey`()

Returns the public key in PEM format without headers or newlines.

Returns
self.public_key – The contents of the public key file

Return type
str

`Sign`(*data*)

Signs given data using a private key.

Parameters
data (*bytes, bytearray*) – The data to be signed

Returns
The signed `data`

Return type
bytes

adb_shell.auth.sign_pythonrsa module

ADB authentication using the `rsa` package.

Contents

`class` adb_shell.auth.sign_pythonrsa.`PythonRSASigner`(*pub=None*, *priv=None*)

Bases: `object`

Implements `adb_protocol.AuthSigner` using http://stuvel.eu/rsa.

Parameters
- **pub** (*str, None*) – The contents of the public key file
- **priv** (*str, None*) – The contents of the private key file

`priv_key`

The loaded private key

Type
rsa.key.PrivateKey

pub_key

The contents of the public key file

Type
str, None

classmethod FromRSAKeyPath(*rsa_key_path*)

Create a *PythonRSASigner* instance using the provided private key.

Parameters
rsa_key_path (*str*) – The path to the private key; the public key must be `rsa_key_path + '.pub'`.

Returns
A *PythonRSASigner* with private key `rsa_key_path` and public key `rsa_key_path + '.pub'`

Return type
PythonRSASigner

GetPublicKey()

Returns the public key in PEM format without headers or newlines.

Returns
self.pub_key – The contents of the public key file, or `None` if a public key was not provided.

Return type
str, None

Sign(*data*)

Signs given data using a private key.

Parameters
data (*bytes*) – The data to be signed

Returns
The signed `data`

Return type
bytes

class adb_shell.auth.sign_pythonrsa.**_Accum**

Bases: `object`

A fake hashing algorithm.

The Python `rsa` lib hashes all messages it signs. ADB does it already, we just need to slap a signature on top of already hashed message. Introduce a "fake" hashing algo for this.

_buf

A buffer for storing data before it is signed

Type
bytes

digest()

Return the digest value as a string of binary data.

Returns
self._buf – `self._buf`

Return type
bytes

update(*msg*)

Update this hash object's state with the provided `msg`.

Parameters
msg (*bytes*) – The message to be appended to `self._buf`

adb_shell.auth.sign_pythonrsa.**_load_rsa_private_key**(*pem*)

PEM encoded PKCS#8 private key -> `rsa.PrivateKey`.

ADB uses private RSA keys in pkcs#8 format. The `rsa` library doesn't support them natively. Do some ASN unwrapping to extract naked RSA key (in der-encoded form).

See:

- https://www.ietf.org/rfc/rfc2313.txt
- http://superuser.com/a/606266

Parameters
pem (*str*) – The private key to be loaded

Returns
The loaded private key

Return type
rsa.key.PrivateKey

Module contents

adb_shell.transport package

Submodules

adb_shell.transport.base_transport module

A base class for transports used to communicate with a device.

class adb_shell.transport.base_transport.**BaseTransport**

Bases: `ABC`

A base transport class.

_abc_impl = <_abc._abc_data object>

abstract bulk_read(*numbytes*, *transport_timeout_s*)

Read data from the device.

Parameters

- **numbytes** (*int*) – The maximum amount of data to be received
- **transport_timeout_s** (*float, None*) – A timeout for the read operation

Returns
The received data

Return type
bytes

abstract bulk_write(*data*, *transport_timeout_s*)

Send data to the device.

Parameters

- **data** (*bytes*) – The data to be sent
- **transport_timeout_s** (*float, None*) – A timeout for the write operation

Returns
The number of bytes sent

Return type
int

abstract close()

Close the connection.

abstract connect(*transport_timeout_s*)

Create a connection to the device.

Parameters
transport_timeout_s (*float, None*) – A connection timeout

adb_shell.transport.base_transport_async module

A base class for transports used to communicate with a device.

class adb_shell.transport.base_transport_async.**BaseTransportAsync**

Bases: ABC

A base transport class.

_abc_impl = <_abc._abc_data object>

abstract async bulk_read(*numbytes*, *transport_timeout_s*)

Read data from the device.

Parameters

- **numbytes** (*int*) – The maximum amount of data to be received
- **transport_timeout_s** (*float, None*) – A timeout for the read operation

Returns
The received data

Return type
bytes

abstract async bulk_write(*data*, *transport_timeout_s*)

Send data to the device.

Parameters

- **data** (*bytes*) – The data to be sent
- **transport_timeout_s** (*float, None*) – A timeout for the write operation

Returns
The number of bytes sent

Return type
int

abstract async close()

Close the connection.

abstract async connect(*transport_timeout_s*)

Create a connection to the device.

Parameters
transport_timeout_s (*float, None*) – A connection timeout

adb_shell.transport.tcp_transport module

A class for creating a socket connection with the device and sending and receiving data.

class adb_shell.transport.tcp_transport.**TcpTransport**(*host*, *port=5555*)

Bases: *BaseTransport*

TCP connection object.

Parameters

- **host** (*str*) – The address of the device; may be an IP address or a host name
- **port** (*int*) – The device port to which we are connecting (default is 5555)

_connection

A socket connection to the device

Type
socket.socket, None

_host

The address of the device; may be an IP address or a host name

Type
str

_port

The device port to which we are connecting (default is 5555)

Type
int

_abc_impl = <_abc._abc_data object>

bulk_read(*numbytes*, *transport_timeout_s*)

Receive data from the socket.

Parameters

- **numbytes** (*int*) – The maximum amount of data to be received
- **transport_timeout_s** (*float, None*) – When the timeout argument is omitted, `select.select` blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

Returns
The received data

Return type
bytes

Raises
TcpTimeoutException – Reading timed out.

bulk_write(*data*, *transport_timeout_s*)

Send data to the socket.

Parameters

- **data** (*bytes*) – The data to be sent
- **transport_timeout_s** (*float, None*) – When the timeout argument is omitted, `select.select` blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

Returns
The number of bytes sent

Return type
int

Raises
TcpTimeoutException – Sending data timed out. No data was sent.

close()

Close the socket connection.

connect(*transport_timeout_s*)

Create a socket connection to the device.

Parameters

transport_timeout_s (*float, None*) – Set the timeout on the socket instance

adb_shell.transport.tcp_transport_async module

A class for creating a socket connection with the device and sending and receiving data.

class adb_shell.transport.tcp_transport_async.**TcpTransportAsync**(*host*, *port=5555*)

Bases: *BaseTransportAsync*

TCP connection object.

Parameters

- **host** (*str*) – The address of the device; may be an IP address or a host name
- **port** (*int*) – The device port to which we are connecting (default is 5555)

_host

The address of the device; may be an IP address or a host name

Type
str

_port

The device port to which we are connecting (default is 5555)

Type
int

_reader

Object for reading data from the socket

Type
StreamReader, None

_writer

Object for writing data to the socket

Type
StreamWriter, None

_abc_impl = <_abc._abc_data object>

async bulk_read(*numbytes*, *transport_timeout_s*)

Receive data from the socket.

Parameters

- **numbytes** (*int*) – The maximum amount of data to be received

- **transport_timeout_s** (*float, None*) – Timeout for reading data from the socket; if it is None, then it will block until the read operation completes

Returns
The received data

Return type
bytes

Raises
TcpTimeoutException – Reading timed out.

async bulk_write(*data*, *transport_timeout_s*)

Send data to the socket.

Parameters

- **data** (*bytes*) – The data to be sent
- **transport_timeout_s** (*float, None*) – Timeout for writing data to the socket; if it is None, then it will block until the write operation completes

Returns
The number of bytes sent

Return type
int

Raises
TcpTimeoutException – Sending data timed out. No data was sent.

async close()

Close the socket connection.

async connect(*transport_timeout_s*)

Create a socket connection to the device.

Parameters
transport_timeout_s (*float, None*) – Timeout for connecting to the socket; if it is None, then it will block until the operation completes

adb_shell.transport.usb_transport module

A class for creating a USB connection with the device and sending and receiving data.

Warning: USB support is an experimental feature.

- *get_interface()*
- *interface_matcher()*
- *UsbTransport*
  - *UsbTransport._find()*
  - *UsbTransport._find_and_open()*
  - *UsbTransport._find_devices()*
  - *UsbTransport._find_first()*

- *UsbTransport._flush_buffers()*
- *UsbTransport._open()*
- *UsbTransport._port_path_matcher()*
- *UsbTransport._serial_matcher()*
- UsbTransport._timeout()
- *UsbTransport.bulk_read()*
- *UsbTransport.bulk_write()*
- *UsbTransport.close()*
- *UsbTransport.connect()*
- *UsbTransport.port_path*
- *UsbTransport.serial_number*
- *UsbTransport.usb_info*

adb_shell.transport.usb_transport.**DEFAULT_TIMEOUT_S = 10**

Default timeout

class adb_shell.transport.usb_transport.**UsbTransport**(*device*, *setting*, *usb_info=None*, *default_transport_timeout_s=None*)

Bases: *BaseTransport*

USB communication object. Not thread-safe.

Handles reading and writing over USB with the proper endpoints, exceptions, and interface claiming.

Parameters

- **device** (*usb1.USBDevice*) – libusb_device to connect to.
- **setting** (*usb1.USBInterfaceSetting*) – libusb setting with the correct endpoints to communicate with.
- **usb_info** (*TODO, None*) – String describing the usb path/serial/device, for debugging.
- **default_transport_timeout_s** (*TODO, None*) – Timeout in seconds for all I/O.

_default_transport_timeout_s

Timeout in seconds for all I/O.

Type
TODO, None

_device

libusb_device to connect to.

Type
TODO

_transport

TODO

Type
TODO

_interface_number

TODO

Type
TODO

_max_read_packet_len

TODO

Type
TODO

_read_endpoint

TODO

Type
TODO

_setting

libusb setting with the correct endpoints to communicate with.

Type
TODO

_usb_info

String describing the usb path/serial/device, for debugging.

Type
TODO

_write_endpoint

TODO

Type
TODO, None

_HANDLE_CACHE = <WeakValueDictionary>

_HANDLE_CACHE_LOCK = <unlocked _thread.lock object>

_abc_impl = <_abc._abc_data object>

classmethod _find(*setting_matcher*, *port_path=None*, *serial=None*, *default_transport_timeout_s=None*)

Gets the first device that matches according to the keyword args.

Parameters

- **setting_matcher** (*TODO*) – TODO
- **port_path** (*TODO*, *None*) – TODO
- **serial** (*TODO*, *None*) – TODO
- **default_transport_timeout_s** (*TODO*, *None*) – TODO

Returns
TODO

Return type
TODO

classmethod _find_and_open(*setting_matcher*, *port_path=None*, *serial=None*, *default_transport_timeout_s=None*)

TODO

Parameters

- **setting_matcher** (*TODO*) – TODO
- **port_path** (*TODO*, *None*) – TODO
- **serial** (*TODO*, *None*) – TODO
- **default_transport_timeout_s** (*TODO*, *None*) – TODO

Returns
dev – TODO

Return type
TODO

classmethod _find_devices(*setting_matcher*, *device_matcher=None*, *usb_info=''*, *default_transport_timeout_s=None*)

_find and yield the devices that match.

Parameters

- **setting_matcher** (*TODO*) – Function that returns the setting to use given a `usb1.USBDevice`, or `None` if the device doesn't have a valid setting.
- **device_matcher** (*TODO*, *None*) – Function that returns `True` if the given `UsbTransport` is valid. `None` to match any device.
- **usb_info** (*str*) – Info string describing device(s).
- **default_transport_timeout_s** (*TODO*, *None*) – Default timeout of commands in seconds.

Yields
TODO – UsbTransport instances

classmethod _find_first(*setting_matcher*, *device_matcher=None*, *usb_info=''*, *default_transport_timeout_s=None*)

Find and return the first matching device.

Parameters

- **setting_matcher** (*TODO*) – Function that returns the setting to use given a `usb1.USBDevice`, or `None` if the device doesn't have a valid setting.
- **device_matcher** (*TODO*) – Function that returns `True` if the given `UsbTransport` is valid. `None` to match any device.
- **usb_info** (*str*) – Info string describing device(s).
- **default_transport_timeout_s** (*TODO*, *None*) – Default timeout of commands in seconds.

Returns
An instance of *UsbTransport*

Return type
TODO

Raises
adb_shell.exceptions.DeviceNotFoundError – Raised if the device is not available.

_flush_buffers()

TODO

Raises
adb_shell.exceptions.UsbReadFailedError – TODO

_open()

Opens the USB device for this setting, and claims the interface.

classmethod _port_path_matcher(*port_path*)

Returns a device matcher for the given port path.

Parameters
port_path (*TODO*) – TODO

Returns
TODO

Return type
function

classmethod _serial_matcher(*serial*)

Returns a device matcher for the given serial.

Parameters
serial (*TODO*) – TODO

Returns
TODO

Return type
function

_timeout_ms(*transport_timeout_s*)

TODO

Returns
TODO

Return type
TODO

bulk_read(*numbytes*, *transport_timeout_s=None*)

Receive data from the USB device.

Parameters

- **numbytes** (*int*) – The maximum amount of data to be received
- **transport_timeout_s** (*float, None*) – When the timeout argument is omitted, `select.select` blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

Returns
The received data

Return type
bytes

Raises
adb_shell.exceptions.UsbReadFailedError – Could not receive data

bulk_write(*data*, *transport_timeout_s=None*)

Send data to the USB device.

Parameters

- **data** (*bytes*) – The data to be sent
- **transport_timeout_s** (*float, None*) – When the timeout argument is omitted, `select.select` blocks until at least one file descriptor is ready. A time-out value of zero specifies a poll and never blocks.

Returns
The number of bytes sent

Return type
int

Raises

- ***adb_shell.exceptions.UsbWriteFailedError*** – This transport has been closed, probably due to another being opened
- ***adb_shell.exceptions.UsbWriteFailedError*** – Could not send data

close()

Close the USB connection.

connect(*transport_timeout_s=None*)

Create a USB connection to the device.

Parameters
transport_timeout_s (*float, None*) – Set the timeout on the USB instance

classmethod find_adb(*serial=None*, *port_path=None*, *default_transport_timeout_s=None*)

TODO

Parameters

- **serial** (*TODO*) – TODO
- **port_path** (*TODO*) – TODO
- **default_transport_timeout_s** (*TODO, None*) – Default timeout of commands in seconds.

Returns
TODO

Return type
UsbTransport

classmethod find_all_adb_devices(*default_transport_timeout_s=None*)

Find all ADB devices attached via USB.

Parameters
default_transport_timeout_s (*TODO, None*) – Default timeout of commands in seconds.

Returns
A generator which yields each ADB device attached via USB.

Return type
generator

property port_path

TODO

Returns
TODO

Return type
TODO

property serial_number

TODO

Returns
TODO

Return type
TODO

property usb_info

TODO

Returns
TODO

Return type
TODO

adb_shell.transport.usb_transport.**get_interface**(*setting*)

Get the class, subclass, and protocol for the given USB setting.

Parameters
setting (*TODO*) – TODO

Returns

- *TODO* – TODO
- *TODO* – TODO
- *TODO* – TODO

adb_shell.transport.usb_transport.**interface_matcher**(*clazz*, *subclass*, *protocol*)

Returns a matcher that returns the setting with the given interface.

Parameters

- **clazz** (*TODO*) – TODO
- **subclass** (*TODO*) – TODO
- **protocol** (*TODO*) – TODO

Returns
matcher – TODO

Return type
function

Module contents

1.1.2 Submodules

adb_shell.adb_device module

Implement the *AdbDevice* class, which can connect to a device and run ADB shell commands.

Contents

  - *AdbDevice.connect()*
  - *AdbDevice.list()*
  - *AdbDevice.max_chunk_size*
  - *AdbDevice.pull()*
  - *AdbDevice.push()*
  - *AdbDevice.root()*
  - *AdbDevice.shell()*
  - *AdbDevice.stat()*
  - *AdbDevice.streaming_shell()*
- *AdbDeviceTcp*
- *AdbDeviceUsb*

class adb_shell.adb_device.**AdbDevice**(*transport*, *default_transport_timeout_s=None*, *banner=None*)

Bases: `object`

A class with methods for connecting to a device and executing ADB commands.

Parameters

- **transport** (BaseTransport) – A user-provided transport for communicating with the device; must be an instance of a subclass of *BaseTransport*
- **default_transport_timeout_s** (*float, None*) – Default timeout in seconds for transport packets, or `None`
- **banner** (*str, bytes, None*) – The hostname of the machine where the Python interpreter is currently running; if it is not provided, it will be determined via `socket.gethostname()`

Raises

adb_shell.exceptions.InvalidTransportError – The passed `transport` is not an instance of a subclass of *BaseTransport*

_available

Whether an ADB connection to the device has been established

Type
bool

_banner

The hostname of the machine where the Python interpreter is currently running

Type
bytearray, bytes

_default_transport_timeout_s

Default timeout in seconds for transport packets, or `None`

Type
float, None

_io_manager

Used for handling all ADB I/O

Type
_AdbIOManager

_local_id

The local ID that is used for ADB transactions; the value is incremented each time and is always in the range `[1, 2^32)`

Type
int

_local_id_lock

A lock for protecting `_local_id`; this is never held for long

Type
Lock

_maxdata

Maximum amount of data in an ADB packet

Type
int

_clse(*adb_info*)

Send a `b'CLSE'` message and then read a `b'CLSE'` message.

> **Warning:** This is not to be confused with the *AdbDevice.close()* method!

Parameters
adb_info (_AdbTransactionInfo) – Info and settings for this ADB transaction

_filesync_flush(*adb_info*, *filesync_info*)

Write the data in the buffer up to `filesync_info.send_idx`, then set `filesync_info.send_idx` to 0.

Parameters

- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction

_filesync_read(*expected_ids*, *adb_info*, *filesync_info*)

Read ADB messages and return FileSync packets.

Parameters

- **expected_ids** (*tuple[bytes]*) – If the received header ID is not in `expected_ids`, an exception will be raised
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction

Returns

- **command_id** (*bytes*) – The received header ID
- *tuple* – The contents of the header
- **data** (*bytearray, None*) – The received data, or `None` if the command ID is `adb_shell.constants.STAT`

Raises

- *adb_shell.exceptions.AdbCommandFailureException* – Command failed
- *adb_shell.exceptions.InvalidResponseError* – Received response was not in `expected_ids`

_filesync_read_buffered(*size*, *adb_info*, *filesync_info*)

Read `size` bytes of data from `self.recv_buffer`.

Parameters

- **size** (*int*) – The amount of data to read
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction

Returns

result – The read data

Return type

bytearray

_filesync_read_until(*expected_ids*, *finish_ids*, *adb_info*, *filesync_info*)

Useful wrapper around *AdbDevice._filesync_read()*.

Parameters

- **expected_ids** (*tuple[bytes]*) – If the received header ID is not in `expected_ids`, an exception will be raised
- **finish_ids** (*tuple[bytes]*) – We will read until we find a header ID that is in `finish_ids`
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction

Yields

- **cmd_id** (*bytes*) – The received header ID
- **header** (*tuple*) – TODO
- **data** (*bytearray*) – The received data

_filesync_send(*command_id*, *adb_info*, *filesync_info*, *data=b''*, *size=None*)

Send/buffer FileSync packets.

Packets are buffered and only flushed when this connection is read from. All messages have a response from the device, so this will always get flushed.

Parameters

- **command_id** (*bytes*) – Command to send.
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction
- **data** (*str, bytes*) – Optional data to send, must set data or size.
- **size** (*int, None*) – Optionally override size from len(data).

_get_transport_timeout_s(*transport_timeout_s*)

Use the provided `transport_timeout_s` if it is not `None`; otherwise, use `self._default_transport_timeout_s`

Parameters
transport_timeout_s (*float, None*) – The potential transport timeout

Returns
`transport_timeout_s` if it is not `None`; otherwise, `self._default_transport_timeout_s`

Return type
float

_okay(*adb_info*)

Send an `b'OKAY'` mesage.

Parameters
adb_info (_AdbTransactionInfo) – Info and settings for this ADB transaction

_open(*destination*, *transport_timeout_s*, *read_timeout_s*, *timeout_s*)

Opens a new connection to the device via an `b'OPEN'` message.

1. *send()* an `b'OPEN'` command to the device that specifies the `local_id`
2. *read()* the response from the device and fill in the `adb_info.remote_id` attribute

Parameters

- **destination** (*bytes*) – `b'SERVICE:COMMAND'`
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or None; see *BaseTransport.bulk_read()* and *BaseTransport.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManager.read()*
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish

Returns
adb_info – Info and settings for this ADB transaction

Return type
_AdbTransactionInfo

_pull(*device_path*, *stream*, *progress_callback*, *adb_info*, *filesync_info*)

Pull a file from the device into the file-like `local_path`.

Parameters

- **device_path** (*str*) – The file on the device that will be pulled
- **stream** (*_io.BytesIO*) – File-like object for writing to
- **progress_callback** (*function, None*) – Callback method that accepts `device_path`, `bytes_written`, and `total_bytes`
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction

_push(*stream*, *device_path*, *st_mode*, *mtime*, *progress_callback*, *adb_info*, *filesync_info*)

Push a file-like object to the device.

Parameters

- **stream** (*_io.BytesIO*) – File-like object for reading from
- **device_path** (*str*) – Destination on the device to write to
- **st_mode** (*int*) – Stat mode for the file
- **mtime** (*int*) – Modification time
- **progress_callback** (*function, None*) – Callback method that accepts `device_path`, `bytes_written`, and `total_bytes`
- **adb_info** (*_AdbTransactionInfo*) – Info and settings for this ADB transaction

Raises

PushFailedError – Raised on push failure.

_read_until(*expected_cmds*, *adb_info*)

Read a packet, acknowledging any write packets.

1. Read data via *_AdbIOManager.read()*
2. If a `b'WRTE'` packet is received, send an `b'OKAY'` packet via *AdbDevice._okay()*
3. Return the `cmd` and `data` that were read by *_AdbIOManager.read()*

Parameters

- **expected_cmds** (*list[bytes]*) – *_AdbIOManager.read()* will look for a packet whose command is in `expected_cmds`
- **adb_info** (*_AdbTransactionInfo*) – Info and settings for this ADB transaction

Returns

- **cmd** (*bytes*) – The command that was received by *_AdbIOManager.read()*, which is in *adb_shell.constants.WIRE_TO_ID* and must be in `expected_cmds`
- **data** (*bytes*) – The data that was received by *_AdbIOManager.read()*

_read_until_close(*adb_info*)

Yield packets until a `b'CLSE'` packet is received.

1. Read the `cmd` and `data` fields from a `b'CLSE'` or `b'WRTE'` packet via *AdbDevice._read_until()*
2. If `cmd` is `b'CLSE'`, then send a `b'CLSE'` message and stop
3. Yield `data` and repeat

Parameters

adb_info (*_AdbTransactionInfo*) – Info and settings for this ADB transaction

Yields

data (*bytes*) – The data that was read by *AdbDevice._read_until()*

_service(*service*, *command*, *transport_timeout_s=None*, *read_timeout_s=10.0*, *timeout_s=None*, *decode=True*)

Send an ADB command to the device.

Parameters

- **service** (*bytes*) – The ADB service to talk to (e.g., `b'shell'`)
- **command** (*bytes*) – The command that will be sent
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransport.bulk_read()* and *BaseTransport.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManager.read()*
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish
- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Returns
The output of the ADB command as a string if `decode` is True, otherwise as bytes.

Return type
bytes, str

_streaming_command(*service*, *command*, *transport_timeout_s*, *read_timeout_s*, *timeout_s*)

One complete set of packets for a single command.

1. *_open()* a new connection to the device, where the `destination` parameter is `service:command`
2. Read the response data via *AdbDevice._read_until_close()*

Note: All the data is held in memory, and thus large responses will be slow and can fill up memory.

Parameters

- **service** (*bytes*) – The ADB service (e.g., `b'shell'`, as used by *AdbDevice.shell()*)
- **command** (*bytes*) – The service command
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransport.bulk_read()* and *BaseTransport.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManager.read()*
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish

Yields
bytes – The responses from the service.

_streaming_service(*service*, *command*, *transport_timeout_s=None*, *read_timeout_s=10.0*, *decode=True*)

Send an ADB command to the device, yielding each line of output.

Parameters

- **service** (*bytes*) – The ADB service to talk to (e.g., `b'shell'`)
- **command** (*bytes*) – The command that will be sent
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransport.bulk_read()* and *BaseTransport.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManager.read()*

- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Yields
bytes, str – The line-by-line output of the ADB command as a string if `decode` is True, otherwise as bytes.

`property available`

Whether or not an ADB connection to the device has been established.

Returns
`self._available`

Return type
bool

`close()`

Close the connection via the provided transport's `close()` method.

`connect`(*rsa_keys=None*, *transport_timeout_s=None*, *auth_timeout_s=10.0*, *read_timeout_s=10.0*, *auth_callback=None*)

Establish an ADB connection to the device.

See *_AdbIOManager.connect()*.

Parameters

- **rsa_keys** (*list, None*) – A list of signers of type *CryptographySigner*, *PycryptodomeAuthSigner*, or *PythonRSASigner*
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or None; see *BaseTransport.bulk_read()* and *BaseTransport.bulk_write()*
- **auth_timeout_s** (*float, None*) – The time in seconds to wait for a `b'CNXN'` authentication response
- **read_timeout_s** (*float*) – The total time in seconds to wait for expected commands in *_AdbIOManager._read_expected_packet_from_device()*
- **auth_callback** (*function, None*) – Function callback invoked when the connection needs to be accepted on the device

Returns
Whether the connection was established (*AdbDevice.available*)

Return type
bool

`exec_out`(*command*, *transport_timeout_s=None*, *read_timeout_s=10.0*, *timeout_s=None*, *decode=True*)

Send an ADB `exec-out` command to the device.

https://www.linux-magazine.com/Issues/2017/195/Ask-Klaus

Parameters

- **command** (*str*) – The exec-out command that will be sent
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or None; see *BaseTransport.bulk_read()* and *BaseTransport.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManager.read()*
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish

- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Returns
The output of the ADB exec-out command as a string if `decode` is True, otherwise as bytes.

Return type
bytes, str

list(*device_path*, *transport_timeout_s=None*, *read_timeout_s=10.0*)

Return a directory listing of the given path.

Parameters

- **device_path** (*str*) – Directory to list.
- **transport_timeout_s** (*float, None*) – Expected timeout for any part of the pull.
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManager.read()*

Returns
files – Filename, mode, size, and mtime info for the files in the directory

Return type
list[*DeviceFile*]

property max_chunk_size

Maximum chunk size for filesync operations

Returns
Minimum value based on *adb_shell.constants.MAX_CHUNK_SIZE* and `_max_data / 2`, fallback to legacy *adb_shell.constants.MAX_PUSH_DATA*

Return type
int

pull(*device_path*, *local_path*, *progress_callback=None*, *transport_timeout_s=None*, *read_timeout_s=10.0*)

Pull a file from the device.

Parameters

- **device_path** (*str*) – The file on the device that will be pulled
- **local_path** (*str, BytesIO*) – The path or BytesIO stream where the file will be downloaded
- **progress_callback** (*function, None*) – Callback method that accepts `device_path`, `bytes_written`, and `total_bytes`
- **transport_timeout_s** (*float, None*) – Expected timeout for any part of the pull.
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManager.read()*

push(*local_path*, *device_path*, *st_mode=33272*, *mtime=0*, *progress_callback=None*, *transport_timeout_s=None*, *read_timeout_s=10.0*)

Push a file or directory to the device.

Parameters

- **local_path** (*str, BytesIO*) – A filename, directory, or BytesIO stream to push to the device
- **device_path** (*str*) – Destination on the device to write to.

- **st_mode** (*int*) – Stat mode for `local_path`
- **mtime** (*int*) – Modification time to set on the file.
- **progress_callback** (*function, None*) – Callback method that accepts `device_path`, `bytes_written`, and `total_bytes`
- **transport_timeout_s** (*float, None*) – Expected timeout for any part of the push.
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManager.read()*

reboot(*fastboot=False*, *transport_timeout_s=None*, *read_timeout_s=10.0*, *timeout_s=None*)

Reboot the device.

Parameters

- **fastboot** (*bool*) – Whether to reboot the device into fastboot
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or None; see *BaseTransport.bulk_read()* and *BaseTransport.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManager.read()*
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish

root(*transport_timeout_s=None*, *read_timeout_s=10.0*, *timeout_s=None*)

Gain root access.

The device must be rooted in order for this to work.

Parameters

- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or None; see *BaseTransport.bulk_read()* and *BaseTransport.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManager.read()*
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish

shell(*command*, *transport_timeout_s=None*, *read_timeout_s=10.0*, *timeout_s=None*, *decode=True*)

Send an ADB shell command to the device.

Parameters

- **command** (*str*) – The shell command that will be sent
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or None; see *BaseTransport.bulk_read()* and *BaseTransport.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManager.read()*
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish
- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Returns

The output of the ADB shell command as a string if `decode` is True, otherwise as bytes.

Return type
bytes, str

`stat`(*device_path*, *transport_timeout_s=None*, *read_timeout_s=10.0*)

Get a file's `stat()` information.

Parameters

- **device_path** (*str*) – The file on the device for which we will get information.
- **transport_timeout_s** (*float, None*) – Expected timeout for any part of the pull.
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManager.read()*

Returns

- **mode** (*int*) – The octal permissions for the file
- **size** (*int*) – The size of the file
- **mtime** (*int*) – The last modified time for the file

`streaming_shell`(*command*, *transport_timeout_s=None*, *read_timeout_s=10.0*, *decode=True*)

Send an ADB shell command to the device, yielding each line of output.

Parameters

- **command** (*str*) – The shell command that will be sent
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransport.bulk_read()* and *BaseTransport.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManager.read()*
- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Yields
bytes, str – The line-by-line output of the ADB shell command as a string if `decode` is True, otherwise as bytes.

`class adb_shell.adb_device.`**AdbDeviceTcp**(*host*, *port=5555*, *default_transport_timeout_s=None*, *banner=None*)

Bases: *AdbDevice*

A class with methods for connecting to a device via TCP and executing ADB commands.

Parameters

- **host** (*str*) – The address of the device; may be an IP address or a host name
- **port** (*int*) – The device port to which we are connecting (default is 5555)
- **default_transport_timeout_s** (*float, None*) – Default timeout in seconds for TCP packets, or `None`
- **banner** (*str, bytes, None*) – The hostname of the machine where the Python interpreter is currently running; if it is not provided, it will be determined via `socket.gethostname()`

`_available`

Whether an ADB connection to the device has been established

Type
bool

_banner

The hostname of the machine where the Python interpreter is currently running

Type
bytearray, bytes

_default_transport_timeout_s

Default timeout in seconds for TCP packets, or `None`

Type
float, None

_local_id

The local ID that is used for ADB transactions; the value is incremented each time and is always in the range `[1, 2^32)`

Type
int

_maxdata

Maximum amount of data in an ADB packet

Type
int

_transport

The transport that is used to connect to the device

Type
TcpTransport

class adb_shell.adb_device.**AdbDeviceUsb**(*serial=None*, *port_path=None*, *default_transport_timeout_s=None*, *banner=None*)

Bases: *AdbDevice*

A class with methods for connecting to a device via USB and executing ADB commands.

Parameters

- **serial** (*str, None*) – The USB device serial ID
- **port_path** (*TODO, None*) – TODO
- **default_transport_timeout_s** (*float, None*) – Default timeout in seconds for USB packets, or `None`
- **banner** (*str, bytes, None*) – The hostname of the machine where the Python interpreter is currently running; if it is not provided, it will be determined via `socket.gethostname()`

Raises
adb_shell.exceptions.InvalidTransportError – Raised if package was not installed with the "usb" extras option (`pip install adb-shell[usb]`)

_available

Whether an ADB connection to the device has been established

Type
bool

_banner

The hostname of the machine where the Python interpreter is currently running

Type
bytearray, bytes

`_default_transport_timeout_s`

Default timeout in seconds for USB packets, or `None`

Type
float, None

`_local_id`

The local ID that is used for ADB transactions; the value is incremented each time and is always in the range `[1, 2^32)`

Type
int

`_maxdata`

Maximum amount of data in an ADB packet

Type
int

`_transport`

The transport that is used to connect to the device

Type
UsbTransport

`class adb_shell.adb_device._AdbIOManager(transport)`

Bases: `object`

A class for handling all ADB I/O.

Notes

When the `self._store_lock` and `self._transport_lock` locks are held at the same time, it must always be the case that the `self._transport_lock` is acquired first. This ensures that there is no potential for deadlock.

Parameters
transport (`BaseTransport`) – A transport for communicating with the device; must be an instance of a subclass of *BaseTransport*

`_packet_store`

A store for holding packets that correspond to different ADB streams

Type
_AdbPacketStore

`_store_lock`

A lock for protecting `self._packet_store` (this lock is never held for long)

Type
Lock

`_transport`

A transport for communicating with the device; must be an instance of a subclass of *BaseTransport*

Type
BaseTransport

`_transport_lock`

A lock for protecting `self._transport`

Type
Lock

`_read_bytes_from_device`(*length*, *adb_info*)

Read `length` bytes from the device.

Parameters

- **length** (*int*) – We will read packets until we get this length of data
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

Returns
The data that was read

Return type
bytes

Raises
adb_shell.exceptions.AdbTimeoutError – Did not read `length` bytes in time

`_read_expected_packet_from_device`(*expected_cmds*, *adb_info*)

Read packets from the device until we get an expected packet type.

Parameters

- **expected_cmds** (*list[bytes]*) – We will read packets until we encounter one whose "command" field is in `expected_cmds`
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

Returns

- **cmd** (*bytes*) – The received command, which is in *adb_shell.constants.WIRE_TO_ID* and must be in `expected_cmds`
- **arg0** (*int*) – TODO
- **arg1** (*int*) – TODO
- **data** (*bytes*) – The data that was read

Raises
adb_shell.exceptions.AdbTimeoutError – Never got one of the expected responses

`_read_packet_from_device`(*adb_info*)

Read a complete ADB packet (header + data) from the device.

Parameters
adb_info (_AdbTransactionInfo) – Info and settings for this ADB transaction

Returns

- **cmd** (*bytes*) – The received command, which is in *adb_shell.constants.WIRE_TO_ID* and must be in `expected_cmds`
- **arg0** (*int*) – TODO
- **arg1** (*int*) – TODO
- *bytes* – The data that was read

Raises

- *adb_shell.exceptions.InvalidCommandError* – Unknown command
- *adb_shell.exceptions.InvalidChecksumError* – Received checksum does not match the expected checksum

_send(*msg*, *adb_info*)

Send a message to the device.

1. Send the message header (`adb_shell.adb_message.AdbMessage.pack`)
2. Send the message data

Parameters

- **msg** (AdbMessage) – The data that will be sent
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

close()

Close the connection via the provided transport's `close()` method and clear the packet store.

connect(*banner*, *rsa_keys*, *auth_timeout_s*, *auth_callback*, *adb_info*)

Establish an ADB connection to the device.

1. Use the transport to establish a connection
2. Send a `b'CNXN'` message
3. Read the response from the device
4. If `cmd` is not `b'AUTH'`, then authentication is not necesary and so we are done
5. If no `rsa_keys` are provided, raise an exception
6. Loop through our keys, signing the last `banner2` that we received
   1. If the last `arg0` was not *adb_shell.constants.AUTH_TOKEN*, raise an exception
   2. Sign the last `banner2` and send it in an `b'AUTH'` message
   3. Read the response from the device
   4. If `cmd` is `b'CNXN'`, we are done
7. None of the keys worked, so send `rsa_keys[0]`'s public key; if the response does not time out, we must have connected successfully

Parameters

- **banner** (*bytearray, bytes*) – The hostname of the machine where the Python interpreter is currently running (*adb_shell.adb_device.AdbDevice._banner*)
- **rsa_keys** (*list, None*) – A list of signers of type *CryptographySigner*, *PycryptodomeAuthSigner*, or *PythonRSASigner*
- **auth_timeout_s** (*float, None*) – The time in seconds to wait for a `b'CNXN'` authentication response
- **auth_callback** (*function, None*) – Function callback invoked when the connection needs to be accepted on the device
- **adb_info** (_AdbTransactionInfo) – Info and settings for this connection attempt

Returns

- *bool* – Whether the connection was established

- **maxdata** (*int*) – Maximum amount of data in an ADB packet

Raises

- *adb_shell.exceptions.DeviceAuthError* – Device authentication required, no keys available
- *adb_shell.exceptions.InvalidResponseError* – Invalid auth response from the device

read(*expected_cmds*, *adb_info*, *allow_zeros=False*)

Read packets from the device until we get an expected packet type.

1. See if the expected packet is in the packet store
2. While the time limit has not been exceeded:
   1. See if the expected packet is in the packet store
   2. Read a packet from the device. If it matches what we are looking for, we are done. If it corresponds to a different stream, add it to the store.
3. Raise a timeout exception

Parameters

- **expected_cmds** (*list[bytes]*) – We will read packets until we encounter one whose "command" field is in `expected_cmds`
- **adb_info** (*_AdbTransactionInfo*) – Info and settings for this ADB transaction
- **allow_zeros** (*bool*) – Whether to allow the received `arg0` and `arg1` values to match with 0, in addition to `adb_info.remote_id` and `adb_info.local_id`, respectively

Returns

- **cmd** (*bytes*) – The received command, which is in *adb_shell.constants.WIRE_TO_ID* and must be in `expected_cmds`
- **arg0** (*int*) – TODO
- **arg1** (*int*) – TODO
- **data** (*bytes*) – The data that was read

Raises

adb_shell.exceptions.AdbTimeoutError – Never got one of the expected responses

send(*msg*, *adb_info*)

Send a message to the device.

Parameters

- **msg** (*AdbMessage*) – The data that will be sent
- **adb_info** (*_AdbTransactionInfo*) – Info and settings for this ADB transaction

adb_shell.adb_device.**_open_bytesio**(*stream*, **args*, ***kwargs*)

A context manager for a BytesIO object that does nothing.

Parameters

- **stream** (*BytesIO*) – The BytesIO stream
- **args** (*list*) – Unused positional arguments

- **kwargs** (*dict*) – Unused keyword arguments

Yields

stream (*BytesIO*) – The *stream* input parameter

adb_shell.adb_device_async module

Implement the *AdbDeviceAsync* class, which can connect to a device and run ADB shell commands.

  - *AdbDeviceAsync.close()*
  - *AdbDeviceAsync.connect()*
  - *AdbDeviceAsync.list()*
  - *AdbDeviceAsync.max_chunk_size*
  - *AdbDeviceAsync.pull()*
  - *AdbDeviceAsync.push()*
  - *AdbDeviceAsync.root()*
  - *AdbDeviceAsync.shell()*
  - *AdbDeviceAsync.stat()*
  - *AdbDeviceAsync.streaming_shell()*
- *AdbDeviceTcpAsync*

class adb_shell.adb_device_async.**AdbDeviceAsync**(*transport*, *default_transport_timeout_s=None*, *banner=None*)

Bases: `object`

A class with methods for connecting to a device and executing ADB commands.

Parameters

- **transport** (BaseTransportAsync) – A user-provided transport for communicating with the device; must be an instance of a subclass of *BaseTransportAsync*
- **default_transport_timeout_s** (*float, None*) – Default timeout in seconds for transport packets, or `None`
- **banner** (*str, bytes, None*) – The hostname of the machine where the Python interpreter is currently running; if it is not provided, it will be determined via `socket.gethostname()`

Raises

adb_shell.exceptions.InvalidTransportError – The passed `transport` is not an instance of a subclass of *BaseTransportAsync*

_available

Whether an ADB connection to the device has been established

Type
bool

_banner

The hostname of the machine where the Python interpreter is currently running

Type
bytearray, bytes

_default_transport_timeout_s

Default timeout in seconds for transport packets, or `None`

Type
float, None

_io_manager

Used for handling all ADB I/O

Type
_AdbIOManagerAsync

_local_id

The local ID that is used for ADB transactions; the value is incremented each time and is always in the range `[1, 2^32)`

Type
int

_local_id_lock

A lock for protecting `_local_id`; this is never held for long

Type
Lock

_maxdata

Maximum amount of data in an ADB packet

Type
int

_transport

The transport that is used to connect to the device; must be a subclass of *BaseTransportAsync*

Type
BaseTransportAsync

async _clse(*adb_info*)

Send a `b'CLSE'` message and then read a `b'CLSE'` message.

> **Warning:** This is not to be confused with the *AdbDeviceAsync.close()* method!

Parameters
adb_info (_AdbTransactionInfo) – Info and settings for this ADB transaction

async _filesync_flush(*adb_info*, *filesync_info*)

Write the data in the buffer up to `filesync_info.send_idx`, then set `filesync_info.send_idx` to 0.

Parameters

- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction

async _filesync_read(*expected_ids*, *adb_info*, *filesync_info*)

Read ADB messages and return FileSync packets.

Parameters

- **expected_ids** (*tuple[bytes]*) – If the received header ID is not in `expected_ids`, an exception will be raised
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction

Returns

- **command_id** (*bytes*) – The received header ID
- *tuple* – The contents of the header
- **data** (*bytearray, None*) – The received data, or `None` if the command ID is `adb_shell.constants.STAT`

Raises

- *adb_shell.exceptions.AdbCommandFailureException* – Command failed
- *adb_shell.exceptions.InvalidResponseError* – Received response was not in `expected_ids`

async _filesync_read_buffered(*size*, *adb_info*, *filesync_info*)

Read `size` bytes of data from `self.recv_buffer`.

Parameters

- **size** (*int*) – The amount of data to read
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction

Returns

result – The read data

Return type

bytearray

async _filesync_read_until(*expected_ids*, *finish_ids*, *adb_info*, *filesync_info*)

Useful wrapper around *AdbDeviceAsync._filesync_read()*.

Parameters

- **expected_ids** (*tuple[bytes]*) – If the received header ID is not in `expected_ids`, an exception will be raised
- **finish_ids** (*tuple[bytes]*) – We will read until we find a header ID that is in `finish_ids`
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction
- **filesync_info** (_FileSyncTransactionInfo) – Data and storage for this FileSync transaction

Yields

- **cmd_id** (*bytes*) – The received header ID
- **header** (*tuple*) – TODO
- **data** (*bytearray*) – The received data

async _filesync_send(*command_id*, *adb_info*, *filesync_info*, *data=b''*, *size=None*)

Send/buffer FileSync packets.

Packets are buffered and only flushed when this connection is read from. All messages have a response from the device, so this will always get flushed.

Parameters

- **command_id** (*bytes*) – Command to send.
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

- **filesync_info** (*_FileSyncTransactionInfo*) – Data and storage for this FileSync transaction
- **data** (*str, bytes*) – Optional data to send, must set data or size.
- **size** (*int, None*) – Optionally override size from len(data).

_get_transport_timeout_s(*transport_timeout_s*)

Use the provided `transport_timeout_s` if it is not `None`; otherwise, use `self._default_transport_timeout_s`

Parameters
transport_timeout_s (*float, None*) – The potential transport timeout

Returns
`transport_timeout_s` if it is not `None`; otherwise, `self._default_transport_timeout_s`

Return type
float

async _okay(*adb_info*)

Send an `b'OKAY'` mesage.

Parameters
adb_info (*_AdbTransactionInfo*) – Info and settings for this ADB transaction

async _open(*destination*, *transport_timeout_s*, *read_timeout_s*, *timeout_s*)

Opens a new connection to the device via an `b'OPEN'` message.

1. *send()* an `b'OPEN'` command to the device that specifies the `local_id`
2. *read()* the response from the device and fill in the `adb_info.remote_id` attribute

Parameters

- **destination** (*bytes*) – `b'SERVICE:COMMAND'`
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransportAsync.bulk_read()* and *BaseTransportAsync.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManagerAsync.read()*
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish

Returns
adb_info – Info and settings for this ADB transaction

Return type
_AdbTransactionInfo

async _pull(*device_path*, *stream*, *progress_callback*, *adb_info*, *filesync_info*)

Pull a file from the device into the file-like `local_path`.

Parameters

- **device_path** (*str*) – The file on the device that will be pulled
- **stream** (*AsyncBufferedIOBase, _AsyncBytesIO*) – File-like object for writing to

- **progress_callback** (*function, None*) – Callback method that accepts `device_path`, `bytes_written`, and `total_bytes`
- **adb_info** (*_AdbTransactionInfo*) – Info and settings for this ADB transaction
- **filesync_info** (*_FileSyncTransactionInfo*) – Data and storage for this FileSync transaction

async _push(*stream*, *device_path*, *st_mode*, *mtime*, *progress_callback*, *adb_info*, *filesync_info*)

Push a file-like object to the device.

Parameters

- **stream** (*AsyncBufferedReader, _AsyncBytesIO*) – File-like object for reading from
- **device_path** (*str*) – Destination on the device to write to
- **st_mode** (*int*) – Stat mode for the file
- **mtime** (*int*) – Modification time
- **progress_callback** (*function, None*) – Callback method that accepts `device_path`, `bytes_written`, and `total_bytes`
- **adb_info** (*_AdbTransactionInfo*) – Info and settings for this ADB transaction

Raises

PushFailedError – Raised on push failure.

async _read_until(*expected_cmds*, *adb_info*)

Read a packet, acknowledging any write packets.

1. Read data via *_AdbIOManagerAsync.read()*
2. If a `b'WRTE'` packet is received, send an `b'OKAY'` packet via *AdbDeviceAsync._okay()*
3. Return the `cmd` and `data` that were read by *_AdbIOManagerAsync.read()*

Parameters

- **expected_cmds** (*list[bytes]*) – *_AdbIOManagerAsync.read()* will look for a packet whose command is in `expected_cmds`
- **adb_info** (*_AdbTransactionInfo*) – Info and settings for this ADB transaction

Returns

- **cmd** (*bytes*) – The command that was received by *_AdbIOManagerAsync.read()*, which is in *adb_shell.constants.WIRE_TO_ID* and must be in `expected_cmds`
- **data** (*bytes*) – The data that was received by *_AdbIOManagerAsync.read()*

async _read_until_close(*adb_info*)

Yield packets until a `b'CLSE'` packet is received.

1. Read the `cmd` and `data` fields from a `b'CLSE'` or `b'WRTE'` packet via *AdbDeviceAsync._read_until()*
2. If `cmd` is `b'CLSE'`, then send a `b'CLSE'` message and stop
3. Yield `data` and repeat

Parameters

adb_info (*_AdbTransactionInfo*) – Info and settings for this ADB transaction

Yields
data (*bytes*) – The data that was read by *AdbDeviceAsync._read_until()*

async _service(*service*, *command*, *transport_timeout_s=None*, *read_timeout_s=10.0*, *timeout_s=None*, *decode=True*)

Send an ADB command to the device.

Parameters

- **service** (*bytes*) – The ADB service to talk to (e.g., `b'shell'`)
- **command** (*bytes*) – The command that will be sent
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransportAsync.bulk_read()* and *BaseTransportAsync.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManagerAsync.read()*
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish
- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Returns
The output of the ADB command as a string if `decode` is True, otherwise as bytes.

Return type
bytes, str

async _streaming_command(*service*, *command*, *transport_timeout_s*, *read_timeout_s*, *timeout_s*)

One complete set of packets for a single command.

1. *_open()* a new connection to the device, where the `destination` parameter is `service:command`
2. Read the response data via *AdbDeviceAsync._read_until_close()*

Note: All the data is held in memory, and thus large responses will be slow and can fill up memory.

Parameters

- **service** (*bytes*) – The ADB service (e.g., `b'shell'`, as used by *AdbDeviceAsync.shell()*)
- **command** (*bytes*) – The service command
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransportAsync.bulk_read()* and *BaseTransportAsync.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManagerAsync.read()*
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish

Yields
bytes – The responses from the service.

async _streaming_service(*service*, *command*, *transport_timeout_s=None*, *read_timeout_s=10.0*, *decode=True*)

Send an ADB command to the device, yielding each line of output.

Parameters

- **service** (*bytes*) – The ADB service to talk to (e.g., `b'shell'`)
- **command** (*bytes*) – The command that will be sent
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransportAsync.bulk_read()* and *BaseTransportAsync.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManagerAsync.read()*
- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Yields

bytes, str – The line-by-line output of the ADB command as a string if `decode` is True, otherwise as bytes.

property available

Whether or not an ADB connection to the device has been established.

Returns

`self._available`

Return type

bool

async close()

Close the connection via the provided transport's `close()` method.

async connect(*rsa_keys=None*, *transport_timeout_s=None*, *auth_timeout_s=10.0*, *read_timeout_s=10.0*, *auth_callback=None*)

Establish an ADB connection to the device.

See *_AdbIOManagerAsync.connect()*.

Parameters

- **rsa_keys** (*list, None*) – A list of signers of type *CryptographySigner*, *PycryptodomeAuthSigner*, or *PythonRSASigner*
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransportAsync.bulk_read()* and *BaseTransportAsync.bulk_write()*
- **auth_timeout_s** (*float, None*) – The time in seconds to wait for a `b'CNXN'` authentication response
- **read_timeout_s** (*float*) – The total time in seconds to wait for expected commands in *_AdbIOManagerAsync._read_expected_packet_from_device()*
- **auth_callback** (*function, None*) – Function callback invoked when the connection needs to be accepted on the device

Returns

Whether the connection was established (*AdbDeviceAsync.available*)

Return type
bool

async exec_out(*command*, *transport_timeout_s=None*, *read_timeout_s=10.0*, *timeout_s=None*, *decode=True*)

Send an ADB `exec-out` command to the device.

https://www.linux-magazine.com/Issues/2017/195/Ask-Klaus

Parameters

- **command** (*str*) – The exec-out command that will be sent
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransportAsync.bulk_read()* and *BaseTransportAsync.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManagerAsync.read()*
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish
- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Returns
The output of the ADB exec-out command as a string if `decode` is True, otherwise as bytes.

Return type
bytes, str

async list(*device_path*, *transport_timeout_s=None*, *read_timeout_s=10.0*)

Return a directory listing of the given path.

Parameters

- **device_path** (*str*) – Directory to list.
- **transport_timeout_s** (*float, None*) – Expected timeout for any part of the pull.
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManagerAsync.read()*

Returns
files – Filename, mode, size, and mtime info for the files in the directory

Return type
list[*DeviceFile*]

property max_chunk_size

Maximum chunk size for filesync operations

Returns
Minimum value based on *adb_shell.constants.MAX_CHUNK_SIZE* and `_max_data / 2`, fallback to legacy *adb_shell.constants.MAX_PUSH_DATA*

Return type
int

async pull(*device_path*, *local_path*, *progress_callback=None*, *transport_timeout_s=None*, *read_timeout_s=10.0*)

Pull a file from the device.

Parameters

- **device_path** (*str*) – The file on the device that will be pulled
- **local_path** (*str, BytesIO*) – The path or BytesIO stream where the file will be downloaded
- **progress_callback** (*function, None*) – Callback method that accepts `device_path`, `bytes_written`, and `total_bytes`
- **transport_timeout_s** (*float, None*) – Expected timeout for any part of the pull.
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManagerAsync.read()*

async push(*local_path*, *device_path*, *st_mode=33272*, *mtime=0*, *progress_callback=None*, *transport_timeout_s=None*, *read_timeout_s=10.0*)

Push a file or directory to the device.

Parameters

- **local_path** (*str, BytesIO*) – A filename, directory, or BytesIO stream to push to the device
- **device_path** (*str*) – Destination on the device to write to
- **st_mode** (*int*) – Stat mode for `local_path`
- **mtime** (*int*) – Modification time to set on the file
- **progress_callback** (*function, None*) – Callback method that accepts `device_path`, `bytes_written`, and `total_bytes`
- **transport_timeout_s** (*float, None*) – Expected timeout for any part of the push
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManagerAsync.read()*

async reboot(*fastboot=False*, *transport_timeout_s=None*, *read_timeout_s=10.0*, *timeout_s=None*)

Reboot the device.

Parameters

- **fastboot** (*bool*) – Whether to reboot the device into fastboot
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransportAsync.bulk_read()* and *BaseTransportAsync.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in `_AdbIOManager.read()`
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish

async root(*transport_timeout_s=None*, *read_timeout_s=10.0*, *timeout_s=None*)

Gain root access.

The device must be rooted in order for this to work.

Parameters

- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransportAsync.bulk_read()* and *BaseTransportAsync.bulk_write()*

- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManagerAsync.read()*
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish

async shell(*command*, *transport_timeout_s=None*, *read_timeout_s=10.0*, *timeout_s=None*, *decode=True*)

Send an ADB shell command to the device.

Parameters

- **command** (*str*) – The shell command that will be sent
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransportAsync.bulk_read()* and *BaseTransportAsync.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManagerAsync.read()*
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish
- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Returns

The output of the ADB shell command as a string if `decode` is True, otherwise as bytes.

Return type

bytes, str

async stat(*device_path*, *transport_timeout_s=None*, *read_timeout_s=10.0*)

Get a file's `stat()` information.

Parameters

- **device_path** (*str*) – The file on the device for which we will get information.
- **transport_timeout_s** (*float, None*) – Expected timeout for any part of the pull.
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManagerAsync.read()*

Returns

- **mode** (*int*) – The octal permissions for the file
- **size** (*int*) – The size of the file
- **mtime** (*int*) – The last modified time for the file

async streaming_shell(*command*, *transport_timeout_s=None*, *read_timeout_s=10.0*, *decode=True*)

Send an ADB shell command to the device, yielding each line of output.

Parameters

- **command** (*str*) – The shell command that will be sent
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransportAsync.bulk_read()* and *BaseTransportAsync.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for a `b'CLSE'` or `b'OKAY'` command in *_AdbIOManagerAsync.read()*
- **decode** (*bool*) – Whether to decode the output to utf8 before returning

Yields
bytes, str – The line-by-line output of the ADB shell command as a string if `decode` is True, otherwise as bytes.

`class` adb_shell.adb_device_async.**AdbDeviceTcpAsync**(*host*, *port=5555*, *default_transport_timeout_s=None*, *banner=None*)

Bases: *AdbDeviceAsync*

A class with methods for connecting to a device via TCP and executing ADB commands.

Parameters

- **host** (*str*) – The address of the device; may be an IP address or a host name
- **port** (*int*) – The device port to which we are connecting (default is 5555)
- **default_transport_timeout_s** (*float, None*) – Default timeout in seconds for TCP packets, or `None`
- **banner** (*str, bytes, None*) – The hostname of the machine where the Python interpreter is currently running; if it is not provided, it will be determined via `socket.gethostname()`

_available

Whether an ADB connection to the device has been established

Type
bool

_banner

The hostname of the machine where the Python interpreter is currently running

Type
bytearray, bytes

_default_transport_timeout_s

Default timeout in seconds for TCP packets, or `None`

Type
float, None

_local_id

The local ID that is used for ADB transactions; the value is incremented each time and is always in the range `[1, 2^32)`

Type
int

_maxdata

Maximum amount of data in an ADB packet

Type
int

_transport

The transport that is used to connect to the device

Type
TcpTransportAsync

class adb_shell.adb_device_async._AdbIOManagerAsync(*transport*)

Bases: object

A class for handling all ADB I/O.

Notes

When the self._store_lock and self._transport_lock locks are held at the same time, it must always be the case that the self._transport_lock is acquired first. This ensures that there is no potential for deadlock.

Parameters
transport (BaseTransportAsync) – A transport for communicating with the device; must be an instance of a subclass of *BaseTransportAsync*

_packet_store

A store for holding packets that correspond to different ADB streams

Type
_AdbPacketStore

_store_lock

A lock for protecting self._packet_store (this lock is never held for long)

Type
Lock

_transport

A transport for communicating with the device; must be an instance of a subclass of *BaseTransportAsync*

Type
BaseTransportAsync

_transport_lock

A lock for protecting self._transport

Type
Lock

async _read_bytes_from_device(*length*, *adb_info*)

Read length bytes from the device.

Parameters

- **length** (*int*) – We will read packets until we get this length of data
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

Returns
The data that was read

Return type
bytes

Raises
adb_shell.exceptions.AdbTimeoutError – Did not read length bytes in time

async _read_expected_packet_from_device(*expected_cmds*, *adb_info*)

Read packets from the device until we get an expected packet type.

Parameters

- **expected_cmds** (*list[bytes]*) – We will read packets until we encounter one whose "command" field is in `expected_cmds`
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

Returns

- **cmd** (*bytes*) – The received command, which is in *adb_shell.constants.WIRE_TO_ID* and must be in `expected_cmds`
- **arg0** (*int*) – TODO
- **arg1** (*int*) – TODO
- **data** (*bytes*) – The data that was read

Raises

adb_shell.exceptions.AdbTimeoutError – Never got one of the expected responses

async _read_packet_from_device(*adb_info*)

Read a complete ADB packet (header + data) from the device.

Parameters

adb_info (_AdbTransactionInfo) – Info and settings for this ADB transaction

Returns

- **cmd** (*bytes*) – The received command, which is in *adb_shell.constants.WIRE_TO_ID* and must be in `expected_cmds`
- **arg0** (*int*) – TODO
- **arg1** (*int*) – TODO
- *bytes* – The data that was read

Raises

- *adb_shell.exceptions.InvalidCommandError* – Unknown command
- *adb_shell.exceptions.InvalidChecksumError* – Received checksum does not match the expected checksum

async _send(*msg*, *adb_info*)

Send a message to the device.

1. Send the message header (`adb_shell.adb_message.AdbMessage.pack`)
2. Send the message data

Parameters

- **msg** (AdbMessage) – The data that will be sent
- **adb_info** (_AdbTransactionInfo) – Info and settings for this ADB transaction

async close()

Close the connection via the provided transport's `close()` method and clear the packet store.

async connect(*banner*, *rsa_keys*, *auth_timeout_s*, *auth_callback*, *adb_info*)

Establish an ADB connection to the device.

1. Use the transport to establish a connection
2. Send a `b'CNXN'` message

3. Read the response from the device
4. If `cmd` is not `b'AUTH'`, then authentication is not necesary and so we are done
5. If no `rsa_keys` are provided, raise an exception
6. Loop through our keys, signing the last `banner2` that we received
    1. If the last `arg0` was not *adb_shell.constants.AUTH_TOKEN*, raise an exception
    2. Sign the last `banner2` and send it in an `b'AUTH'` message
    3. Read the response from the device
    4. If `cmd` is `b'CNXN'`, we are done
7. None of the keys worked, so send `rsa_keys[0]`'s public key; if the response does not time out, we must have connected successfully

Parameters

- **banner** (*bytearray, bytes*) – The hostname of the machine where the Python interpreter is currently running (*adb_shell.adb_device_async.AdbDeviceAsync._banner*)
- **rsa_keys** (*list, None*) – A list of signers of type *CryptographySigner*, *PycryptodomeAuthSigner*, or *PythonRSASigner*
- **auth_timeout_s** (*float, None*) – The time in seconds to wait for a `b'CNXN'` authentication response
- **auth_callback** (*function, None*) – Function callback invoked when the connection needs to be accepted on the device
- **adb_info** (*_AdbTransactionInfo*) – Info and settings for this connection attempt

Returns

- *bool* – Whether the connection was established
- **maxdata** (*int*) – Maximum amount of data in an ADB packet

Raises

- ***adb_shell.exceptions.DeviceAuthError*** – Device authentication required, no keys available
- ***adb_shell.exceptions.InvalidResponseError*** – Invalid auth response from the device

async read(*expected_cmds*, *adb_info*, *allow_zeros=False*)

Read packets from the device until we get an expected packet type.

1. See if the expected packet is in the packet store
2. While the time limit has not been exceeded:
    1. See if the expected packet is in the packet store
    2. Read a packet from the device. If it matches what we are looking for, we are done. If it corresponds to a different stream, add it to the store.
3. Raise a timeout exception

Parameters

- **expected_cmds** (*list[bytes]*) – We will read packets until we encounter one whose "command" field is in `expected_cmds`
- **adb_info** (*_AdbTransactionInfo*) – Info and settings for this ADB transaction
- **allow_zeros** (*bool*) – Whether to allow the received `arg0` and `arg1` values to match with 0, in addition to `adb_info.remote_id` and `adb_info.local_id`, respectively

Returns

- **cmd** (*bytes*) – The received command, which is in *adb_shell.constants.WIRE_TO_ID* and must be in `expected_cmds`
- **arg0** (*int*) – TODO
- **arg1** (*int*) – TODO
- **data** (*bytes*) – The data that was read

Raises

adb_shell.exceptions.AdbTimeoutError – Never got one of the expected responses

async send(*msg*, *adb_info*)

Send a message to the device.

Parameters

- **msg** (*AdbMessage*) – The data that will be sent
- **adb_info** (*_AdbTransactionInfo*) – Info and settings for this ADB transaction

class adb_shell.adb_device_async.**_AsyncBytesIO**(*bytesio*)

Bases: `object`

An async wrapper for *BytesIO*.

Parameters

bytesio (*BytesIO*) – The BytesIO object that is wrapped

async read(*size=-1*)

Read data.

Parameters

size (*int*) – The size of the data to be read

Returns

The data that was read

Return type

bytes

async write(*data*)

Write data.

Parameters

data (*bytes*) – The data to be written

adb_shell.adb_device_async.**_open_bytesio**(*stream*, **args*, ***kwargs*)

An async context manager for a BytesIO object that does nothing.

Parameters

- **stream** (*BytesIO*) – The BytesIO stream
- **args** (*list*) – Unused positional arguments

- **kwargs** (*dict*) – Unused keyword arguments

Yields

_AsyncBytesIO – The wrapped *stream* input parameter

adb_shell.adb_message module

Functions and an *AdbMessage* class for packing and unpacking ADB messages.

Contents

class adb_shell.adb_message.**AdbMessage**(*command*, *arg0*, *arg1*, *data=b''*)

Bases: `object`

A helper class for packing ADB messages.

Parameters

- **command** (*bytes*) – A command; examples used in this package include `adb_shell.constants.AUTH`, `adb_shell.constants.CNXN`, `adb_shell.constants.CLSE`, `adb_shell.constants.OPEN`, and `adb_shell.constants.OKAY`
- **arg0** (*int*) – Usually the local ID, but *connect()* and *connect()* provide *adb_shell.constants.VERSION*, *adb_shell.constants.AUTH_SIGNATURE*, and *adb_shell.constants.AUTH_RSAPUBLICKEY*
- **arg1** (*int*) – Usually the remote ID, but *connect()* and *connect()* provide *adb_shell.constants.MAX_ADB_DATA*
- **data** (*bytes*) – The data that will be sent

arg0

Usually the local ID, but *connect()* and *connect()* provide *adb_shell.constants.VERSION*, *adb_shell.constants.AUTH_SIGNATURE*, and *adb_shell.constants.AUTH_RSAPUBLICKEY*

Type
int

arg1

Usually the remote ID, but *connect()* and *connect()* provide *adb_shell.constants.MAX_ADB_DATA*

Type
int

command

The input parameter `command` converted to an integer via *adb_shell.constants.ID_TO_WIRE*

Type
int

data

The data that will be sent

Type
bytes

magic

`self.command` with its bits flipped; in other words, `self.command + self.magic == 2**32 - 1`

Type
int

property checksum

Return `checksum(self.data)`

Returns
The checksum of `self.data`

Return type
int

pack()

Returns this message in an over-the-wire format.

Returns
The message packed into the format required by ADB

Return type
bytes

adb_shell.adb_message.**checksum**(*data*)

Calculate the checksum of the provided data.

Parameters
data (*bytearray, bytes, str*) – The data

Returns
The checksum

Return type
int

adb_shell.adb_message.**int_to_cmd**(*n*)

Convert from an integer (4 bytes) to an ADB command.

Parameters
n (*int*) – The integer that will be converted to an ADB command

Returns
The ADB command (e.g., `'CNXN'`)

Return type
str

adb_shell.adb_message.**unpack**(*message*)

Unpack a received ADB message.

Parameters
message (*bytes*) – The received message

Returns

- **cmd** (*int*) – The ADB command

- **arg0** (*int*) – TODO
- **arg1** (*int*) – TODO
- **data_length** (*int*) – The length of the message's data
- **data_checksum** (*int*) – The checksum of the message's data

Raises

ValueError – Unable to unpack the ADB command.

adb_shell.constants module

Constants used throughout the code.

adb_shell.constants.**AUTH_RSAPUBLICKEY = 3**

AUTH constant for `arg0`

adb_shell.constants.**AUTH_SIGNATURE = 2**

AUTH constant for `arg0`

adb_shell.constants.**AUTH_TOKEN = 1**

AUTH constant for `arg0`

adb_shell.constants.**CLASS = 255**

From adb.h

adb_shell.constants.**DEFAULT_AUTH_TIMEOUT_S = 10.0**

Default authentication timeout (in s) for *adb_shell.adb_device.AdbDevice.connect()* and *adb_shell.adb_device_async.AdbDeviceAsync.connect()*

adb_shell.constants.**DEFAULT_PUSH_MODE = 33272**

Default mode for pushed files.

adb_shell.constants.**DEFAULT_READ_TIMEOUT_S = 10.0**

Default total timeout (in s) for reading data from the device

adb_shell.constants.**FILESYNC_IDS = (b'DATA', b'DENT', b'DONE', b'FAIL', b'LIST', b'OKAY', b'QUIT', b'RECV', b'SEND', b'STAT')**

Commands that are recognized by *adb_shell.adb_device.AdbDevice._filesync_read()* and *adb_shell.adb_device_async.AdbDeviceAsync._filesync_read()*

adb_shell.constants.**FILESYNC_ID_TO_WIRE = {b'DATA': 1096040772, b'DENT': 1414415684, b'DONE': 1162760004, b'FAIL': 1279869254, b'LIST': 1414744396, b'OKAY': 1497451343, b'QUIT': 1414092113, b'RECV': 1447249234, b'SEND': 1145980243, b'STAT': 1413567571}**

A dictionary where the keys are the commands in *FILESYNC_IDS* and the values are the keys converted to integers

adb_shell.constants.**FILESYNC_LIST_FORMAT = b'<5I'**

The format for FileSync "list" messages

adb_shell.constants.**FILESYNC_PULL_FORMAT = b'<2I'**

The format for FileSync "pull" messages

adb_shell.constants.**FILESYNC_PUSH_FORMAT = b'<2I'**

The format for FileSync "push" messages

adb_shell.constants.**FILESYNC_STAT_FORMAT = b'<4I'**

The format for FileSync "stat" messages

adb_shell.constants.**FILESYNC_WIRE_TO_ID = {1096040772: b'DATA', 1145980243: b'SEND', 1162760004: b'DONE', 1279869254: b'FAIL', 1413567571: b'STAT', 1414092113: b'QUIT', 1414415684: b'DENT', 1414744396: b'LIST', 1447249234: b'RECV', 1497451343: b'OKAY'}**

A dictionary where the keys are integers and the values are their corresponding commands (type = bytes) from *FILESYNC_IDS*

adb_shell.constants.**IDS = (b'AUTH', b'CLSE', b'CNXN', b'OKAY', b'OPEN', b'SYNC', b'WRTE')**

Commands that are recognized by *adb_shell.adb_device._AdbIOManager._read_packet_from_device()* and *adb_shell.adb_device_async._AdbIOManagerAsync._read_packet_from_device()*

adb_shell.constants.**ID_TO_WIRE = {b'AUTH': 1213486401, b'CLSE': 1163086915, b'CNXN': 1314410051, b'OKAY': 1497451343, b'OPEN': 1313165391, b'SYNC': 1129208147, b'WRTE': 1163154007}**

A dictionary where the keys are the commands in *IDS* and the values are the keys converted to integers

adb_shell.constants.**MAX_ADB_DATA = 1048576**

//android.googlesource.com/platform/system/core/+/master/adb/adb.h

Type
Maximum amount of data in an ADB packet. According to

Type
https

adb_shell.constants.**MAX_CHUNK_SIZE = 65536**

//android.googlesource.com/platform/system/core/+/master/adb/SYNC.TXT

Type
Maximum chunk size. According to https

adb_shell.constants.**MAX_PUSH_DATA = 2048**

Maximum size of a filesync DATA packet. Default size.

adb_shell.constants.**MESSAGE_FORMAT = b'<6I'**

An ADB message is 6 words in little-endian.

adb_shell.constants.**MESSAGE_SIZE = 24**

The size of an ADB message

adb_shell.constants.**PROTOCOL = 1**

From adb.h

adb_shell.constants.**SUBCLASS = 66**

From adb.h

adb_shell.constants.**VERSION = 16777216**

ADB protocol version.

adb_shell.constants.**WIRE_TO_ID = {1129208147: b'SYNC', 1163086915: b'CLSE', 1163154007: b'WRTE', 1213486401: b'AUTH', 1313165391: b'OPEN', 1314410051: b'CNXN', 1497451343: b'OKAY'}**

A dictionary where the keys are integers and the values are their corresponding commands (type = bytes) from *IDS*

adb_shell.exceptions module

ADB-related exceptions.

exception adb_shell.exceptions.**AdbCommandFailureException**

Bases: Exception

A `b'FAIL'` packet was received.

exception adb_shell.exceptions.**AdbConnectionError**

Bases: Exception

ADB command not sent because a connection to the device has not been established.

exception adb_shell.exceptions.**AdbTimeoutError**

Bases: Exception

ADB command did not complete within the specified time.

exception adb_shell.exceptions.**DeviceAuthError**(*message*, **args*)

Bases: Exception

Device authentication failed.

exception adb_shell.exceptions.**DevicePathInvalidError**

Bases: Exception

A file command was passed an invalid path.

exception adb_shell.exceptions.**InvalidChecksumError**

Bases: Exception

Checksum of data didn't match expected checksum.

exception adb_shell.exceptions.**InvalidCommandError**

Bases: Exception

Got an invalid command.

exception adb_shell.exceptions.**InvalidResponseError**

Bases: Exception

Got an invalid response to our command.

exception adb_shell.exceptions.**InvalidTransportError**

Bases: Exception

The provided transport does not implement the necessary methods: `close`, `connect`, `bulk_read`, and `bulk_write`.

exception adb_shell.exceptions.**PushFailedError**

Bases: Exception

Pushing a file failed for some reason.

exception adb_shell.exceptions.**TcpTimeoutException**

Bases: Exception

TCP connection timed read/write operation exceeded the allowed time.

exception adb_shell.exceptions.**UsbDeviceNotFoundError**

Bases: Exception

TODO

exception adb_shell.exceptions.**UsbReadFailedError**(*msg*, *usb_error*)

Bases: Exception

TODO

Parameters

- **msg** (*str*) – The error message
- **usb_error** (*libusb1.USBError*) – An exception from libusb1

usb_error

An exception from libusb1

Type
libusb1.USBError

exception adb_shell.exceptions.**UsbWriteFailedError**

Bases: Exception

adb_shell.transport.usb_transport.UsbTransport.bulk_write() failed.

adb_shell.hidden_helpers module

Implement helpers for the *AdbDevice* and *AdbDeviceAsync* classes.

Contents

class adb_shell.hidden_helpers.**DeviceFile**(*filename*, *mode*, *size*, *mtime*)

Bases: tuple

_asdict()

Return a new dict which maps field names to their values.

_field_defaults = {}

_fields = ('filename', 'mode', 'size', 'mtime')

classmethod _make(*iterable*)

Make a new DeviceFile object from a sequence or iterable

_replace(***kwds*)

Return a new DeviceFile object replacing specified fields with new values

filename

Alias for field number 0

mode

Alias for field number 1

mtime

Alias for field number 3

size

Alias for field number 2

class adb_shell.hidden_helpers.**_AdbPacketStore**

Bases: object

A class for storing ADB packets.

This class is used to support multiple streams.

_dict

A dictionary of dictionaries of queues. The first (outer) dictionary keys are the `arg1` return values from the *adb_shell.adb_device._AdbIOManager._read_packet_from_device()* and *adb_shell.adb_device_async._AdbIOManagerAsync._read_packet_from_device()* methods. The second (inner) dictionary keys are the `arg0` return values from those methods. And the values of this inner dictionary are queues of `(cmd, data)` tuples.

Type
dict[int: dict[int: Queue]]

clear(*arg0*, *arg1*)

Delete the entry for `(arg0, arg1)`, if it exists.

Parameters

- **arg0** (*int*) – The `arg0` return value from the *adb_shell.adb_device._AdbIOManager._read_packet_from_device()* and *adb_shell.adb_device_async._AdbIOManagerAsync._read_packet_from_device()* methods
- **arg1** (*int*) – The `arg1` return value from the *adb_shell.adb_device._AdbIOManager._read_packet_from_device()* and *adb_shell.adb_device_async._AdbIOManagerAsync._read_packet_from_device()* methods

`clear_all`()

Clear all the entries.

`find`(*arg0*, *arg1*)

Find the entry corresponding to `arg0` and `arg1`.

Parameters

- **arg0** (*int, None*) – The `arg0` value that we are looking for; `None` serves as a wildcard
- **arg1** (*int, None*) – The `arg1` value that we are looking for; `None` serves as a wildcard

Returns

The `(arg0, arg1)` pair that was found in the dictionary of dictionaries, or `None` if no match was found

Return type

tuple[int, int], None

`find_allow_zeros`(*arg0*, *arg1*)

Find the entry corresponding to (`arg0` or 0) and (`arg1` or 0).

Parameters

- **arg0** (*int, None*) – The `arg0` value that we are looking for; `None` serves as a wildcard
- **arg1** (*int, None*) – The `arg1` value that we are looking for; `None` serves as a wildcard

Returns

The first matching `(arg0, arg1)` pair that was found in the dictionary of dictionaries, or `None` if no match was found

Return type

tuple[int, int], None

`get`(*arg0*, *arg1*)

Get the next entry from the queue for `arg0` and `arg1`.

This function assumes you have already checked that `(arg0, arg1) in self`.

Parameters

- **arg0** (*int, None*) – The `arg0` return value from the *adb_shell.adb_device._AdbIOManager._read_packet_from_device()* and *adb_shell.adb_device_async._AdbIOManagerAsync._read_packet_from_device()* methods; `None` serves as a wildcard
- **arg1** (*int, None*) – The `arg1` return value from the *adb_shell.adb_device._AdbIOManager._read_packet_from_device()* and *adb_shell.adb_device_async._AdbIOManagerAsync._read_packet_from_device()* methods; `None` serves as a wildcard

Returns

- **cmd** (*bytes*) – The ADB packet's command
- **arg0** (*int*) – The `arg0` value from the returned packet
- **arg1** (*int*) – The `arg1` value from the returned packet
- **data** (*bytes*) – The ADB packet's data

put(*arg0*, *arg1*, *cmd*, *data*)

Add an entry to the queue for `arg0` and `arg1`.

Note that a new dictionary entry will not be created if `cmd == constants.CLSE`.

Parameters

- **arg0** (*int*) – The `arg0` return value from the *adb_shell.adb_device._AdbIOManager._read_packet_from_device()* and *adb_shell.adb_device_async._AdbIOManagerAsync._read_packet_from_device()* methods
- **arg1** (*int*) – The `arg1` return value from the *adb_shell.adb_device._AdbIOManager._read_packet_from_device()* and *adb_shell.adb_device_async._AdbIOManagerAsync._read_packet_from_device()* methods
- **cmd** (*bytes*) – The ADB packet's command
- **data** (*bytes*) – The ADB packet's data

class adb_shell.hidden_helpers.**_AdbTransactionInfo**(*local_id*, *remote_id*, *transport_timeout_s=None*, *read_timeout_s=10.0*, *timeout_s=None*)

Bases: `object`

A class for storing info and settings used during a single ADB "transaction."

Note that if `timeout_s` is not `None`, then:

```
self.transport_timeout_s <= self.read_timeout_s <= self.timeout_s
```

If `timeout_s` is `None`, the first inequality still applies.

Parameters

- **local_id** (*int*) – The ID for the sender (i.e., the device running this code)
- **remote_id** (*int*) – The ID for the recipient
- **transport_timeout_s** (*float, None*) – Timeout in seconds for sending and receiving data, or `None`; see *BaseTransport.bulk_read()*, *BaseTransport.bulk_write()*, *BaseTransportAsync.bulk_read()*, and *BaseTransportAsync.bulk_write()*
- **read_timeout_s** (*float*) – The total time in seconds to wait for data and packets from the device
- **timeout_s** (*float, None*) – The total time in seconds to wait for the ADB command to finish

local_id

The ID for the sender (i.e., the device running this code)

Type
int

read_timeout_s

The total time in seconds to wait for data and packets from the device

Type
float

remote_id

The ID for the recipient

Type
int

timeout_s

The total time in seconds to wait for the ADB command to finish

Type
float, None

transport_timeout_s

Timeout in seconds for sending and receiving data, or None; see *BaseTransport.bulk_read()*, *BaseTransport.bulk_write()*, *BaseTransportAsync.bulk_read()*, and *BaseTransportAsync.bulk_write()*

Type
float, None

args_match(*arg0*, *arg1*, *allow_zeros=False*)

Check if `arg0` and `arg1` match this object's `remote_id` and `local_id` attributes, respectively.

Parameters

- **arg0** (*int*) – The `arg0` value from an ADB packet, which will be compared to this object's `remote_id` attribute
- **arg1** (*int*) – The `arg1` value from an ADB packet, which will be compared to this object's `local_id` attribute
- **allow_zeros** (*bool*) – Whether to check if `arg0` and `arg1` match 0, in addition to this object's `local_id` and `remote_id` attributes

Returns
Whether `arg0` and `arg1` match this object's `local_id` and `remote_id` attributes

Return type
bool

class adb_shell.hidden_helpers.**_FileSyncTransactionInfo**(*recv_message_format*, *maxdata=1048576*)

Bases: `object`

A class for storing info used during a single FileSync "transaction."

Parameters

- **recv_message_format** (*bytes*) – The FileSync message format
- **maxdata** (*int*) – Maximum amount of data in an ADB packet

_maxdata

Maximum amount of data in an ADB packet

Type
int

recv_buffer

A buffer for storing received data

Type
bytearray

recv_message_format

The FileSync message format

Type
bytes

recv_message_size

The FileSync message size

Type
int

send_buffer

A buffer for storing data to be sent

Type
bytearray

send_idx

The index in `recv_buffer` that will be the start of the next data packet sent

Type
int

can_add_to_send_buffer(*data_len*)

Determine whether `data_len` bytes of data can be added to the send buffer without exceeding `constants.MAX_ADB_DATA`.

Parameters
data_len (*int*) – The length of the data to be potentially added to the send buffer (not including the length of its header)

Returns
Whether `data_len` bytes of data can be added to the send buffer without exceeding `constants.MAX_ADB_DATA`

Return type
bool

adb_shell.hidden_helpers.**get_banner**()

Get the `banner` that will be signed in *adb_shell.adb_device.AdbDevice.connect()* / *adb_shell.adb_device_async.AdbDeviceAsync.connect()*.

Returns
The hostname, or "unknown" if it could not be determined

Return type
bytearray

adb_shell.hidden_helpers.**get_files_to_push**(*local_path*, *device_path*)

Get a list of the file(s) to push.

Parameters

- **local_path** (*str*) – A path to a local file or directory
- **device_path** (*str*) – A path to a file or directory on the device

Returns

- **local_path_is_dir** (*bool*) – Whether or not `local_path` is a directory
- **local_paths** (*list[str]*) – A list of the file(s) to push

- **device_paths** (*list[str]*) – A list of destination paths on the device that corresponds to `local_paths`

1.1.3 Module contents

ADB shell functionality.

Prebuilt wheel can be downloaded from nightly.link.

This Python package implements ADB shell and FileSync functionality. It originated from python-adb.

CHAPTER

TWO

INSTALLATION

```
pip install adb-shell
```

2.1 Async

To utilize the async version of this code, you must install into a Python 3.7+ environment via:

```
pip install adb-shell[async]
```

2.2 USB Support (Experimental)

To connect to a device via USB, install this package via:

```
pip install adb-shell[usb]
```

CHAPTER

THREE

EXAMPLE USAGE

(Based on androidtv/adb_manager.py)

```
from adb_shell.adb_device import AdbDeviceTcp, AdbDeviceUsb
from adb_shell.auth.sign_pythonrsa import PythonRSASigner

# Load the public and private keys
adbkey = 'path/to/adbkey'
with open(adbkey) as f:
    priv = f.read()
with open(adbkey + '.pub') as f:
     pub = f.read()
signer = PythonRSASigner(pub, priv)

# Connect
device1 = AdbDeviceTcp('192.168.0.222', 5555, default_transport_timeout_s=9.)
device1.connect(rsa_keys=[signer], auth_timeout_s=0.1)

# Connect via USB (package must be installed via `pip install adb-shell[usb])`
device2 = AdbDeviceUsb()
device2.connect(rsa_keys=[signer], auth_timeout_s=0.1)

# Send a shell command
response1 = device1.shell('echo TEST1')
response2 = device2.shell('echo TEST2')
```

3.1 Generate ADB Key Files

If you need to generate a key, you can do so as follows.

```
from adb_shell.auth.keygen import keygen

keygen('path/to/adbkey')
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

a

INDEX

Symbols

A

B

C

P

R

S

T

U

V

W