Test-SDK

Contents

Introduction

Product information

The ABC-Test SDK is a stand-alone, cross-platform, static library, which you can use to communicate to the Cloud.

The SDK is mainly written in C.

You can find the latest revision of the SDK at: *si.abc.com*

System requirements

Supported operating systems:

OS	x86	x86_64
Windows	✓	
Linux	✓	✓
MacOS X		✓

Third-party libraries

Depending on the platform, the following third-party libraries are used by ABC-Test SDK.

Library	License	Version
Apache APR	APACHE	1.4.5
Apache APR-UTIL	APACHE	1.3.12
bzlib2	PUBLIC	1.0.6
libcurl	PUBLIC DOMAIN	7.25.1
libcares	MIT	1.7.5
protobuf	BSD	2.3.0
stlport	PUBLIC DOMAIN	5.2.1
openssl	PUBLIC	1.0.1c

SDK archive structure

The SDK is organized in a standard structure.

bin

.dll files on Windows platforms, along with the SSL certificate used to validate the identity of our servers (cacert.crt)

docs

Documentation for the SDK.

include

C headers used to compile applications using the SDK.

lib

.lib files on Windows platforms (used for linking applications), or shared libraries on Unix-like platforms.

licenses

Licenses for all the third-party libraries used by the SDK.

src

The SDK demo application.

Compiling the sample application

The SDK contains a sample application, to demonstrate its basic usage, for example, to check the status of a file with the cloud and to upload it, if needed.

- Edit the file **src/file_demo_sdk.c** , by replacing the define values with the ones provided to you by ABC.

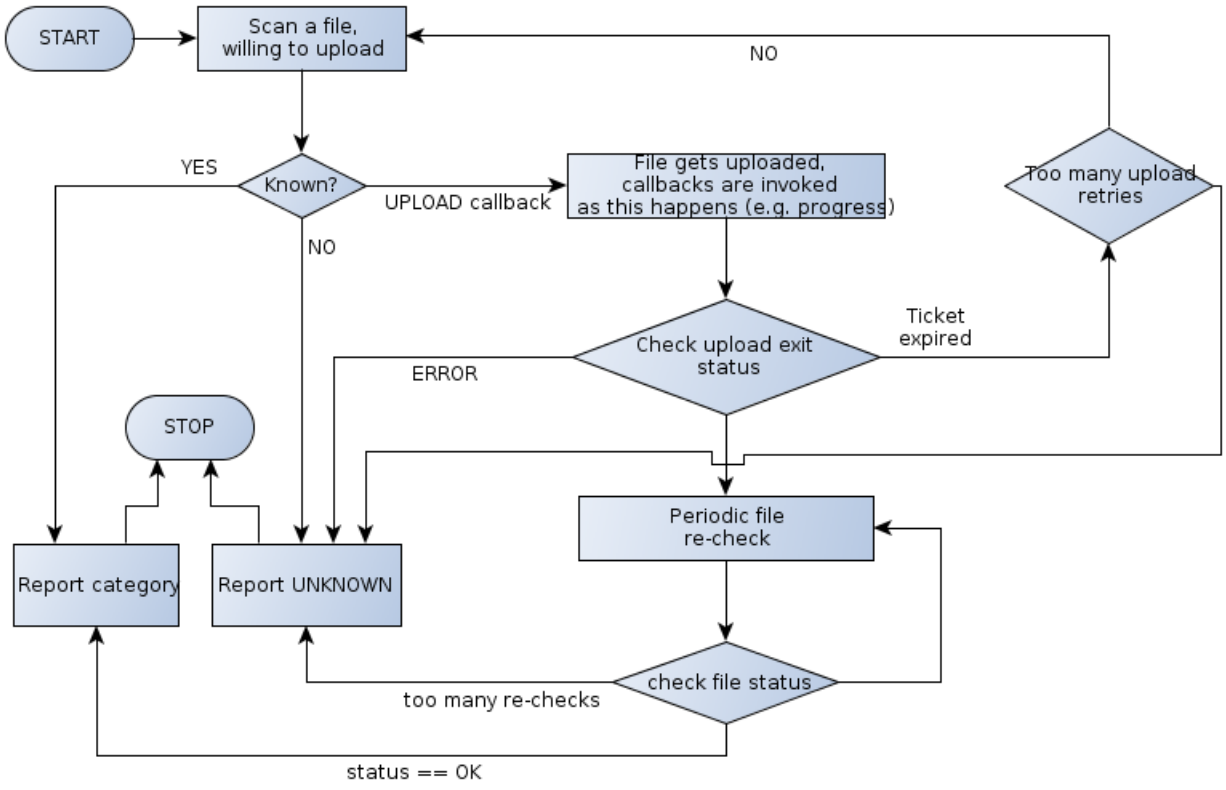
```
#define ABC_SDK_PRODUCT_ID 0
#define ABC_SDK_PRODUCT_LICENSE "-md5-sum-license-here-"
#define ABC_SDK_SERVICE_ADDRESS "https://auth.abc.com/"
```

- Compile the application using the **.sln** file (for Visual Studio 2010) or the **Makefile** (for UNIX platforms) in the **src** folder.

The executable is created in the **bin** folder.

Possible workflow

The diagram below provides a possible, simple workflow for working with the SDK.



The first step is to scan a file, or a series of files and, maybe, upload those unknown. For each of them, a status callback function is called, containing data like **category** and **status** . The next step is to check the **status** .

Possible status values:

- **OK** - The file has been analized and the **category** parameter shows the file's category. Having an **UNKNOWN** category can indicate an error on the server side, but it does not imply any action from the client. This step is followed by the **YES** and **NO** branch cases (depending on the category).
- **IN_PROGRESS** - The file has been uploaded by someone and it is in the process of being analyzed. The SDK user should call the check file/ hash function again, preferably after **ttl** seconds. This case is not drawn on the flowchart, for simplicity's sake.
- **UPLOAD** - The server signals the SDK to upload the file. This is the `UPLOAD callback` branch. The SDK will invoke the `upload init` callback (no intervention needed).

Once the file has been uploaded, the `upload finished` callback is triggered. If an error occurs, the user can be informed that the category is **UNKNOWN** . If the ticket expired, the entire scan request has to run from the top again, for a number of times. If the upload finished successfully, then a repeated check of this file should be performed, up to a number of times.

Using the ABC-Test SDK

Initializing the ABC-Test SDK

In an application's lifetime, you need to initialize and uninitialize the ABC-Test SDK only once.

- Before beginning to use the SDK, call the initialization function `abc_file_initialize()`.

If you don't initialize the SDK, most functions return an error code, except for `abc_enable_log()`.

Uninitializing the ABC-Test SDK

- To shut down the library, call the uninitialization function `abc_file_uninitialize()`.

You cannot use the library after uninitialization.

Creating a context

To create a connection to our ABC-Test service for online queries, you have to create and initialize a context containing internal state data.

Warning:

The contexts are not thread-safe. Do not use them for multiple threads simultaneously. Create a context for each thread you use in your application.

- Call `abc_create_context()` to create a context.
- Call `abc_file_initialize_context()` to start using the context.
- When you don't need the context any longer, call `abc_file_uninitialize_context()`, then `abc_destroy_context()`.

An error will occur, if you don't call the functions in this order.

Configuration

The API provides functions for global settings, which are used by existing and new contexts:

- Setting a proxy
- Using proxy credentials (user, password)
- Setting the connection and data timeout
- Using an SSL CA certificate

Callbacks

The ABC-Test SDK uses a callback mechanism to report and retrive information related to the actual analysing task from the application. These callbacks are functions which allow the application to perfom online lookups, taking a structure of function pointers as parameter.

None of these callbacks is mandatory. If you do not set any callback, the queries will be performed, but you will not receive any notifications about the events.

Warning:

If you have multiple threads and set the same callback function inside your application, make sure the function is thread safe, since it will be called from different threads simultaneously.

- `abc_file_error_cb` - The error callback, triggered when a file error occurs (for example when reading or opening a file).
- `abc_file_hash_status_cb` - The status callback, triggered to report the online status of a file/ hash (from the cloud).
- `abc_file_upload_init_cb` - The upload init callback, triggered to ask if the user agrees with uploading a file, and to pass some more information about it, if available.
- `abc_file_upload_start_cb` - The upload start callback, triggered when the upload is starting (called from a different thread).
- `abc_file_upload_progress_cb` - The upload progress callback, triggered to signal the upload progress (called from a different thread).
- `abc_file_upload_finish_cb` - The upload finish callback, triggered to signal that an upload has finished (called from a different thread).

Cloud queries

The SDK provides functions which can query for one or multiple files/ hashes online. The query functions are synchronous and they block program execution until all the results are available.

Warning:

If the SDK is 32-bit and runs under Windows x64, you will encounter an issue where the OS redirects the access to certain locations, so you will not be able to access parts of the filesystem. See *Wow64* .

Note:

When looking up multiple files/ hashes:

Before the functions return to the caller, some of the callbacks are called, with the results for each file/ hash.

- An online query by hash(es) triggers the `abc_file_hash_status_cb` callbacks until the function exits.
- An online query by file(es) triggers the `abc_file_hash_status_cb` callbacks and eventually `abc_file_fs_error_cb` and `abc_file_upload_init_cb` .
- In case the user accepts the upload, a newly created thread triggers the callbacks `abc_file_upload_start_cb` , `abc_file_upload_progress_cb` and `abc_file_upload_finish_cb` .

File uploads

Two API functions can be used to indicate if the SDK application agrees to upload files to the cloud or not: `abc_file_willing_to_upload()` and `abc_file_unwilling_to_upload()` . The reason for this is the following: uploading files to the cloud is not triggered by the application, but by the cloud. This is done via a special ticket, which is attached to the reply for a hash. When the SDK receives the ticket, it triggers the `abc_file_upload_init_cb` callback. After the application accepts, the file is added to an upload queue and sent to the cloud.

Warning:

If you do not set upload callbacks, the SDK behaves as if the ticket had been called.

Managing the process scan

There is a fixed number of threads which upload files to the cloud. Due to bandwidth limits, files can pile up in the upload queue, and the last ones can even get upload errors because their upload tickets will actually expire by the time they actually get processed. In order to avoid errors, you must manage the scan process from outside the SDK.

Solution #1: Feed the SDK tiny batches of files, and if there are uploads, wait until they are completed before continuing with the scan. This solution has the disadvantage of finishing the initial scan slower, because of the waiting for uploads.

Solution #2: Use `abc_file_unwilling_to_upload()` , which makes the SDK return the status **UPLOAD** on the `abc_file_hash_status_cb` , but not actually trigger the upload callback and the actual upload. The application using the SDK can then keep track of all the files which received this status and perform a second pass after calling `abc_file_willing_to_upload()` . Note that not all files might be requested for upload again (for example, if someone else uploaded a file in the mean time).

Tips and tricks

Memory handling

In the rare case of an out-of-memory error, the default behaviour is for the SDK to abort program execution: `abort()` . However, the user can set a different out-of-memory handling function which does not have the same behaviour.

Callback thread safety

Make sure the code is threadsafe, because the callbacks can be triggered from different threads.

Proxy usage

In UNIX environments, the **curl** library respects the **http(s)_proxy** variables by default, if set. If you want a different behaviour, you need to explicitely disable the proxy by calling the `abc_set_proxy` function with an empty string as proxy address.

SSL

When using the SSL support, call `abc_ssl_threads_initialize()` prior to initializing the library and `abc_ssl_threads_uninitialize()` after uninitializing it.

Wow64

If running a 32-bit SDK on Windows x64, the OS performs automatically a redirection at filesystem level, effectively “hiding” the contents of the native **system32** and of some other folders.

To access all the files, you have to disable this redirection in the thread where you call `abc_file_check_file` or `abc_file_check_files` (see *Wow64DisableWow64FsRedirection*). For the upload threads, this is performed automatically.

Retrying

There are cases in which you should keep retrying to scan a file, for example when you get the status **IN_PROGRESS** for a file. This status means that someone uploaded that file and it is being analyzed in the cloud. In this case, you should use the returned **ttl** value as a "retry after ttl seconds" measure. Likewise, if your application itself uploads a file, the same process applies. Limits should be set for the number of retries, for the maximum **ttl** value to be taken into consideration (for example, "do not retry after 1 week, if ttl says so"), etc.

Contact

For technical questions during evaluation or implementation do not hesitate to contact our Support Team at:

Support by email