

BoxedApp

Application Virtualization Solutions
from Softanics

Complete Guide

Contents

Table of contents	2
Introduction	3
Which Product to Choose	3
System Requirements	3
Virtual File System	4
Introduction	4
Creating Virtual Files	4
Custom Virtual Files: IStream-Based Files	4
Virtual Registry	5
Processes that Share Virtual Environment	5
Attached Processes	5
How Attachment Works	6
Virtual Process	6
Shared Memory	6
Typical Use Scenarios for BoxedApp SDK	6
Loading DLL from Memory	6
Using COM / ActiveX Object without Registering It in the Registry	7
Starting Application Directly from Memory	7
Intercepting Functions	7
BoxedApp SDK	9
BoxedApp SDK Functions	9
BoxedAppSDK_Init	11
BoxedAppSDK_Exit	11
BoxedAppSDK_EnableDebugLog	11
BoxedAppSDK_SetLogFile	12
BoxedAppSDK_WriteLog	12
BoxedAppSDK_EnableOption	13
BoxedAppSDK_IsOptionEnabled	13
BoxedAppSDK_RemoteProcess_EnableOption	14
BoxedAppSDK_RemoteProcess_IsOptionEnabled	14
BoxedAppSDK_CreateVirtualFile	14
BoxedAppSDK_CreateVirtualFileBasedOnIStream	16
BoxedAppSDK_CreateVirtualFileBasedOnBuffer	17
BoxedAppSDK_CreateVirtualDirectory	18
BoxedAppSDK_DeleteFileFromVirtualFilesystem	18
BoxedAppSDK_CreateVirtualRegKey	19
BoxedAppSDK_EnumVirtualRegKeys	20
BoxedAppSDK_RegisterCOMLibraryInVirtualRegistry	21
BoxedAppSDK_RegisterCOMServerInVirtualRegistry	21
BoxedAppSDK_Alloc	22
BoxedAppSDK_Free	22
BoxedAppSDK_AttachToProcess	22
BoxedAppSDK_DetachFromProcess	22
BoxedAppSDK_HookFunction	22
BoxedAppSDK_GetOriginalFunction	23
BoxedAppSDK_EnableHook	23
BoxedAppSDK_UnhookFunction	23
BoxedAppSDK_RemoteProcess_LoadLibrary	23

THE COMPLETE GUIDE

BoxedAppSDK_EmulateBoxedAppSDKDLL	24
BoxedAppSDK_SharedMem_Alloc	24
BoxedAppSDK_SharedMem_Free	24
BoxedAppSDK_SharedMem_Lock	24
BoxedAppSDK_SharedMem_Unlock	25
BoxedAppSDK_SharedMem_CreateStreamOnSharedMem	25
BoxedApp SDK Options	25
DEF_BOXEDAPPSDK_OPTION__ALL_CHANGES_ARE_VIRTUAL	25
DEF_BOXEDAPPSDK_OPTION__EMBED_BOXEDAPP_IN_CHILD_PROCESSES	26
DEF_BOXEDAPPSDK_OPTION__ENABLE_VIRTUAL_FILE_SYSTEM	26
DEF_BOXEDAPPSDK_OPTION__ENABLE_VIRTUAL_REGISTRY	26
DEF_BOXEDAPPSDK_OPTION__EMULATE_OUT_OF_PROC_COM_SERVERS	27
DEF_BOXEDAPPSDK_OPTION__INHERIT_OPTIONS	27

Introduction

BoxedApp is a family of products for meeting the challenges in the area of application virtualization for Windows® operating systems.

Application virtualization is primarily a simulation of a filesystem ("virtual filesystem"), registry ("virtual registry"), COM/ActiveX subsystem, and the process start system ("virtual processes").

BoxedApp is based on the interception and emulation of system calls, which an application makes to call the operating system with its requests to create files, list, remove, read, write them, etc., requests to create COM objects, run registry-related processes.

BoxedApp family of products:

- BoxedApp SDK
- BoxedApp Packer

BoxedApp SDK is a developer library for emulating the file system, registry and other Windows subsystems.

BoxedApp Packer is an end-user solution for packing third-party applications with all their dependencies into a single executable file (the so-called *portable applications*); it requires no programming whatsoever.

BoxedApp SDK comes as:

- bxsdk32.dll (for 32 bit applications) and bxsdk64.dll (for 64 bit applications), which can be used from any development environment, such as Visual C++, VB6, C#, VB.Net, Delphi, Builder C++.
- bxsdk32.lib (for 32 bit applications) and bxsdk64.lib (for 64 bit applications), a static library for Visual C++ (for all versions, from 6.0 to 2010).
- BoxedAppSDK_Static.pas, a delphi file for static linking with applications written in Delphi or Builder C++ (for all versions from 3.0 to 2011 / XE).

Supports both x86 and x64 platforms.

Which Product to Choose

BoxedApp SDK is a library for developers who want to emulate the file system and registry, run processes by loading them directly from the memory, use ActiveX without actually registering in the registry, and so on. Thus, BoxedApp SDK is a great choice for application developers.¹

BoxedApp Packer is an executable file packer solution. Use it if the files to be packaged are known. You can use BoxedApp Packer for turning your or a third-party application into a portable application. BoxedApp Packer does not require the source code of the application; it deals with the binary executable files. The result of BoxedApp Packer's performance is an executable binary file that contains all the source files in the packed form. When started, the compressed files are extracted and placed in a virtual environment, so the application runs exactly as the original application would run if the files actually existed on the disk.

All the products in the BoxedApp line of products use the same technology, which supports virtual environment.

System Requirements

All the products of the BoxedApp family support both 32-bit and 64-bit environments. Supports all current versions of Windows, beginning with Windows 2000.

BoxedApp Packer packs both 32-bit and 64-bit executable files. It can even pack 64-bit files on a 32-bit Windows system (and vice versa).

BoxedApp does not have any special requirements for RAM or disk space.

¹See also: [Typical use scenarios for BoxedApp SDK](#)

Virtual File System

Introduction

The heart of BoxedApp is the mechanism of intercepting calls to the operating system in the so-called user-mode). Unlike the systems that run in the kernel mode, BoxedApp does not require installing drivers and, hence, having the administrator privileges. In particular, BoxedApp intercepts all the calls that an application makes for using the file system.

BoxedApp introduces the concept of virtual file — a "file" that does not physically exist on the disk, but the application runs as if the "file" actually existed there. For example, an application attempts to open a virtual file C:\1.dll, by issuing the respective call. BoxedApp gets the control and checks the path to the file to be opened. If the path points to a virtual file, instead of calling the original file open function, BoxedApp returns a virtual handle that points to the virtual file.

A set of virtual files forms a virtual file system.

Creating Virtual Files

Virtual files can be created explicitly and implicitly. Explicitly, a virtual file can be created with the following functions:

- [BoxedAppSDK_CreateVirtualFile](#) — creates a virtual file with the contents located in the [shared memory](#).
- [BoxedAppSDK_CreateVirtualFileBasedOnIStream](#) — creates a virtual file with the behavior determined by the implementation of the [standard interface IStream](#).
- [BoxedAppSDK_CreateVirtualFileBasedOnBuffer](#) — creates a virtual file with the contents located in the memory buffer, which is used when reading and for writing (see function [BoxedAppSDK_CreateVirtualFileBasedOnBuffer](#)).
- [BoxedAppSDK_CreateVirtualDirectory](#) — creates a virtual directory with the contents located in the [shared memory](#).
- [BoxedAppSDK_CreateCustomVirtualDirectory](#) — creates a virtual directory with the behavior determined by the implementation of the special interface.

Implicitly, a virtual file can appear when the "[all changes are virtual](#)" mode is on. In this mode, a virtual file is created when you create a new file. If the parent directory of the file was created with [BoxedAppSDK_CreateCustomVirtualDirectory](#), the same method will be called for creating a file within it. And if the parent directory is neither a virtual file nor a file created with [BoxedAppSDK_CreateVirtualDirectory](#), then, depending on whether it's a directory or a regular file that is to be created, BoxedApp will use the [BoxedAppSDK_CreateVirtualFile](#) or [BoxedAppSDK_CreateVirtualDirectory](#) method.

Custom Virtual Files: IStream-Based Files

[BoxedAppSDK_CreateVirtualFile](#) creates a file with the contents located in the shared memory. This is the easiest way to create a virtual file, which is sufficient for the majority of cases. However, sometimes you may need to have a virtual file with its data stored in other places: database, encrypted file or the Internet. For that purpose, BoxedApp SDK allows creating a virtual file, based on the implementation of the standard interface IStream: [BoxedAppSDK_CreateVirtualFileBasedOnIStream](#).

Reading from such file takes the method `IStream::Read()`; writing to it calls `IStream::Write()`. Creating a new handler that points to such file requires `IStream::Clone()`, which returns IStream with its own current pointer. To set and get the current position in that file, BoxedApp uses `IStream::Seek()`. To resize the file, it uses `IStream::SetSize()`. To get the size of the file, it can call the method `IStream::Stat()`.

THE COMPLETE GUIDE

File operation	Method	Notes
Reading	<code>IStream::Read()</code>	
Writing	<code>IStream::Write()</code>	
Getting new file handler	<code>IStream::Clone()</code>	Creating a new handler that points to such file requires <code>IStream::Clone()</code> , which returns <code>IStream</code> with its own current pointer.
Set size	<code>IStream::SetSize()</code>	
Get current position	<code>IStream::Seek()</code>	
Get file size	<code>IStream::Stat()</code>	The size can be also obtained with <code>IStream::Seek()</code> ; BoxedApp can move the pointer to the end of the file (by setting the <code>STREAM_SEEK_END</code> flag and zero offset) and get the new position value. This would be the sought file size. That is why it is important to properly implement the <code>IStream::Seek()</code> method.

The virtual file system is shared among all the processes that share a [virtual environment](#).

Virtual Registry

Similar to the virtual file system, virtual registry is an internal structure of BoxedApp SDK, located in the memory. Virtual registry allows emulating registry keys and key values.

And since the registry is especially critical for using the COM system, the virtual registry allows setting up the [virtual registration of COM objects in the virtual registry](#).

To create a virtual registry key, use the function

[BoxedAppSDK_CreateVirtualRegKey](#).

Virtual keys and values can also be created implicitly, if the option "all changes are virtual" is set.

The virtual registry is shared among all the processes that share a [virtual environment](#).

Processes that Share Virtual Environment

Attached Processes

Virtual environment, virtual file system and virtual registry can be shared by multiple processes.

Suppose, some process has initialized BoxedApp (see the function [BoxedAppSDK_Init](#)). When initializing the shared memory (uses MMF — memory mapped files), BoxedApp creates a virtual registry and a virtual file system. Now, other process can also use the virtual file system and the virtual registry. We say that these two processes *share a virtual environment*. *Each of these processes is attached to the virtual environment*. The initialization of BoxedApp, when an "attachment" to a virtual environment takes place, is called attaching to virtual environment.

The primary way of attaching a process to a virtual environment is running it in the context of an already attached process.

Such inheritance is regulated by the BoxedApp SDK options:

[DEF_BOXEDAPPSDK_OPTION__EMBED_BOXEDAPP_IN_CHILD_PROCESSES](#) and
[DEF_BOXEDAPPSDK_OPTION__EMULATE_OUT_OF_PROC_COM_SERVERS](#).

If the option

[DEF_BOXEDAPPSDK_OPTION__EMBED_BOXEDAPP_IN_CHILD_PROCESSES](#) is set, all the child processes inherit the link to the virtual environment; i.e. use the same virtual file system and registry as the parent process.

The option [DEF_BOXEDAPPSDK_OPTION__EMULATE_OUT_OF_PROC_COM_SERVERS](#) matters when dealing with non-hosted COM servers. If the COM object is implemented as an EXE file that is virtual, BoxedApp will [run the virtual EXE](#) file to create the COM object.

If the COM object is implemented as an EXE file that does not exist in the virtual file system, by default the process based on the EXE file will be created by the system. And even if the option `DEF_BOXEDAPPSDK_OPTION__EMBEDDED` is set, the new process will not be attached to the virtual environment, and hence will not have access, for example, to the virtual files. To attach such processes, set the option `DEF_BOXEDAPPSDK_OPTION__EMULATE_OUT_OF_CONTEXT`.

To explicitly attach a process to a virtual environment, use the function [BoxedAppSDK_AttachToProcess](#). To detach a process, use the function [BoxedAppSDK_DetachFromProcess](#).

How Attachment Works

Technically, a process can be attached by running a thread (uses the winapi function `CreateRemoteThread`) in the context of the process to be attached. The thread takes care of configuring the system function interceptors.

When attaching to a child process that is to be created, once initialized, the thread passes the control to the main thread of the child process.

Virtual Process

If the executable file is virtual, creating a process based on that file (such process is called a *virtual process*) requires a different approach.

To create a virtual process, BoxedApp SDK selects a suitable exe file, which it can use for creating a new process. That exe file is called stub exe. Stub exe is run by `CreateProcess` with the `CREATE_SUSPENDED` flag. The actual stub exe file will not be run. BoxedApp expands the code of the virtual exe file in the context of the created process and passes the control to it.

A virtual process is always attached to a virtual environment, since it's created from a virtual file.

For the stub executable, BoxedApp uses the most suitable exe file from the Windows system folder.

Shared Memory

All [attached processes](#) have access to the shared memory, implemented on the basis of memory mapped files. It's the type of memory that stores data of the virtual file system and virtual registry. Developer can allocate and release memory blocks in the shared memory. These actions are facilitated by the [BoxedAppSDK_SharedMem_Alloc](#) and [BoxedAppSDK_SharedMem_Free](#) functions. [BoxedAppSDK_SharedMem_Alloc](#) returns a *handler* pointing at the memory block. To get the direct access to the allocated memory block, use the function [BoxedAppSDK_SharedMem_Lock](#), which returns the pointer. Once the data from the memory block is read, or the block is modified, BoxedApp calls the function [BoxedAppSDK_SharedMem_Unlock](#). This API is very similar to the winapi functions `LocalAlloc` / `LocalLock` / `LocalUnlock` / `LocalFree`.

Typical Use Scenarios for BoxedApp SDK

The ability to create virtual files, folders, registry keys and values gives developer a handy tool. Let's review the most typical cases of using the SDK.

Loading DLL from Memory

Suppose that a third-party component is available exclusively as a DLL. Nevertheless, we need to get a single exe file at the output. There are several reasons for that:

- We may need to conceal the fact of using the DLL.
- Or we may need to protect the DLL from tampering by hackers.

With BoxedApp SDK, developer creates a virtual file (for example, using the function [BoxedAppSDK_CreateVirtualFile](#)) and writes the contents of the DLL file to that file. It can obtain data from any source: Internet, LAN, database or application resources. Finally, the data can be generated on the fly.

Now you can download the library using the `LoadLibrary` function, and the application will run just as if the DLL file actually existed on the disk.

Using COM / ActiveX Object without Registering It in the Registry

COM/ActiveX components are traditionally registered in the system registry to allow the corresponding Windows API functions find the modules (DLL or EXE) that contain the code of the components.

A major problem that occurs with the registration is the lack of user privileges, necessary for writing to the registry.

Also, when creating portable applications, it is important to be able to run applications without prior installation.

BoxedApp SDK allows registering COM / ActiveX components in the virtual registry.

The two functions that facilitate this are:

- [BoxedAppSDK_RegisterCOMLibraryInVirtualRegistry](#), which registers the library of COM components made out as a DLL in the virtual registry.
- [BoxedAppSDK_RegisterCOMServerInVirtualRegistry](#), which registers the library of COM components made out as an EXE in the virtual registry.

After calling the required function, the application can create COM / ActiveX objects just as if they were actually registered in the system registry.

Starting Application Directly from Memory

With a virtual file at hand, you can create a process based on the file by calling any function that creates a process: `CreateProcess`, `WinExec` or `ShellExecute`.

BoxedApp SDK intercepts the process creation request. If it detects an attempt to create a process based on the virtual file, it loads the so-called "stub executable" creates the process based on it, writes the contents of the virtual file to it and then passes the control to it. Such [processes are called virtual](#).

Intercepting Functions

As you already know, [BoxedApp is based on intercepting system functions](#). The interception mechanism used by BoxedApp is also available to developers. It includes the following functions:

- [BoxedAppSDK_HookFunction](#), which creates a hook and, optionally, activates it.
- [BoxedAppSDK_EnableHook](#), which activates the hook.
- [BoxedAppSDK_GetOriginalFunction](#), which returns the pointer, which can be used for calling the original function.
- [BoxedAppSDK_UnhookFunction](#), which clears the hook.

Here is how hooks work. The address of the function, calls from which are to be intercepted, is passed to `BoxedAppSDK_HookFunction`. For example, for the function `kernel32.dll!CreateFileW`:

```
[C++]
PVOID pCreateFileW =
    (PVOID)GetProcAddress(GetModuleHandle(L"kernel32.dll"), "CreateFileW");

HANDLE g_hCreateFileWHook =
    BoxedAppSDK_HookFunction(
        pCreateFileW,
        &My_CreateFileW,
        TRUE);
```

The interceptor function gets the control when someone calls the function. In this example, that's the function `kernel32.dll!CreateFileW`, which is called when creating or opening files. You can always call the original function, the address of which you can get using `BoxedAppSDK_GetOriginalFunction`:


```
[C++]
HANDLE WINAPI My_CreateFileW(
    LPCWSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
)
{
    // ...

    // You can call original function if you need
    typedef HANDLE (WINAPI *P_CreateFileW)(
        LPCWSTR lpFileName,
        DWORD dwDesiredAccess,
        DWORD dwShareMode,
        LPSECURITY_ATTRIBUTES lpSecurityAttributes,
        DWORD dwCreationDisposition,
        DWORD dwFlagsAndAttributes,
        HANDLE hTemplateFile);

    P_CreateFileW pCreateFileW =
        (P_CreateFileW)BoxedAppSDK_GetOriginalFunction(g_hCreateFileWHook);

    return pCreateFileW(
        lpFileName,
        dwDesiredAccess,
        dwShareMode,
        lpSecurityAttributes,
        dwCreationDisposition,
        dwFlagsAndAttributes,
        hTemplateFile
    );
}
```

Here is a similar example in Delphi:

```

[Delphi]
type
TCreateFileW =
    function(lpFileName: PWideChar;
            dwDesiredAccess, dwShareMode: Integer;
            lpSecurityAttributes: PSecurityAttributes;
            dwCreationDisposition, dwFlagsAndAttributes: DWORD;
            hTemplateFile: THandle): THandle; stdcall;

var
    OriginalCreateFileW: TCreateFileW;

function My_CreateFileW(
    lpFileName: PWideChar;
    dwDesiredAccess, dwShareMode: Integer;
    lpSecurityAttributes: PSecurityAttributes;
    dwCreationDisposition, dwFlagsAndAttributes: DWORD;
    hTemplateFile: THandle): THandle; stdcall;
begin
    ...
    Result :=
        OriginalCreateFileW(
            lpFileName,
            dwDesiredAccess,
            dwShareMode,
            lpSecurityAttributes,
            dwCreationDisposition,
            dwFlagsAndAttributes,
            hTemplateFile);
end;

var
    pCreateFileW: Pointer;
    hHook__CreateFileW: THandle;

begin
    BoxedAppSDK_Init;

    pCreateFileW := GetProcAddress(GetModuleHandle('kernel32.dll'), 'CreateFileW');
    hHook__CreateFileW := BoxedAppSDK_HookFunction(pCreateFileW, @My_CreateFileW, FALSE);

    OriginalCreateFileW := BoxedAppSDK_GetOriginalFunction(hHook__CreateFileW);
    BoxedAppSDK_EnableHook(hHook__CreateFileW, TRUE);

end.

```

BoxedApp SDK

BoxedApp SDK Functions

Here is the list of BoxedApp SDK functions, split by functionality:

- Basic functions
 - [BoxedAppSDK_Init](#)
 - [BoxedAppSDK_Exit](#)

- Memory
 - [BoxedAppSDK_Alloc](#)
 - [BoxedAppSDK_Free](#)
- Logging
 - [BoxedAppSDK_EnableDebugLog](#)
 - [BoxedAppSDK_SetLogFile](#)
 - [BoxedAppSDK_WriteLog](#)
- Options
 - [BoxedAppSDK_EnableOption](#)
 - [BoxedAppSDK_IsOptionEnabled](#)
- Virtual file system
 - [BoxedAppSDK_CreateVirtualFile](#)
 - [BoxedAppSDK_CreateVirtualFileBasedOnIStream](#)
 - [BoxedAppSDK_CreateVirtualFileBasedOnBuffer](#)
 - [BoxedAppSDK_CreateVirtualDirectory](#)
 - [BoxedAppSDK_DeleteFileFromVirtualFileSystem](#)
- Virtual registry
 - [BoxedAppSDK_CreateVirtualRegKey](#)
 - [BoxedAppSDK_EnumVirtualRegKeys](#)
- COM subsystem
 - [BoxedAppSDK_RegisterCOMLibraryInVirtualRegistry](#)
 - [BoxedAppSDK_RegisterCOMServerInVirtualRegistry](#)
- Attached processes
 - [BoxedAppSDK_AttachToProcess](#)
 - [BoxedAppSDK_DetachFromProcess](#)
 - [BoxedAppSDK_RemoteProcess_LoadLibrary](#)
 - [BoxedAppSDK_RemoteProcess_EnableOption](#)
 - [BoxedAppSDK_RemoteProcess_IsOptionEnabled](#)
- Hooks
 - [BoxedAppSDK_HookFunction](#)
 - [BoxedAppSDK_GetOriginalFunction](#)
 - [BoxedAppSDK_EnableHook](#)
 - [BoxedAppSDK_UnhookFunction](#)
- Shared memory
 - [BoxedAppSDK_SharedMem_Alloc](#)
 - [BoxedAppSDK_SharedMem_Free](#)
 - [BoxedAppSDK_SharedMem_Lock](#)
 - [BoxedAppSDK_SharedMem_Unlock](#)
 - [BoxedAppSDK_SharedMem_CreateStreamOnSharedMem](#)
- Miscellaneous
 - [BoxedAppSDK_EmulateBoxedAppSDKDLL](#)

BoxedAppSDK_Init

Initializes BoxedApp SDK, installs all the hooks, necessary for SDK operation, creates a virtual file system and a virtual registry in the memory.

```
[C++]
BOOL __stdcall BoxedAppSDK_Init();

[Delphi]
function BoxedAppSDK_Init: BOOL; stdcall;
```

BoxedAppSDK_Exit

Stops BoxedApp SDK, clears all hooks that were set.

```
[C++]
void __stdcall BoxedAppSDK_Exit();

[Delphi]
function BoxedAppSDK_Exit: BOOL; stdcall;
```

BoxedAppSDK_EnableDebugLog

Enables or disables outputting debug data.

During its operation, the SDK can record debug data that contains details on the performance of the SDK. By default, outputting debug data is disabled, but it can be enabled with `BoxedAppSDK_EnableDebugLog`. The SDK will output the data to the debugger window, but you can also have it [duplicate all the output to a file using `BoxedAppSDK_SetLogFile`](#).

```
[C++]
void __stdcall BoxedAppSDK_EnableDebugLog(BOOL bEnable);

[Delphi]
procedure BoxedAppSDK_EnableDebugLog(bEnable: BOOL); stdcall;
```

Examples of use:

```
[C++]
...
void main()
{
    BoxedAppSDK_Init();

#ifdef _DEBUG

    BoxedAppSDK_SetLogFileW(L"boxedapp.log");
    BoxedAppSDK_EnableDebugLog(TRUE);

#endif // _DEBUG
}

[Delphi]
...
begin
    BoxedAppSDK_Init();

    BoxedAppSDK_SetLogFile('boxedapp.log');
    BoxedAppSDK_EnableDebugLog(true);
...
end;
```

BoxedAppSDK_SetLogFile

During its operation, the SDK can record debug data that contains details on the performance of the SDK. By default, outputting debug data is disabled, but it can be turned on with [BoxedAppSDK_EnableDebugLog](#). The SDK will output the data to the debugger window, but you can also have it duplicate all the output to a file with `BoxedAppSDK_SetLogFile`.

```
[C++]
void __stdcall BoxedAppSDK_SetLogFile(LPCTSTR szLogFilePath);
void __stdcall BoxedAppSDK_SetLogFileA(LPCSTR szLogFilePath);
void __stdcall BoxedAppSDK_SetLogFileW(LPCWSTR szLogFilePath);
```

```
[Delphi]
procedure BoxedAppSDK_SetLogFile(szLogFilePath: PAnsiChar); stdcall;
procedure BoxedAppSDK_SetLogFileA(szLogFilePath: PAnsiChar); stdcall;
procedure BoxedAppSDK_SetLogFileW(szLogFilePath: PWideChar); stdcall;
```

Examples of use:

```
[C++]
...
void main()
{
    BoxedAppSDK_Init();

#ifdef _DEBUG

    BoxedAppSDK_SetLogFileW(L"boxedapp.log");
    BoxedAppSDK_EnableDebugLog(TRUE);

#endif // _DEBUG
}
```

```
[Delphi]
...
begin
    BoxedAppSDK_Init();

    BoxedAppSDK_SetLogFile('boxedapp.log');
    BoxedAppSDK_EnableDebugLog(true);
...
end;
```

BoxedAppSDK_WriteLog

During its operation, the SDK can record debug data that contains details on the performance of the SDK. By default, outputting debug data is disabled, but it can be enabled with

[BoxedAppSDK_EnableDebugLog](#). Plus, you can add some custom data to the log; that is what this function is meant for.

```
[C++]
void __stdcall BoxedAppSDK_WriteLog(LPCTSTR szMessage);
void __stdcall BoxedAppSDK_WriteLogA(LPCSTR szMessage);
void __stdcall BoxedAppSDK_WriteLogW(LPCWSTR szMessage);
```

```
[Delphi]
procedure BoxedAppSDK_WriteLog(szLogFilePath: PAnsiChar); stdcall;
procedure BoxedAppSDK_WriteLogA(szLogFilePath: PAnsiChar); stdcall;
procedure BoxedAppSDK_WriteLogW(szLogFilePath: PWideChar); stdcall;
```

BoxedAppSDK_EnableOption

You can control the behavior of the SDK by setting the required options. The function `BoxedAppSDK_EnableOption` sets option values: each option can be either enabled or disabled. Also, each option has a default value. The details on the available options are accumulated [here](#).

[C++]

```
void __stdcall BoxedAppSDK_EnableOption(DWORD dwOptionIndex, BOOL bEnable);
```

[Delphi]

```
procedure BoxedAppSDK_EnableOption(dwOptionIndex: DWORD; bEnable: BOOL); stdcall;
```

BoxedAppSDK_IsOptionEnabled

You can control the behavior of the SDK by setting the required options. With the function `BoxedAppSDK_IsOptionEnabled`, you can find out the current value of the option; each option can be either enabled or disabled. Each option has a default value. The details on the available options are accumulated [here](#).

[C++]

```
BOOL __stdcall BoxedAppSDK_IsOptionEnabled(DWORD dwOptionIndex);
```

[Delphi]

```
function BoxedAppSDK_IsOptionEnabled(dwOptionIndex: DWORD): BOOL; stdcall;
```

Examples of use:

[C++]

```
...
void main()
{
    BoxedAppSDK_Init();
    ...
    BoxedAppSDK_EnableOption(DEF_BOXEDAPPSDK_OPTION__ALL_CHANGES_ARE_VIRTUAL, TRUE);
    {
        // File 'virtual.txt' will be created as virtual

        HANDLE hFile =
            CreateFile(
                _T("virtual.txt"),
                GENERIC_WRITE,
                FILE_SHARE_READ,
                NULL,
                CREATE_NEW,
                0,
                NULL
            );

    }
    BoxedAppSDK_EnableOption(DEF_BOXEDAPPSDK_OPTION__ALL_CHANGES_ARE_VIRTUAL, FALSE);
    ...
}
```

```
[Delphi]
...
var
    fs: TFileStream;
begin
    BoxedAppSDK_Init();

    BoxedAppSDK_EnableOption(DEF_BOXEDAPPSDK_OPTION__ALL_CHANGES_ARE_VIRTUAL, true);

    // File 'virtual.txt' will be created as virtual
    fs := TFileStream.Create('virtual.txt', fmCreate or fmOpenWrite);
    fs.Free();

    BoxedAppSDK_EnableOption(DEF_BOXEDAPPSDK_OPTION__ALL_CHANGES_ARE_VIRTUAL, false);
...
end;
```

BoxedAppSDK_RemoteProcess_EnableOption

You can control the behavior of the SDK by setting the required options. If for current process that's the function [BoxedAppSDK_EnableOption](#), for other, [attached process](#), that's the function [BoxedAppSDK_RemoteProcess_EnableOption](#).

```
[C++]
void __stdcall BoxedAppSDK_RemoteProcess_EnableOption(
    DWORD dwProcessId,
    DWORD dwOptionIndex,
    BOOL bEnable);
```

```
[Delphi]
procedure BoxedAppSDK_RemoteProcess_EnableOption(
    dwProcessId: DWORD;
    dwOptionIndex: DWORD;
    bEnable: BOOL); stdcall;
```

BoxedAppSDK_RemoteProcess_IsOptionEnabled

You can control the behavior of the SDK by setting the required options. If for current process that's the function [BoxedAppSDK_IsOptionEnabled](#), for other, [attached process](#), that's the function [BoxedAppSDK_RemoteProcess_IsOptionEnabled](#).

```
[C++]
BOOL __stdcall BoxedAppSDK_RemoteProcess_IsOptionEnabled(
    DWORD dwProcessId,
    DWORD dwOptionIndex);
```

```
[Delphi]
function BoxedAppSDK_RemoteProcess_IsOptionEnabled(
    dwProcessId: DWORD;
    dwOptionIndex: DWORD): BOOL; stdcall;
```

BoxedAppSDK_CreateVirtualFile

The function creates a [virtual file](#) with the contents located in the memory, shared among all the [attached processes](#).

The arguments of [BoxedAppSDK_CreateVirtualFile](#) are similar to the arguments of the winapi function [CreateFile](#).

```
[C++]
HANDLE __stdcall BoxedAppSDK_CreateVirtualFile(
    LPCTSTR szPath,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);

[Delphi]
function BoxedAppSDK_CreateVirtualFile(
    lpFileName: PAnsiChar;
    dwDesiredAccess, dwShareMode: Integer;
    lpSecurityAttributes: PSecurityAttributes;
    dwCreationDisposition, dwFlagsAndAttributes: DWORD;
    hTemplateFile: THandle): THandle; stdcall;

function BoxedAppSDK_CreateVirtualFileA(
    lpFileName: PAnsiChar;
    dwDesiredAccess, dwShareMode: Integer;
    lpSecurityAttributes: PSecurityAttributes;
    dwCreationDisposition, dwFlagsAndAttributes: DWORD;
    hTemplateFile: THandle): THandle; stdcall;

function BoxedAppSDK_CreateVirtualFileW(
    lpFileName: PWideChar;
    dwDesiredAccess, dwShareMode: Integer;
    lpSecurityAttributes: PSecurityAttributes;
    dwCreationDisposition, dwFlagsAndAttributes: DWORD;
    hTemplateFile: THandle): THandle; stdcall;
```

Examples of use:

```
[C++]
HANDLE hFile__DLL1 =
    BoxedAppSDK_CreateVirtualFile(
        _T("DLL1.dll"),
        GENERIC_WRITE,
        FILE_SHARE_READ,
        NULL,
        CREATE_NEW,
        0,
        NULL
    );

LPVOID pBuffer;
DWORD dwSize;
LoadResourceHelper(_T("IDR_BIN_DLL"), _T("DLL"), pBuffer, dwSize);

DWORD dwTemp;
WriteFile(hFile__DLL1, pBuffer, dwSize, &dwTemp, NULL);

CloseHandle(hFile__DLL1);
```



```
[Delphi]
var
  Handle1: THandle;
  HandleStream: THandleStream;
  ResourceStream: TResourceStream;
begin
  ...
  Handle1 :=
    BoxedAppSDK_CreateVirtualFile(
      'c:\1.jpg',
      GENERIC_WRITE,
      FILE_SHARE_READ,
      nil,
      CREATE_NEW,
      0,
      0);

  HandleStream := THandleStream.Create(Handle2);

  ResourceStream := TResourceStream.Create(0, 'IMG1', 'IMG');

  HandleStream.CopyFrom(ResourceStream, ResourceStream.Size);

  HandleStream.Free;
  ResourceStream.Free;

  CloseHandle(Handle1);
end;
```

BoxedAppSDK_CreateVirtualFileBasedOnIStream

The function creates a [virtual file](#), the behavior of which is determined by the implementation of the [standard interface IStream](#).

The arguments of BoxedAppSDK_CreateVirtualFileBasedOnIStream are similar to the arguments of the winapi function CreateFile.

```
[C++]
HANDLE __stdcall BoxedAppSDK_CreateVirtualFileBasedOnIStream(
    LPCTSTR szPath,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile,

    LPSTREAM pStream
);
```

```
[Delphi]
function BoxedAppSDK_CreateVirtualFileBasedOnIStream(
    lpFileName: PAnsiChar;
    dwDesiredAccess, dwShareMode: Integer;
    lpSecurityAttributes: PSecurityAttributes;
    dwCreationDisposition, dwFlagsAndAttributes: DWORD;
    hTemplateFile: THandle;
    pStream: IStream): THandle; stdcall;

function BoxedAppSDK_CreateVirtualFileBasedOnIStreamA(
    lpFileName: PAnsiChar;
    dwDesiredAccess, dwShareMode: Integer;
    lpSecurityAttributes: PSecurityAttributes;
    dwCreationDisposition, dwFlagsAndAttributes: DWORD;
    hTemplateFile: THandle;
    pStream: IStream): THandle; stdcall;

function BoxedAppSDK_CreateVirtualFileBasedOnIStreamW(
    lpFileName: PWideChar;
    dwDesiredAccess, dwShareMode: Integer;
    lpSecurityAttributes: PSecurityAttributes;
    dwCreationDisposition, dwFlagsAndAttributes: DWORD;
    hTemplateFile: THandle;
    pStream: IStream): THandle; stdcall;
```

BoxedAppSDK_CreateVirtualFileBasedOnBuffer

The function creates a [virtual file](#) with the contents located in the specified buffer. Thus, the contents is both read from the buffer and written to the specified buffer. That file is useful when you need to create a virtual file of an invariable size (in particular, when the file is read-only).

The arguments of BoxedAppSDK_CreateVirtualFileBasedOnBuffer are similar to the arguments of the winapi function CreateFile.

```
[C++]
HANDLE _stdcall BoxedAppSDK_CreateVirtualFileBasedOnBuffer(
    LPCTSTR szPath,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile,

    PVOID pData,
    DWORD dwSize
);
```

```
[Delphi]
function BoxedAppSDK_CreateVirtualFileBasedOnBuffer(
    lpFileName: PAnsiChar;
    dwDesiredAccess, dwShareMode: Integer;
    lpSecurityAttributes: PSecurityAttributes;
    dwCreationDisposition, dwFlagsAndAttributes: DWORD;
    hTemplateFile: THandle;
    pData: Pointer;
    dwSize: DWORD): THandle; stdcall;

function BoxedAppSDK_CreateVirtualFileBasedOnBufferA(
    lpFileName: PAnsiChar;
    dwDesiredAccess, dwShareMode: Integer;
    lpSecurityAttributes: PSecurityAttributes;
    dwCreationDisposition, dwFlagsAndAttributes: DWORD;
    hTemplateFile: THandle;
    pData: Pointer;
    dwSize: DWORD): THandle; stdcall;

function BoxedAppSDK_CreateVirtualFileBasedOnBufferW(
    lpFileName: PWideChar;
    dwDesiredAccess, dwShareMode: Integer;
    lpSecurityAttributes: PSecurityAttributes;
    dwCreationDisposition, dwFlagsAndAttributes: DWORD;
    hTemplateFile: THandle;
    pData: Pointer;
    dwSize: DWORD): THandle; stdcall;
```

BoxedAppSDK_CreateVirtualDirectory

The function creates a [virtual directory](#) with the contents located in the shared memory.

The arguments of BoxedAppSDK_CreateVirtualDirectory are similar to the arguments of the winapi function CreateDirectory.

```
[C++]
BOOL __stdcall BoxedAppSDK_CreateVirtualDirectory(
    LPCTSTR lpPathName,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);

[Delphi]
function BoxedAppSDK_CreateVirtualDirectory(
    lpPathName: PAnsiChar;
    lpSecurityAttributes: PSecurityAttributes): BOOL; stdcall;

function BoxedAppSDK_CreateVirtualDirectoryA(
    lpPathName: PAnsiChar;
    lpSecurityAttributes: PSecurityAttributes): BOOL; stdcall;

function BoxedAppSDK_CreateVirtualDirectoryW(
    lpPathName: PWideChar;
    lpSecurityAttributes: PSecurityAttributes): BOOL; stdcall;
```

BoxedAppSDK_DeleteFileFromVirtualFileSystem

The function deletes the file from the virtual file system. If the file is real, it is marked as deleted.

If the file is virtual, the function completely removes it from the virtual environment. It is important to note the difference between this case and the situation when the virtual file is deleted by the regular winapi function

THE COMPLETE GUIDE

DeleteFile. When deleting the file this way, the virtual file is only marked as deleted but is not actually removed. It is done this way, so that when the virtual file is created all over, its behavior would be determined by the function that the file was originally created with. Suppose, the [virtual file was created based on IStream](#). It is removed with the DeleteFile, which is followed by calling the CreateFile with the name of that virtual file. In this case, the behavior of the newly created file will also be determined by the implementation of IStream. On the other hand, if the virtual file is deleted using BoxedAppSDK_DeleteFileFromVirtualFileSystem, the new file can be created as a real one. Thus, BoxedAppSDK_DeleteFileFromVirtualFileSystem, deletes information on the behavior of the virtual file.

[C++]

```
DWORD __stdcall BoxedAppSDK_DeleteFileFromVirtualFileSystem(LPCTSTR szPath);
```

[Delphi]

```
function BoxedAppSDK_DeleteFileFromVirtualFileSystem(szPath: PAnsiChar): Longint; stdcall;  
function BoxedAppSDK_DeleteFileFromVirtualFileSystemA(szPath: PAnsiChar): Longint; stdcall;  
function BoxedAppSDK_DeleteFileFromVirtualFileSystemW(szPath: PWideChar): Longint; stdcall;
```

BoxedAppSDK_CreateVirtualRegKey

The function creates a virtual registry key.

Its arguments are similar to the arguments of the winapi function RegCreateKeyEx.

[C++]

```
LONG __stdcall BoxedAppSDK_CreateVirtualRegKey(  
    HKEY hKey,  
    LPCTSTR lpSubKey,  
    DWORD Reserved,  
    LPCTSTR lpClass,  
    DWORD dwOptions,  
    REGSAM samDesired,  
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,  
    PHKEY phkResult,  
    LPDWORD lpdwDisposition  
);
```

```
[Delphi]
function BoxedAppSDK_CreateVirtualRegKey(
    hKey: HKEY;
    lpSubKey: PAnsiChar;
    Reserved: DWORD;
    lpClass: PAnsiChar;
    dwOptions: DWORD;
    samDesired: REGSAM;
    lpSecurityAttributes: PSecurityAttributes;
    var phkResult: HKEY;
    lpdwDisposition: PDWORD): Longint; stdcall;

function BoxedAppSDK_CreateVirtualRegKeyA(
    hKey: HKEY;
    lpSubKey: PAnsiChar;
    Reserved: DWORD;
    lpClass: PAnsiChar;
    dwOptions: DWORD;
    samDesired: REGSAM;
    lpSecurityAttributes: PSecurityAttributes;
    var phkResult: HKEY;
    lpdwDisposition: PDWORD): Longint; stdcall;

function BoxedAppSDK_CreateVirtualRegKeyW(
    hKey: HKEY;
    lpSubKey: PWideChar;
    Reserved: DWORD;
    lpClass: PWideChar;
    dwOptions: DWORD;
    samDesired: REGSAM;
    lpSecurityAttributes: PSecurityAttributes;
    var phkResult: HKEY;
    lpdwDisposition: PDWORD): Longint; stdcall;
```

BoxedAppSDK_EnumVirtualRegKeys

The function lists all virtual registry keys.

The argument of the function is a pointer to a callback function that is called for each virtual key. The callback function returns TRUE to continue or FALSE to halt listing the keys.

```
[C++]
typedef BOOL (WINAPI *P_BoxedAppSDK_EnumVirtualRegKeysCallback)(
    HKEY hRootKey,
    LPCTSTR szSubKey,
    LPARAM lParam);

BOOL __stdcall BoxedAppSDK_EnumVirtualRegKeys(
    P_BoxedAppSDK_EnumVirtualRegKeysCallback pEnumFunc,
    LPARAM lParam);
```

```
[Delphi]
TBoxedAppSDK_EnumVirtualRegKeysCallbackA =
    function(hRootKey: HKEY;
              szSubKey: PAnsiChar;
              lParam: Cardinal): Boolean; stdcall;

TBoxedAppSDK_EnumVirtualRegKeysCallbackW =
    function(hRootKey: HKEY;
              szSubKey: PWideChar;
              lParam: Cardinal): Boolean; stdcall;

TBoxedAppSDK_EnumVirtualRegKeysCallback = TBoxedAppSDK_EnumVirtualRegKeysCallbackA;

function BoxedAppSDK_EnumVirtualRegKeys(
    pEnumFunc: TBoxedAppSDK_EnumVirtualRegKeysCallback;
    lParam: Cardinal): Boolean; stdcall;

function BoxedAppSDK_EnumVirtualRegKeysA(
    pEnumFunc: TBoxedAppSDK_EnumVirtualRegKeysCallbackA;
    lParam: Cardinal): Boolean; stdcall;

function BoxedAppSDK_EnumVirtualRegKeysW(
    pEnumFunc: TBoxedAppSDK_EnumVirtualRegKeysCallbackW;
    lParam: Cardinal): Boolean; stdcall;
```

BoxedAppSDK_RegisterCOMLibraryInVirtualRegistry

The function registers a COM/ActiveX library in the virtual registry, thus making the COM objects that are implemented in the library available to all the [attached processes](#). This allows [using COM objects without registering in the system registry](#).

```
[C++]
DWORD __stdcall BoxedAppSDK_RegisterCOMLibraryInVirtualRegistry(LPCTSTR szPath);

[Delphi]
function BoxedAppSDK_RegisterCOMLibraryInVirtualRegistry(
    szVirtualFilePath: PAnsiChar): Longint; stdcall;

function BoxedAppSDK_RegisterCOMLibraryInVirtualRegistryA(
    szVirtualFilePath: PAnsiChar): Longint; stdcall;

function BoxedAppSDK_RegisterCOMLibraryInVirtualRegistryW(
    szVirtualFilePath: PWideChar): Longint; stdcall;
```

BoxedAppSDK_RegisterCOMServerInVirtualRegistry

The function registers a COM server, implemented as an executable file (exe), in the virtual registry, thus making the COM objects that are implemented in the server available to all the [attached processes](#). This allows [using COM objects without registering in the system registry](#).

```
[C++]
DWORD __stdcall BoxedAppSDK_RegisterCOMServerInVirtualRegistry(LPCTSTR szCommandLine);
```

```
[Delphi]
function BoxedAppSDK_RegisterCOMServerInVirtualRegistry(
    szCommandLine: PAnsiChar): Longint; stdcall;
function BoxedAppSDK_RegisterCOMServerInVirtualRegistryA(
    szCommandLine: PAnsiChar): Longint; stdcall;
function BoxedAppSDK_RegisterCOMServerInVirtualRegistryW(
    szCommandLine: PWideChar): Longint; stdcall;
```

BoxedAppSDK__Alloc

The function allocates a memory block.

```
[C++]
PVOID __stdcall BoxedAppSDK_Alloc(DWORD dwSize);

[Delphi]
function BoxedAppSDK_Alloc(dwSize: DWORD): Pointer; stdcall;
```

BoxedAppSDK__Free

Releases a memory block.

```
[C++]
BOOL __stdcall BoxedAppSDK_Free(PVOID pData);

[Delphi]
function BoxedAppSDK_Free(pData: Pointer): DWORD; stdcall;
```

BoxedAppSDK__AttachToProcess

The function attaches the process to the [shared virtual environment](#). Please note that the argument of the function is the handle of the process, and not its id.

```
[C++]
BOOL __stdcall BoxedAppSDK_AttachToProcess(HANDLE hProcess);

[Delphi]
function BoxedAppSDK_AttachToProcess(hProcess: THandle): BOOL; stdcall;
```

BoxedAppSDK__DetachFromProcess

The function detaches the process from the [shared virtual environment](#). Please note that the argument of the function is the handle of the process, and not its id.

```
[C++]
BOOL __stdcall BoxedAppSDK_DetachFromProcess(HANDLE hProcess);

[Delphi]
function BoxedAppSDK_DetachFromProcess(hProcess: THandle): BOOL; stdcall;
```

BoxedAppSDK__HookFunction

Creates a hook for the specified function. For more information on using hooks, please see ["Intercepting Functions"](#).

```
[C++]
HANDLE __stdcall BoxedAppSDK_HookFunction(
    PVOID pFunction,
    PVOID pHook,
    BOOL bEnable);
```

```
[Delphi]
function BoxedAppSDK_HookFunction(
    pFunction: Pointer;
    pHook: Pointer;
    bEnable: BOOL): THandle; stdcall;
```

BoxedAppSDK_GetOriginalFunction

Returns an address, which you can use to call the original function. For more information on using hooks, please see ["Intercepting Functions"](#).

```
[C++]
PVOID __stdcall BoxedAppSDK_GetOriginalFunction(HANDLE hHook);

[Delphi]
function BoxedAppSDK_GetOriginalFunction(hHook: THandle): Pointer; stdcall;
```

BoxedAppSDK_EnableHook

Activates or deactivates the hook. If the hook was created with `bEnable = FALSE` (see the function [BoxedAppSDK_HookFunction](#)), the hook is created but not activated. With the function `BoxedAppSDK_EnableHook`, you can do both activate a hook and deactivate it. For more information on using hooks, please see ["Intercepting Functions"](#).

```
[C++]
BOOL __stdcall BoxedAppSDK_EnableHook(
    HANDLE hHook,
    BOOL bEnable);

[Delphi]
function BoxedAppSDK_EnableHook(
    hHook: THandle;
    bEnable: BOOL): BOOL; stdcall;
```

BoxedAppSDK_UnhookFunction

Clears and removes the hook. For more information on using hooks, please see ["Intercepting Functions"](#).

```
[C++]
BOOL __stdcall BoxedAppSDK_UnhookFunction(HANDLE hHook);

[Delphi]
function BoxedAppSDK_UnhookFunction(hHook: THandle): BOOL; stdcall;
```

BoxedAppSDK_RemoteProcess_LoadLibrary

Loads the specified DLL to the specified process.

```
[C++]
HMODULE __stdcall BoxedAppSDK_RemoteProcess_LoadLibrary(
    DWORD dwProcessId,
    LPCTSTR szPath);
```



```
[Delphi]
function BoxedAppSDK_RemoteProcess_LoadLibrary(
    dwProcessId: DWORD;
    szPath: PAnsiChar): HMODULE; stdcall;

function BoxedAppSDK_RemoteProcess_LoadLibraryA(
    dwProcessId: DWORD;
    szPath: PAnsiChar): HMODULE; stdcall;

function BoxedAppSDK_RemoteProcess_LoadLibraryW(
    dwProcessId: DWORD;
    szPath: PWideChar): HMODULE; stdcall;
```

BoxedAppSDK_EmulateBoxedAppSDKDLL

BoxedAppSDK_SharedMem_Alloc

The function allocates a memory block in the [memory shared among all attached processes](#).

```
[C++]
typedef LONGLONG BOXEDAPP_SHARED_PTR;

BOXEDAPP_SHARED_PTR __stdcall BoxedAppSDK_SharedMem_Alloc(int nSize);

[Delphi]
type
    BOXEDAPP_SHARED_PTR = LargeInt;

function BoxedAppSDK_SharedMem_Alloc(dwSize: DWORD): BOXEDAPP_SHARED_PTR; stdcall;
```

BoxedAppSDK_SharedMem_Free

The function releases the memory block in the [memory shared among all attached processes](#). To allocate a memory block, use the function [BoxedAppSDK_SharedMem_Alloc](#).

```
[C++]
typedef LONGLONG BOXEDAPP_SHARED_PTR;

void __stdcall BoxedAppSDK_SharedMem_Free(BOXEDAPP_SHARED_PTR shared_ptr);

[Delphi]
type
    BOXEDAPP_SHARED_PTR = LargeInt;

procedure BoxedAppSDK_SharedMem_Free(shared_ptr: BOXEDAPP_SHARED_PTR); stdcall;
```

BoxedAppSDK_SharedMem_Lock

The function returns a pointer that can be used for accessing the memory block, allocated in the [memory shared among all attached processes](#). To allocate a memory block, use the function [BoxedAppSDK_SharedMem_Alloc](#).

```
[C++]
typedef LONGLONG BOXEDAPP_SHARED_PTR;

PVOID __stdcall BoxedAppSDK_SharedMem_Lock(BOXEDAPP_SHARED_PTR shared_ptr);
```

```
[Delphi]
type
    BOXEDAPP_SHARED_PTR = LargeInt;

function BoxedAppSDK_SharedMem_Lock(shared_ptr: DWORD): Pointer; stdcall;
```

BoxedAppSDK_SharedMem_Unlock

The function releases the pointer (not the memory!), which was used for accessing the memory block (see the [BoxedAppSDK_SharedMem_Lock](#)), allocated in the [memory shared among all attached processes](#). To allocate a memory block, use the function [BoxedAppSDK_SharedMem_Alloc](#).

```
[C++]
typedef LONGLONG BOXEDAPP_SHARED_PTR;

BOOL __stdcall BoxedAppSDK_SharedMem_Unlock(BOXEDAPP_SHARED_PTR shared_ptr);

[Delphi]
type
    BOXEDAPP_SHARED_PTR = LargeInt;

function BoxedAppSDK_SharedMem_Unlock(shared_ptr: DWORD): BOOL; stdcall;
```

BoxedAppSDK_SharedMem_CreateStreamOnSharedMem

Returns IStream with its data located in the [memory shared among all attached processes](#).

```
[C++]
HRESULT __stdcall BoxedAppSDK_SharedMem_CreateStreamOnSharedMem(LPSTREAM* ppStream);

[Delphi]
function BoxedAppSDK_SharedMem_CreateStreamOnSharedMem(var stm: IStream): HRESULT; stdcall;
```

BoxedApp SDK Options

DEF_BOXEDAPPSDK_OPTION__ALL_CHANGES_ARE_VIRTUAL

By default, this option is not set. If this option is set, all the changes, made in the file system or the registry, are stored in the virtual environment. For example:

```
[C++]
...
void main()
{
    BoxedAppSDK_Init();
    ...
    BoxedAppSDK_EnableOption(DEF_BOXEDAPPSDK_OPTION__ALL_CHANGES_ARE_VIRTUAL, TRUE);
    {
        // File 'virtual.txt' will be created as virtual

        HANDLE hFile =
            CreateFile(
                _T("virtual.txt"),
                GENERIC_WRITE,
                FILE_SHARE_READ,
                NULL,
                CREATE_NEW,
                0,
                NULL
            );
    }
    BoxedAppSDK_EnableOption(DEF_BOXEDAPPSDK_OPTION__ALL_CHANGES_ARE_VIRTUAL, FALSE);
    ...
}
```

```
[Delphi]
...
var
    fs: TFileStream;
begin
    BoxedAppSDK_Init();

    BoxedAppSDK_EnableOption(DEF_BOXEDAPPSDK_OPTION__ALL_CHANGES_ARE_VIRTUAL, true);

    // File 'virtual.txt' will be created as virtual
    fs := TFileStream.Create('virtual.txt', fmCreate or fmOpenWrite);
    fs.Free();

    BoxedAppSDK_EnableOption(DEF_BOXEDAPPSDK_OPTION__ALL_CHANGES_ARE_VIRTUAL, false);
    ...
end;
```

DEF_BOXEDAPPSDK_OPTION__EMBED_BOXEDAPP_IN_CHILD_PROCESSES

By default, this option is not set. If this option is set, all the child processes will [inherit virtual environment](#).

DEF_BOXEDAPPSDK_OPTION__ENABLE_VIRTUAL_FILE_SYSTEM

By default, this option is set. If this option is set, all the calls to the file system are intercepted by BoxedApp. Disabling this option turns off the [virtual file system](#).

DEF_BOXEDAPPSDK_OPTION__ENABLE_VIRTUAL_REGISTRY

By default, this option is set. If this option is set, all the calls to the registry are intercepted by BoxedApp. Disabling this option turns off the [virtual registry](#).

DEF_BOXEDAPPSDK_OPTION__EMULATE_OUT_OF_PROC_COM_SERVERS

By default, this option is not set. If this option is set, BoxedApp assumes the creation of COM objects that are implemented as executable files. [This allows attaching a COM server process to a shared virtual environment.](#)

DEF_BOXEDAPPSDK_OPTION__INHERIT_OPTIONS

By default, this option is not set. If this option is set, the newly created child process will get the same set of option values as the process that is creating the child process.