This document contains a description of functions and extensions for graphic and console applications working with Winflector software.

Winflector
Winflector Console
V. 3.9

Programmers manual

Table of contents

I. Introduction

The main task of Winflector and Winflector Console software is to allow a comfortable and effective use of dedicated business applications in terminal mode. Many businesses either own the source codes of the transaction systems they use, or work closely together with the developers and providers of those systems. This is why Winflector is outfitted with a set of interfaces and libraries which make it possible to further integrate the applications with the Winflector environment. Taking advantage of those features makes it easy to fix problems that cannot be resolved or are poorly resolved in any other way.

II. *Extending the functionality of applications*

Winflector enables launching of existing applications without introducing any changes to them. In some cases an extension of application capabilities is recommended through better integration with the Winflector environment. An extension of capabilities for applications working in terminal mode is accomplished by using additional Winflector functionality accessible through the *application interface*. The application interface allows calls to two classes of functions:

1. built-in functions,
2. users functions attached on the terminal side (RPC).

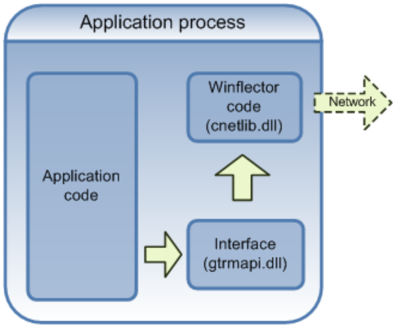
Using the *application interface*

The application interface consists of the *gtrmapi.dll* library containing functions that offer access to extended functionality of the Winflector. The *Gtrmapi.dll* library is written in C. It exports (allows external access to) all interface functions. Functions contained in the library can be made accessible to applications in two ways:

1. By static appending of the *gtrmapi.lib* library at the stage of application linking. In this case the *gtrmapi.dll* library is automatically loaded into memory as the application is started. The *gtrmapi.lib* library does not contain any static references to Winflector functions and variables. Thus, the applications linked to it can also be executed in non-terminal mode. A call of the `TApiInitialize()`function attaches the library to the Winflector or returns an error if the application works in non-terminal mode.

2. By manual loading of the library into memory using the `LoadLibrary()` function and drawing the interface function addresses using `GetProcAddress()`. Also in this case it is necessary to call the `TApiInitialize()`function before the application starts using the interface.

The names of the interface functions of the application start with the prefix `TApi`...
If the application is written in C or C++, a header *gtrmapi.h* containing all necessary function declarations has to be included. For applications written in other languages other methods of declaration and importing of the interface functions can be used. It is important to maintain appropriate types of parameters (compatible with *gtrmapi.h*) and appropriate convention of the function call (`cdecl`). The drawing below illustrates components of the application process executed in terminal mode with loaded *gtrmapi.dll* library. The arrows show how the call of the interface function is done. In some cases (such as a remote procedure call) it is necessary to call the *gte.exe* process running on the terminal as illustrated by the dashed arrow.

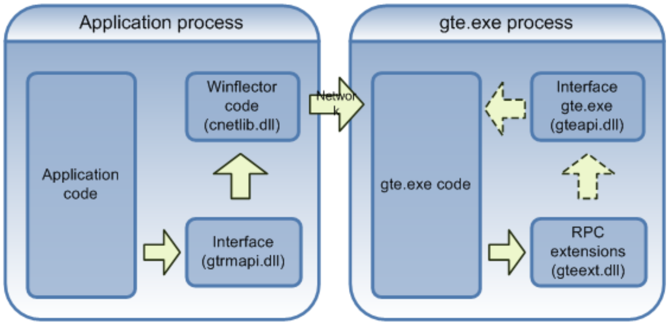


An example of the application which uses the described application interface is *testcapi.exe*.
A detailed description of the application interface functions can be found in chapter III, page 13.

Remote procedure call (RPC)

In the Winflector environment it is possible to create DLL libraries containing user functions and to attach such libraries to the *gte.exe* process being run at the terminal. Properly prepared functions can be called from the application using one of the following functions: `TApiSyncRPC()`, `TApiSyncRPC_VSR()` or `TApiAsyncRPC()`. As a consequence, some tasks can be executed entirely on the terminal. This can be helpful for instance for the handling of a non-standard identification method on the side of the terminal or for effective handling of a fiscal printer attached to the terminal.

The parameters and call convention of the functions, which will be called using the RPC scheme, must comply with special requirements. An example DLL library containing functions called via RPC is *gteext.dll*. An example of a call of a function contained in the library can be found in the application *testcapi.exe*. DLL libraries containing RPC functions are loaded to *gte.exe* address space from the application level using `TApiRemoteLoadLibraryEx()`. DLL libraries loaded in this manner have access to certain functions of *gte.exe* via the *extension interface* implemented by the *gteapi.dll* library. This library is necessary only when the user defined extension library (DLL) is to use the *gte.exe* functions. Names of the extension interface functions start with the prefix `GteApi`... , their declarations can be found in the *gteapi.h* header file. The *gteapi.dll* library can be attached to the extension library statically by linking the *gteapi.lib* library or dynamically by explicitly calling the `LoadLibrary()` function from the code initializing the extension library.



The figure shows the application process on the server and the *gte.exe* process executed on the terminal. Arrows illustrate calls between respective elements of the processes during a remote procedure call (RPC). In this case *gteext.dll* calls additional functions of *gte.exe*, thus the plot includes also a loaded *gteapi.dll* library. Since such calls are optional the corresponding arrows are dashed. The *gteext.dll* library which contains user extension functions can be called otherwise. Moreover, user extensions can be located in a number of independent DLL libraries. Each of those libraries has to be loaded before use by the `TApiRemoteLoadLibraryEx()`call .

III. Application interface functions (TApi)

The available application interface functions are described below in alphabetical order. The names of the functions start with the TApi... prefix. The functions are contained in the `gtrmapi.dll` library. An import library *gtrmapi.lib* is available. Function prototypes and all required types and constants are defined in the *gtrmapi.h* header file. All text parameters are passed to/from interface functions as UNICODE. The function interface has to be initialized before use by calling `TApiInitialize()`. The following functions are exceptions and can be called before the interface initialization:

- ✓ `TApiInitialized()`
- ✓ `TApiTerminalMode()`
- ✓ `TApiGetApiVersion()`

A `TAPI_SYSERR` code returned by any of the interface functions means that a system error occurred during its execution. Depending on the function, the error could have occurred on the server or terminal. The error code is drawn from the system by function `GetLastError()` and can be read by the interface user using one of the three functions:

- ✓ `TApiGetLastError()`
- ✓ `TApiGetLastSrvError()`
- ✓ `TApiGetLastTrmError()`

1. TApiAsyncRPC

Syntax

```
int TApiAsyncRPC( HREMMODULE hRemoteModule,
                  WCHAR *pFunName,
                  void *pCallData,
                  int callDataSize );
```

Parameters

- ✓ `hRemoteModule` – handle of the remote DLL library obtained from function `TApiRemoteLoadLibraryEx()`.
- ✓ `pFunName` – unicode name of the called DLL library function
- ✓ `pCallData` – pointer to data which will be sent to the function or `NULL`.
- ✓ `callDataSize` – size (in bytes) of the data pointed by `pCallData` or `0`, if `pCallData == NULL`.

Result

Function returns `TAPI_SUCCESS` in case of success or one of the error codes:

- ✓ `TAPI_NOTCONN` – no network connection to the terminal
- ✓ `TAPI_BADPARAMS` – bad parameters of the call

Description

The function allows asynchronous call of a users function that resides on the terminal and is appended to *gte.exe* as a DLL library. Before the call the library containing the function has to be loaded in the *gte.exe* address space by calling `TApiRemoteLoadLibraryEx()`.
An asynchronous call sends to *gte.exe* the command to call the function with given parameters and return to the calling application immediately without waiting for completion of the function. Thus, when calling a users function using `TApiAsyncRPC()`one does not obtain return information from the function. The return of `TAPI_SUCCESS` means only that the command to call the function has been sent to *gte.exe*.
Functions called using `TApiAsyncRPC()`have to be properly exported from a DLL library and declared along with a call type `cdecl`. The returned type and types of the parameters have to be consistent with the ASYNCRPCFUN type defined in gtrmapi.h. The header of the function not including the export directive and the call convention should be as follows:

```
void MyAsyncRPCFun( void *pCallData,
                    int callDataSize);
```

An example of an asynchronously called users function is the function `UserAsyncRPCBeepCallback()`from *gteext.dll* library. An example of the call of that function is given in program *testcapi.exe*.

2. TApiCheckClientProcess

Syntax

```
int TApiCheckClientProcess(
   DWORD ProcessId,
   WCHAR *pExeName,
   DWORD checkFlags );
```

Parameters

- ✓ `ProcessId` – ID of the client process returned from `TApiStartClientProcess() or TApiCheckClientProcess().` If 0 (zero) then only `pExeName` is used for process identification.
- ✓ `pExeName`– NULL or ”” (empty string) or name of the executable without the path, with or without extention. If not specified, only `ProcessId` is used for process identification. When present, both `ProcessId` and executable name are used together to identify the process.
- ✓ `checkFlags` – additional flags related to checking process status:
    - o F_PCHECK_NAMECASESENSITIVE (1) - `pExeName` should be treated as case sensitive on the client

Result

Function returns checked process ID if succesfull and specified process is still running on the client. Return value 0 means there was an error or specified process is not longer running. You can use `TApiGetLastError()` to check the type of error.

Description

The function is used to check the status of the client process started using `TApiStartClientProcess()` or any other method.

IMPORTANT! When checking the process satus in a loop, insert at least 500-1000 milliseconds delay between calls to `TApiCheckClientProcess()`. Otherwise you can overload the application-client network connection.

3. TApiCheckConnected

Syntax

```
int TApiCheckConnected( void );
```

Result

The function returns 1 if there is a connection between the application and *gte.exe* or `0` if there is no connection.

Description

The function verifies if there is a connection between the application (the process) calling the function and the corresponding *gte.exe*

4. TApiFindClientWindow

Syntax

```
int TApiFindClientWindow(
  WCHAR *pTitle1,
  WCHAR *pTitle2,
  WCHAR *pTitle3,
  DWORD flags );
```

Parameters

- ✓ `pTitle1` – window title or title fragment or ”” or NULL
- ✓ `pTitle2` – window title or title fragment or ”” or NULL
- ✓ `pTitle1` – window title or title fragment or ”” or NULL
- ✓ `flags –` additional flags specifying the meaning of the title parameters or 0 (zero). Without additional flags, first specified title parameter should be full window title. Search is made case insensitive. Other title parameters are ignored. There are following optional flags defined:

- `F_FCW_TITLECASESENSITIVE (1)` – title comparison should be made case sensitive i.e. distinguishing large and small letters. Without this flag all comparisons are case insensitive.
- `F_FCW_TITLEFRAGMENTS (2)` – means specified title parameters are title fragments not the entire title. Without this flag parameters are assumed to be full window titles.
- `F_FCW_TITLE_MATCHANY (4)` – specifies the window is found if there is a positive match on ANY of the specified titles or title fragments. Without this flag all specified fragments must match. If `F_FCW_TITLEFRAGMENTS` flag is not specified then first specified title should match.

Result

Function returns 1 if exists the top-level window on the client which fulfills the specified title criteria. Otherwise the function returns 0.

Description

Function allows for checking if exists the window on the client with the title which meets the specified conditions. It is possible to check using full window title, title fragment or title fragments. Comparison can be made case insensitive (default) or case sensitive.

To specify several search flags specify a sum of their values such as 1+2 (3) means both `F_FCW_TITLECASESENSITIVE` and `F_FCW_TITLEFRAGMENTS` should be used.

Sample

```
//  returns 1 if “Window 1” exists on the client (case insensitive)
TApiFindClientWindow(“Window 1”,NULL,NULL,0)

//  returns 1 if “Window 1” or “Window 2” exists on the client (case insensitive)
TApiFindClientWindow(“Window 1”,”Window 2”,NULL,
F_FCW_TITLE_MATCHANY)

//  returns 1 if  window with title containing “MyDocument” and “Notepad”
// exists on the client (case-insensitive)
```

TApiFindClientWindow(“MyDocument”,”Notepad”,NULL, F_FCW_TITLEFRAGMENTS)

5. TApiGetApiVersion

Syntax

`int TApiGetApiVersion( GTRMVERSION *pApiVer );`

Parameters

- ✓ `pApiVer` – pointer to `GTRMVERSION` structure. This is where the result of the function is to be stored.

Result

The function fills the `GTRMVERSION` structure and returns `0`.

Description

The function returns in the `GTRMVERSION` structure information on the interface, that is the *gtrmapi.dll* library version. The version should identical or older than version of the used Winflector. If this is not the case, the API initialization using `TApiInitialize()` will fail. This means that Winflector server should be identical or newer than *gtrmapi.dll* used

6. TApiGetAuthenticationMode

Syntax

`int TApiGetAuthenticationMode( void );`

Result

Returns the mode of authentication used for user logon:
`TAPI_AUTH_TERMINAL –` authentication using users defined via Wfserver.exe application. Application is run under the user used to start Wfserver.exe server.
`TAPI_AUTH_WINDOWS–` authentication using Windows user accounts and the application is run under the account of the specified user.

Opis

Function returns the information regarding the authentication method used to log the user into Winflector server.

7. TApiGetClientDir

Syntax

`wchar_t * TApiGetClientDir( int dirType );`

Result

The function returns a pointer to the name of the directory (path) on the terminal side or `NULL` in case of an error. Non-empty directory name is always terminated with backslash (\) character.

Description

The function allows to get the name of the directory (path) specified by the *dirType* parameter on the terminal. At this time the following directories are being handled:

- `TAPI_DIRTYPE_GTE (1)` – directory of the *gte.exe* program.
- `TAPI_DIRTYPE_DESKTOP (2)` – `SHGetFolderPath(CSIDL_DESKTOP)`
- `TAPI_DIRTYPE_DESKTOPDIRECTORY (3)` – `SHGetFolderPath(CSIDL_DESKTOPDIRECTORY)`
- `TAPI_DIRTYPE_COMMONDESKTOPDIRECTORY (4)` – `SHGetFolderPath(CSIDL_COMMON_DESKTOPDIRECTORY)`
- `TAPI_DIRTYPE_APPDATA (5)` – `SHGetFolderPath(CSIDL_APPDATA)`
- `TAPI_DIRTYPE_COMMONAPPDATA (6)` – `SHGetFolderPath(CSIDL_COMMON_APPDATA)`
- `TAPI_DIRTYPE_LOCALAPPDATA (7)` – `SHGetFolderPath(CSIDL_LOCAL_APPDATA)`
- `TAPI_DIRTYPE_PROFILE (8)` – `SHGetFolderPath(CSIDL_PROFILE)`

- `TAPI_DIRTYPE_MYDOCUMENTS (9)` – `SHGetFolderPath(CSIDL_MYDOCUMENTS)`

The returned pointer points at the dynamically allocated memory block which should be released after use by calling the `TApiMemGlobalFree()`function.

8. TApiGetClientId

Syntax
`int TApiGetClientId( void );`

Result

Function returns the unique id of the Winflector server client session.

Description

Unique client Id is assigned to the process when it is started by Winflector server. Secondary processes started by the initial process get their own unique client ids. Client/process ids are valid until process terminates. Then, id can be reused for identifying the new process. Note that client id value can be bigger then maximum number of devices returned from TApiGetMaxDevices() function.

9. TApiGetClientMode

Syntax
`int TApiGetClientMode( void );`

Result

Function returns the type of Winflector client connected to application.

Description

There are two types of Winflector client. Android and HTML5 client creates virtual desktop (separate window) which is used to present all application windows to the user. In these cases function returns TAPI_CLIENT_DESKTOP (0). Second client type integrates application windows directly with client desktop. For such clients function returns TAPI_CLIENT_MULTIWINDOW (1).

10. TApiGetComputerNameEx

Syntax

```
wchar_t * TApiGetComputerNameEx( int nameType );
```

Result

The function returns a pointer to the name of the client computer or `NULL` in case of an error.

Description

The function allows to get the name of the computer where Winflector client executes. The `name`*`Type`* parameter is an integer in range 0-8 and determines the requested name format. On the terminal name is retrieved using `GetComputerNameEx()` Windows API. Name formats are documented in MSDN documentation for `GetComputerNameEx()` API. The most frequently used name format is 0 (zero) which returns the NetBIOS name of the client computer.

The returned pointer points at the dynamically allocated memory block which should be released after use by calling the `TApiMemGlobalFree()`function.

11. TApiGetDomainName

Syntax

```
int TApiGetDomainName( WCHAR *pDomainName );
```

Parameters

- ✓ `pDomainName` – pointer to the buffer which can hold `TAPI_MAXDOMAINNAME` of UNICODE characters (of the `sizeof(wchar_t)` size). In case of success, the domain name of the

Winflector server user, who has launched the application, is saved to the buffer with a zero appended at its end.

Result

The function returns `TAPI_SUCCESS` for success or one of the following error message:

- ✓ `TAPI_NOTCONN` – no network connection to the terminal,
- ✓ `TAPI_BADPARAMS` – wrong call parameter, for instance `pDomainName == NULL`.

Description

The function allows the reading of the domain name of the Winflector server user who has launched the application.

12. TApiGetExpirationDate

Syntax

```
int TApiGetExpirationDate( void );
```

Result

The function returns the date of expiry of the evaluation version of the Winflector server or 0 if it is a production version. Bits 16-31 contain the year, bits 8-15 the month, and bits 0-7 the day of the expiry date.

Description

The function serves to check whether the application works under control of an evaluation version of the Winflector server and what is the date of expiry of that version. This information can be used for instance to issue a reminder about the end of evaluation and the required purchase of a production version of the software.

13. TApiGetFileFromTerminal

Syntax

```
int TApiGetFileFromTerminal(  WCHAR *pSrvFileName,
                              WCHAR *pTrmFileName,
                              REMFILEERROR *pFlError,
                              DWORD flags );
```

Parameters

- ✓ `pSrvFileName` – unicode file name on the server, which is to be used to save the file transferred from the terminal.
- ✓ `pTrmFileName` – unicode file name of the file that is to be transferred from the terminal.
- ✓ `pFlError` – pointer to `REMFILEERROR` structure in which additional information will be saved in case of an error. If `NULL` is given no additional diagnostics will be available. The field `pFlError->error` contains a value identical with the value returned by the function. The field `pFlError->nbytes` contains information about the number of bytes of the file transferred.
- ✓ `flags` – additional flags concerning the file transfer.
  The flag `TAPI_F_FILEOVERWRITE` means that the result file is to be overwritten in case if it already exists. No `TAPI_F_FILEOVERWRITE` flag means that the existance of the result file will constitute an error and the transfer will not take place.

Result

The function returns `TAPI_SUCCESS` if the transfer has been successfull or one of the following error codes:

- ✓ `TAPI_NOTCONN` – no connection to the terminal (gte.exe).
- ✓ `TAPI_BADPARAMS` – bad parameters, e.g. one of the required file names is missing.
- ✓ `TAPI_SYSERR` – a system error has occured.
  Fields `pFlError->lastSrvSysError` and `pFlError->lastTrmSysError` contain information about the type

of the error returned by the function `GetLastError()` of the Windows system on the server and on the terminal.

Description

The function allows to transfer from the terminal a file named `pTrmFileName` and to save it on the server using `pSrvFileName` as the file name. The name of the file on the server should be given as seen by the server system, the name of the file on the terminal should be as seen by the terminal system.
In case the `TAPI_SYSERR` error one of the following functions: `GetLastError()`, `GetLastSrvError()` or `GetLastTrmError()` can be used to obtain additional information about the error.

14. TApiGetGroups

Syntax

```
int TApiGetGroups(DWORD *pNGroups,WCHAR ***pGrpArray);
```

Parameters

- ✓ `pNGroups` – pointer to DWORD variable where number of the groups read is to be placed.
- ✓ `pGrpArray` – pointer to WCHAR** variable which will be set to a pointer of allocated memory block of type WCHAR **.Allocated block holds the pointers to group names and the names themselves.

Result

The function returns `TAPI_SUCCESS` for success or one of the following error message:

- ✓ `TAPI_NOTCONN` – no network connection to the terminal,
- ✓ `TAPI_BADPARAMS` – wrong call parameter, for instance `pNGroups == NULL`.

Opis

Fuction allows reading of the names of the groups to which the Winflector user belongs. The function can be used only if the user was authenticated using Windows accounts, which means the TApiGetAuthenticationMode() function returns `TAPI_AUTH_WINDOWS`. The pointer returned in **pGrpArray* points to allocated memory block. It is important to free it when no longer needed using `TApiMemGlobalFree()` function..

Sample

```
DWORD nGroups = 0;
WCHAR **GrpTab = NULL;
if( TApiGetGroups(&nGroups,&GrpTab) == TAPI_SUCCESS )
{
    if( GrpTab )
    {
              for( int grp=0; grp<nGroups; grp++)
                        wprintf(L"Group  %d: [%s]\n",grp,GrpTab[grp]);
              TApiMemGlobalFree(GrpTab);     // free memory
    }
}
```

15. TApiGetLastError

Syntax

`DWORD TApiGetLastError( void );`

Result

The function returns the code of the last system error which occured on the terminal or, if there was no error on the terminal, the code of the last error on the server. `0` is returned if there were no errors.

Description

If an error code `TAPI_SYSERR` was received as a result of an interface function call, the `TApiGetLastError()` function will return the system error code obtained from `GetLastError()`. The function will return the error code for an error which occurred on both the terminal or the server. If errors have occurred on both systems the terminal error code will be returned.

16. TApiGetLastSrvError

Syntax

```
DWORD TApiGetLastSrvError( void );
```

Result

The function returns the error code of the last system error which occurred on the server or `0`.

Description

If a `TAPI_SYSERR` error code was received as a consequence of an interface function call and the system error occurred on the server, the `TApiGetLastSrvError()` function will return the error code obtained from `GetLastError()`. The function returns `0` if there was no error on server.

17. TApiGetLastTrmError

Syntax

```
DWORD TApiGetLastTrmError( void );
```

Result

The function returns the error code of the last system error which occurred on the terminal or `0`.

Description

If a `TAPI_SYSERR` error code was received as a consequence of an interface function call and the system error occurred on the terminal, the `TApiGetLastTrmError()` function will return the error code obtained from `GetLastError()`. The function returns `0` if there was no error on the terminal.

18. TApiGetMaxDevices

Syntax

```
int TApiGetMaxDevices( void );
```

Result

Function return the maximum number of devices supported (licensed) by Winflector server.

Description

Function can be used to check the maximum number of devices which can be used with the cooperating Winflector server.

19. TApiGetRemoteIPAddr

Syntax

```
unsigned TApiGetRemoteIPAddr( void );
```

Result

The function returns the IP address of the terminal or NAT implementing router which provides routing to the terminal computer. Upper eight bits of the result contain the most significant part of the IP address, etc.

Description

The function allows to obtain the IP address of the terminal.

20. TApiGetRemoteIPAddrStr

Syntax

```
int TApiGetRemoteIPAddrStr( WCHAR *pIPStrBuf );
```

Result

The function places in *pIPStrBuf* the IP address of the terminal or NAT implementing router which provides routing to the terminal computer. The string representing remote address is terminated with 0 (zero). Buffer size should be at least 16 characters (32 bytes). If succesfull, function returns 0.

Description

The function allows to obtain text of the IP address of the terminal.

21. TApiGetRemoteIPPort

Syntax

```
unsigned TApiGetRemoteIPPort( void );
```

Result

The function returns the port number of the IP connection on the terminal side.

Description

The function allows to obtain the number of the IP port created by *gte.exe* on the terminal side of the connection to communicate with the application.

22. TApiGetRemoteIPPortStr

Syntax

```
Int TApiGetRemoteIPPortStr( WCHAR *pPortStrBuf );
```

Result

The function places into *pPortStrBuf* the port number of the IP connection on the terminal side. The string representing remote port is terminated with 0 (zero). Buffer size should be at least 6 characters (12 bytes). If succesfull, function returns 0.

Description

The function allows to obtain the text of a number of the IP port created by *gte.exe* on the terminal side of the connection to communicate with the application.

23. TApiGetSrvOSVer

Syntax

`int TApiGetSrvOSVer( TAPI_OSVERSIONINFO *pOSVInfo );`

Parameters

- ✓ `pOSVInfo` – pointer to `TAPI_OSVERSIONINFO` structure which is used to save information about the version of the operating system of the server obtained from the `GetVersionEx()` Windows function.

Result

The function returns `TAPI_SUCCESS` or `TAPI_NOTCONN` if there is no connection to the terminal.

Description

The function allows reading of information which identifies the version of the operating system of the server.

24. TApiGetTrmOSVer

Syntax

`int TApiGetTrmOSVer( TAPI_OSVERSIONINFO *pOSVInfo );`

Parameters

- ✓ `pOSVInfo` – pointer to `TAPI_OSVERSIONINFO` structure which is used to save information about the version of the operating system of the terminal obtained from the `GetVersionEx()` Windows function.

Result

The function returns `TAPI_SUCCESS` or `TAPI_NOTCONN` if there is no connection to the terminal. When Winflector client is not run on Windows, filed `pOSVInfo->dwPlatformId` contains one of the following values defined in *gtrmapi.h*.

```
VER_PLATFORM_LINUX   (101)    // Linux operating system
VER_PLATFORM_ANDROID (151)    // Android operating system
VER_PLATFORM_IOS     (161)    // IOS operating system
VER_PLATFORM_MACOS   (171)    // MacOS operating system
```

Description

The function allows reading of information which identifies the version of the operating system of the terminal.

25. TApiGetUserName

Syntax

```
int TApiGetUserName( WCHAR *pUserName );
```

Parameters

- ✓ `pUserName` – pointer to the buffer which can hold `TAPI_MAXUSERNAME` of UNICODE characters (of the `sizeof(wchar_t)` size). In case of success, the username of the Winflector server user, who has launched the application, is saved to the buffer with a zero appended at its end.

Result

The function returns `TAPI_SUCCESS` for success or one of the following error message:

- ✓ `TAPI_NOTCONN` – no network connection to the terminal,

- ✓ `TAPI_BADPARAMS` – wrong call parameter, for instance `pUserName == NULL`.

Description

The function allows the reading of the username of the Winflector server user who has launched the application.

26. TApiGetUserNameEx

Syntax

```
wchar_t * TApiGetUserNameEx( int nameType );
```

Result

The function returns a pointer to the name of the system user using client computer or `NULL` in case of an error.

Description

The function allows to get the name of the system user using computer where Winflector client executes. The `name`*`Type`* parameter is an integer in range 0-12 and determines the requested name format. On the terminal name is retrieved using `GetUserNameEx()` Windows API. Name formats are documented in MSDN documentation for `GetUserNameEx()` API.

The returned pointer points at the dynamically allocated memory block which should be released after use by calling the `TApiMemGlobalFree()`function.

27. TApiGetTrmVersion

Syntax

```
int TApiGetTrmVersion( GTRMVERSION *pCallerVer,
                       GTRMVERSION *pCnetlibVer );
```

Parameters

- ✓ `pCallerVer` – reserved – should always be `NULL`.

- ✓ `pCnetlibVer` – pointer to `GTRMVERSION` structure which will be used to save the result.

Result

The function returns `TAPI_SUCCESS`.

Description

The function allows the reading of the version of *cnetlib.dll* library which is used be the application. This version is in accordance with the version of the used Winflector server.

28. TApiHwndToNetId

Syntax

`int TApiHwndToNetId( HWND hWnd );`

Parameters

- ✓ `hWnd` – handle of the application window.

Result

Network identifier of the window or `0`.

Description

The function allows the converting of the application window handle into its network identifier. The network identifier can be passed to a function executed on the terminal and converted into a corresponding system window on the terminal using the `GteApiNetIdToHwnd()`function. The function returns `0` if `hWnd` is a window handle which does not have a direct counterpart on the terminal side.

29. TApiHwndToRemotedNetId

Syntax

`int TApiHwndToRemotedNetId( HWND hWnd );`

Parameters

- ✓ `hWnd` – handle of the application window.

Result

Network identifier of the window or its parent or `0`.

Description

The function allows the converting of the application window handle into its network identifier. The network identifier can be passed to a function executed on the terminal and converted into a corresponding system window on the terminal using the `GteApiNetIdToHwnd()`function. Handles of windows run directly on the terminal (top level) are converted into the network identifiers of those windows. Handles of windows which do not have direct counterparts on the terminal are converted into the identifiers of the top level windows whose child is the `hWnd` window.

30. TApiInitialize

Syntax

```
int TApiInitialize( void );
```

Result

The function returns `TAPI_SUCCESS` or one of the error codes:

- ✓ `TAPI_NOCNETLIB` – the *cnetlib.dll* library was not found – most probably the application process was not launched in the terminal mode.
- ✓ `TAPI_BADAPIVERSION` – the first four digits of the application interface version (*gtrmapi.dll*) do not correspond to the first four digits of the Winflector software version (*cnetlib.dll*). Please verify if correct DLL libraries have been copied.
- ✓ `TAPI_CANTIMPORTFUN` – the address of one of the API functions within *cnetlib.dll* can not be imported.

Description

The function initializes internal structures of the application interface contained in *gtrmapi.dll*. During initialization functions of *gtrmapi.dll* are linked to the auxiliary functions contained in *cnetlib.dll*. The interface should be initialized before the first call of the API functions with the exception of three functions which can be called before API initialization:

- ✓ `TApiInitialized()`
- ✓ `TApiTerminalMode()`
- ✓ `TApiGetApiVersion()`

31. TApiInitialized

Syntax

```
int TApiInitialized( void );
```

Result

The function returns `1` if the application interface has already been successfully initialized using `TApiInitialize()` or `0` if this is not the case.

Description

The function can be used to verify at any time if the application interface has been initialized and the API functions can be called.

32. TApiKillClientProcess

Syntax

```
int TApiKillClientProcess(
  DWORD ProcessId,
  WCHAR *pExeName,
  DWORD killFlags );
```

Parameters

- ✓ `ProcessId` – ID of the client process returned from `TApiStartClientProcess() or TApiCheckClientProcess().` If 0 (zero) then only `pExeName` is used for process identification.

- ✓ `pExeName`– NULL or ”” (empty string) or name of the executable without the path, with or without extention. If not specified, only `ProcessId` is used for process identification. When present, both `ProcessId` and executable name are used together to identify the process.
- ✓ `killFlags` – additional flags related to killing the process:
  - o F_PKILL_NAMECASESENSITIVE (1) - `pExeName` should be treated as case sensitive on the client

Result

Function returns terminated process ID if specified process was found and terminated on the client. Return value 0 means there was an error or specified process was not found. You can use `TApiGetLastError()` to check the type of error.

Description

The function is used to kill (terminate) the client process started using `TApiStartClientProcess()` or any other method.

33. TApiMemGlobalAlloc

Syntax

`void *TApiMemGlobalAlloc( unsigned size );`

Parameters

- ✓ `size` – size of the memory block that is to be allocated.

Result

The function returns the pointer to the allocated memory block of the size `size` or `NULL` if the memory can not be allocated.

Description

The function allows dynamical allocation of memory for the application. The allocated memory block is initialized by setting it to zero. The memory allocated using the function `TApiMemGlobalAlloc()` must be released using the `TApiMemGlobalFree()`function.

34. TApiMemGlobalFree

Syntax

```
void TApiMemGlobalFree( void *ptr );
```

Parameters

- ✓ `ptr` – pointer to the memory block allocated using the `TApiMemGlobalAlloc()` function or obtained from the `TApiSyncRPC_VSR()` function.

Description

The function releases memory dynamically allocated by `TApiMemGlobalAlloc()`. If the function `TApiSyncRPC_VSR()` returned a non-zero block size, this memory block must also be released using the described function.

35. TApiNetIdToHwnd

Syntax

```
HWND TApiNetIdToHwnd( WNDNETID netid );
```

Parameters

- ✓ `netid` – network identifier of a window obtained from the function `TApiHwndToNetId()`, `TApiHwndToRemotedNetId()` or `GteApiHwndToNetId()`.

Result

The handle of the Windows window or `NULL`.

Description

The function converts the network identifier of a window into the corresponding window of the Windows system. The network identifier of a window will be in most cases transferred to an application from RPC functions of the extension libraries attached to *gte.exe*. The RPC function can convert on the terminal the actual handle of the Windows system window into the network identifier using `GteApiHwndToNetId()` and return it to the application. The application can find the window on the server which corresponds to the terminal window, by converting the network identifier into the server window handle using the described function. The function returns `NULL` if the Windows window described by the network identifier does not exist anymore.

36. TApiPutFileToTerminal

Syntax

```
int TApiPutFileToTerminal( WCHAR *pSrvFileName,
                           WCHAR *pTrmFileName,
                           REMFILEERROR *pFlError,
                           DWORD flags );
```

Parameters

- ✓ `pSrvFileName` – unicode name of the file on the server that is to be sent to the terminal.
- ✓ `pTrmFileName` – unicode name of the file on the terminal that is to be used to save the file transferred from the server.
- ✓ `pFlError` – pointer to `REMFILEERROR` structure in which additional information will be saved in case of an error. If `NULL` is given no error diagnostics will be available. The value returned in the `pFlError->error` field is identical with the value returned by the function. The number of bytes of the file transferred is returned in the `pFlError->nbytes` field.
- ✓ `flags` – additional flags concerning the file transfer. The flag `TAPI_F_FILEOVERWRITE` means that the destination file is to be overwritten if it exists. If the `TAPI_F_FILEOVERWRITE` flag is not

specified, the existence of the destination file will cause an error and the file transfer will not take place.

Result

The function returns `TAPI_SUCCESS` if the transfer has been successful or one of the error codes:

- ✓ `TAPI_NOTCONN` – no connection to the terminal (gte.exe).
- ✓ `TAPI_BADPARAMS` – bad parameters, e.g. one of the required file names is missing.
- ✓ `TAPI_SYSERR` – a system error has occured. The fields `pFlError->lastSrvSysError` and `pFlError->lastTrmSysError` contain information about the type of the error as returned by the Windows system function `GetLastError()` on the server and terminal, respectively.

Description

The function allows to send to the terminal the file named `pSrvFileName` and to save it on the terminal as `pTrmFileName`. The file names on the server and terminal should be given as seen by the server and terminal system, respectively. In case of an `TAPI_SYSERR` error, functions `GetLastError()`, `GetLastSrvError()` or `GetLastTrmError()` can be used to read additional error diagnostics.

37. TApiRaiseFinalError

Syntax

```
int TApiRaiseFinalError( wchar_t *pDescription,
                         wchar_t *pFunction,
                         int ival1,
                         int ival2 );
```

Parameters

- ✓ `pDescription` – error description as a unicode string.
- ✓ `pFunction` – unicode name of the function which had an error.

- ✓ `ival1`, `ival2` – additional diagnostics values which will be saved and displayed with the error message.

Result

The function does not return to the parent process, thus it does not return any value.

Description

The function allows notification of a terminating error. Following actions are part of the terminating error handling:

- The error message and other error information is saved to the log file on the server. The log file is located in the *applogs* subdirectory of the Winflector server directory and is called *tapplog.txt*.
- If there is a connection between *gte.exe* and the application, the error information is sent to *gte.exe* and an error message is displayed on the terminal.
- The error message is saved to the log file on the terminal (*tcllog.txt*).
- A forced termination of the *gte.exe* process and of the application process with the exit code larger than 0 takes place.

As seen from the description above the function concludes the execution of the program. The control is not returned to the parent process and the function does not return any value.

38. TApiRegisterAppCallback

Syntax

```
int TApiRegisterAppCallback(
      WINFLECTOREVTPROC pEvtCallbackProc,
      void *pCallbackData,
      WINFLECTOREVTPROC *pPrevProc,
      Void **pPrevData );
```

Parameters

- ✓ `pEvtCallbackProc` – pointer to application-defined function of type WINFLECTOREVTPROC which is to be called when supported Winflector event happen. Specify NULL to cancel callback registration.
- ✓ `pCallbackData` – pointer that will be passed as `ptr` parameter to `pEvtCallbackProc` function or NULL.
- ✓ `pPrevProc` – pointer to location that receives previous value set as `pEvtCallbackProc`. This may be `NULL`.
- ✓ `pPrevData` – pointer to location that receives previous value set as `pCallbackData`. This may be `NULL`.

Result

Function returns 1 on success, 0 on failure.

Description

This function allows registering of the application-defined callback function that will be called when some events related to the remote application execution happen. The registered function must be of type WINFLECTOREVTPROC that is defined as follows:

```
typedef DWORD (CALLBACK* WINFLECTOREVTPROC)(int nEvt, void *ptr);
```

First parameter of this callback function specifies the type of the event that triggered callback call. The second (`ptr`) parameter depends on the event type. For `WFLEVT_NORMALEXIT and WFLEVT_CRITICALERROR` events callback function should return 0 or the number of milliseconds to wait before continuing processing. At the moment the following event types are signaled by Winflector library by calling the callback function:

- `WFLEVT_NORMALEXIT` – application is going to finish in normal way (no error). The application will be closed when callback function returns. `Ptr` parameter specifies pointer registered as `pCallbackData.`
- `WFLEVT_CRITICALERROR` – critical error happened in Winflector library and application cannot continue execution. The application will be terminated right after the callback function returns. When handling this type

of callback event avoid using functions that display graphics or text. `Ptr` parameter specifies pointer registered as `pCallbackData.`

- `WFLEVT_ASYNCGTEEVENT –` client extension DLL has called `GteApiRaiseAsyncAppEvent`() function on the client. Callback function is always called in the context of main application thread, *`Ptr`* parameter is a pointer to GTEEVENTINFO structure that contains data received from `GteApiRaiseAsyncAppEvent`() function. Data is located in *`clntData`* structure field while its length in bytes in *`clntDataSize`* field. Callback function should always return 0.
- `WFLEVT_SYNCGTEEVENT -` client extension DLL has called `GteApiRaiseSyncAppEvent`() function on the client. Callback function is always called in the context of main application thread, *`Ptr`* parameter is a pointer to GTEEVENTINFO structure that contains data received from `GteApiRaiseSyncAppEvent`() function. Data is located in *`clntData`* structure field while its length in bytes in *`clntDataSize`* field. Callback function can return any DWORD value which will be returned to the client and made accessible by `GteApiRaiseSyncAppEvent`() function.

Attention! For `WFLEVT_NORMALEXIT and WFLEVT_CRITICALERROR` events the registered callback function is called in the context of the thread that triggered error or is closing the application. In general, you cannot assume this is a thread that registered callback function by calling `TApiRegisterAppCallback()` These two events must always be handled – return 0 from callback function if you are not interested in handling these events.

39. TApiRemoteFreeLibrary

Syntax

`BOOL TApiRemoteFreeLibrary( HREMMODULE hRemoteModule );`

Parameters

✓ `hRemoteModule` – handle of the remote extension library obtained from `TApiRemoteLoadLibraryEx()`.

Result

The function returns `1` if the library has been successfully released or `0` otherwise.

Description

The function allows to release (remove from the *gte.exe* address space) a DLL library loaded earlier using the `TApiRemoteLoadLibrary()`function. After the library is released the RPC functions contained in the library can not longer be used. In case of an error additional diagnostics information can be obtained using the function `TApiGetLastTrmError()`.

40. TApiRemoteLoadLibraryEx

Syntax

```
HREMMODULE TApiRemoteLoadLibraryEx( WCHAR *pFileName,
                                    DWORD dwFlags );
```

Parameters

- ✓ `pFileName` – unicode name of the DLL remote extension library as seen on the terminal by the *gte.exe* process.
- ✓ `dwFlags` – a parameter transferred directly to the `LoadLibraryEx()` Windows function on the terminal. The value used most often is `0`.

Result

The function returns the handle to the remotely loaded DLL library or `0` in case of an error.

Description

The function allows to load an extension library of a given name to the address space of the *gte.exe* process running on the terminal. Extension libraries usually contain user functions which can be remotely called by the application using one of the following RPC calls: `TApiAsyncRPC()`, `TApiSyncRPC()` or `TApiSyncRPC_VSR()`. If the loaded extension library will not be used any more it can be released using the function `TApiRemoteFreeLibrary()`. In case of an error additional diagnostics information can be obtained from the `TApiGetLastTrmError()`function.

41. TApiRemotePrintFile

Syntax

```
int TApiRemotePrintFile( WCHAR *pFileName,
                         WCHAR *pPrinterName,
                         WCHAR *pDatatype );
```

Parameters

- ✓ `pFileName` – unicode name of the file containing data to be printed.
- ✓ `pPrinterName` – unicode name of the printer connected to the terminal which is to be used for the printout. The printer name can contain the postfix `@WFC` which will be removed before opening the printer on the terminal. For the default terminal printer printer name `L”DEFPRN”` can be used.
- ✓ `pDataType` – file data type. At this time only `L”TEXT”` format is supported.

Result

The function returns `TAPI_SUCCESS` if printing was successful or one of the following error codes:

- ✓ `TAPI_NOTCONN` – no connection to the terminal (gte.exe).
- ✓ `TAPI_BADPARAMS` – bad parameters.
- ✓ `TAPI_SYSERR` – a system error occurred. Additional information can be obtained from the functions `TApiGetLastSrvError()` and `TApiGetLastTrmError()`.

Description

The function allows printing of the contents of a given file on a printer connected to the terminal. The function returns after the file is sent to the printer. At this time only `L”TEXT”` data format is supported.

42. TApiRemoteShellExecute

Syntax

```
int TApiRemoteShellExecute( HWND hWnd,
              WCHAR *pOperation,
              WCHAR *pFile,
              WCHAR *pParameters,
              WCHAR *pDirectory,
              int nShowCmd );
```

Parameters

- ✓ `hWnd` – handle to server-side window created by applicatrion or NULL. This handle will be converted to corresponding client-side window handle. `ShellExecute`() use this window handle as a parent for eventuall user interface or message boxes.
- ✓ `pOperation` – see `ShellExecute`() documentation in MSDN
- ✓ `pFile` – see `ShellExecute`() documentation in MSDN
- ✓ `pParameters` – see `ShellExecute`() documentation in MSDN
- ✓ `pDirectory` – see `ShellExecute`() documentation in MSDN
  `nShowCmd` – see `ShellExecute`() documentation in MSDN

Result

Function returns value returned by `ShellExecute`() API executed on the clinet. See MSDN documentation for details.

Description

Function allows for passing parameters and executing `ShellExecute()` API on the client computer (terminal). You can use it for instance for opening client document using the default registered application such as opening TXT file in Notepad or WWW link in default browser. See MSDN documentation for details.

Sample

```
//  opens internet site in client default browser
Res = TApiRemoteShellExecute(NULL,  L"open",
                        L”http://www.winflector.com”,
                        NULL, NULL, SW_SHOWNORMAL);

//  opens the specified  spreadsheet in client application
//  registered for opening XLS files (e.g. Excel)
Res = TApiRemoteShellExecute(NULL,  L”open”,
                        L”c:/documents/prices.xls”,
                        NULL, NULL, SW_SHOWNORMAL);
```

43. TApiSendUpdates

Syntax

`int TApiSendUpdates( HWND hWnd );`

Parameters

- ✓ `hWnd` – handle of the window which is to be updated or `NULL` for updating all windows of the application.

Result

The function returns `TAPI_SUCCESS` or `TAPI_NOTCONN` if there is no connection to the terminal.

Description

To optimize network transmission in the Winflector environment the changes made by the application in windows of the Windows system are sent to the terminal with a small delay. The function `TApiSendUpdates()` allows forced immediate sending of changes for window `hWnd` or for all windows of the application. A forced sending of the changes can be useful for instance before calling an RPC function, which assumes that certain information has already been displayed on the terminal window.

44.TApiSetClientParam

Syntax

```
int TApiSetClientParam( DWORD ParamCode,
                        DWORD ParamValue );
```

Parameters

- ✓ `ParamCode` – code determining the parameter to be changed. Following codes can be used:

  `TAPI_CLTPARAM_DRAGOPTIMIZATION (1) –` turns on (1) or off (2) dragging optimization.
  `TAPI_CLTPARAM_DRAGMOVEINTERVAL (2) –` sets the milliseconds interval that determines how often WM_MOUSEMOVE messages should be sent to application during dragging (while keeping left mouse button pressed). The interval can be 10 – 150 milliseconds, the default value is 30 milliseconds.

- ✓ `ParamValue` –value of the parameter specified as `ParamCode`.

Result

Function returns TAPI_SUCCESS (0) or TAPI_NOTCONN (11) if application is not connected to the client.

Description

Function allows dynamic setting of the client (vtm.exe/gte.exe) parameters from the application level.

45. TApiSetClientWindowTopMost

Syntax

```
int TApiSetClientWindowTopMost(
  HWND hSrvWnd,
  int bTopMost );
```

Parameters

- ✓ `hSrvWnd` – handle of existing, top-level, server-side application window
- ✓ `bTopMost` – value 1 means make client copy of the window TOPMOST, value 0 means make client copy of the window non-TOPMOST (remove WS_EX_TOPMOST style bit).

Result

Function returns TAPI_SUCCESS (0) or TAPI_BADPARAMS (5) in case window is not a top-level window.

Description

You can use this function, to set or remove the TOPMOST property (WS_EX_TOPMOST window style) of the client Window that corresponds to the application window identified by the `hSrvWnd` handle. Function does not modify the style of the application (server-side) window.

46. TApiSetDiscTmt

Syntax

```
DWORD TApiSetDiscTmt( DWORD timeout );
```

Parameters

- ✓ `timeout` – timeout time in seconds after which the application will disconnect the client station (terminal) in case it has lost connection to the application due to a network error or shut down. The range of the parameter is 20-10000. The 0 value means return to the default setting, the 65535 value switches the mechanism of active checking the connection off.

Result

The function returns the previous value of the parameter or `0` for an incorrect value of the timeout parameter.

Description

The function allows to change the maximum time after which the application will disconnect a client station (terminal) in case it has lost connection to the application due to a network error or shut down. The disconnect can occur earlier if a mechanism of the TCP/IP protocol signals the loss of connection.

47. TApiSetPDFPrintParams

Syntax

```
int TApiSetPDFPrintParams(
  WCHAR *pClientDirectoryName,
  WCHAR *pClientPdfFileName,
  DWORD flags );
```

Parameters

- ✓ `pClientDirectoryName` – name of the client directory where PDF printout file should be placed. NULL value means the default directory should be used.
- ✓ `pClientPDFFileName` – name of the clinet file to use for PDF printout. NULL value means default Winflector-generated name should be used.
- ✓ flags – additional flags related to PDF printout savin on the client. By default, previous parameters are used to initialize the file save dialog presented on the clinet to save PDF file. By passing the F_PDF_NOFILEDIALOG (1) option, you request to save the file immediately without presenting the file save dialog.

Result

Function returns `0` if PDF parameters were succesfully set.

Description

The function is used to controll the way PDF printout is saved on the client computer. It allows setting the save directory, PDF file name and option to ommit the save file dialog.

48. TApiStartClientProcess

Syntax

```
int TApiStartClientProcess(
  WCHAR *pCommandLine,
  WCHAR *pWorkDirectory,
  DWORD flags );
```

Parameters

- ✓ `pCommandLine` – executable path or name and (optionally) parameters
- ✓ `pWorkDirectory` – path of the client directory to be used as default for new process. If NULL or ”” process will be cretated with the same working directory as current gte.exe (client executable).
- ✓ flags – reserved - additional flags related to process creation, specify 0 (zero).

Result

Function returns PID (process id) of the process created on the client machine. The returned PID can be used with `TApiCheckClientProcess() and TApiKillClientProcess()` to check the status of created process or terminate it. Return value 0 means no process was created – use `TApiGetLastError()` to check the type of error.

Description

The function is used to start new process on the client (terminal) machine.

49. TApiSyncRPC

Syntax

```
int TApiSyncRPC(  HREMMODULE hRemoteModule,
                  WCHAR *pFunName,
                  void *pCallData,
                  int callDataSize,
                  void *pResData,
                  int *pMaxResDataSize );
```

Parameters

- ✓ `hRemoteModule` – handle of a remote DLL library obtained from the function `TApiRemoteLoadLibraryEx()`.
- ✓ `pFunName` – unicode name of a DLL library function that is to be called.
- ✓ `pCallData` – pointer to buffer containing data to be transferred to the called function or `NULL`.
- ✓ `callDataSize` – data size (in bytes) pointed by `pCallData` or `0` if `pCallData == NULL`.
- ✓ `pResData` – pointer to buffer in which the results of the function will be placed or `NULL` if no result is expected.
- ✓ `pMaxResDataSize` – pointer to an `int` type variable which contains the size of the buffer pointed by `pResData`. This size determines the maximum number of bytes of the expected result. The parameter should be `NULL` if no result is expected. After the function call, the variable `*pMaxResDataSize` will contain the actual number of bytes copied to `pResData` buffer.

Result

The function returns `TAPI_SUCCESS` for success or one of the error codes:

- ✓ `TAPI_NOTCONN` – no network connection to the terminal.
- ✓ `TAPI_BADPARAMS` – bad call parameters.
- ✓ `TAPI_RESULTTOLARGE` – the result buffer is too small.
- ✓ `TAPI_NOFUNCTION` – the called function can not be found in module `hRemoteModule`.

Description

The function allows a synchronous call of a users function residing on the terminal and attached to *gte.exe* as a DLL library. Before a call the library containing the function has to be loaded to the *gte.exe* address space using the `TApiRemoteLoadLibraryEx()`function. In a synchronous call the control is transferred to the RPC function and the parent process awaits its result. The result is returned in the `pResData` buffer, its size is limited in advance by the buffer size. The functions called using `TApiSyncRPC()` have to be properly exported from a DLL library and declared with a call type `cdecl`. The returned type and the types of the parameters have to be consistent with the `SYNCRPCFUN` type defined in *gtrmapi.h*. The header of the function not including the export directive and the call convention should be as follows:

```
void MySyncRPCFun(  void *pCallData,
                    int callDataSize,
                    void *pResData,
                    int *pMaxResDataSize  );
```

Before returning to the parent process the RPC function should copy up to `*pMaxResDataSize` bytes of the result to `pResData` buffer and place the actual number of bytes copied to the buffer to the `*pMaxResDataSize` variable. The contents of the buffer and the information about its size is transferred back to the calling function.
An example of a synchronously called users function is the function `UserSyncRPCMsgBoxCallback()` from the *gteext.dll* library.
An example of its call can be found in the program *testcapi.exe*.

50. TApiSyncRPC_VSR

Syntax

```
int TApiSyncRPC_VSR( HREMMODULE hRemoteModule,
                     WCHAR *pFunName,
                     void *pCallData,
                     int callDataSize,
                     void **ppResData,
                     int *pResDataSize );
```

Parameters

- ✓ `hRemoteModule` – handle of a remote DLL library obtained from the function `TApiRemoteLoadLibraryEx()`.
- ✓ `pFunName` – unicode name of a DLL library function that is to be called.
- ✓ `pCallData` – pointer to the buffer containing data to be transferred to the called function or `NULL`.
- ✓ `callDataSize` – data size (in bytes) pointed by `pCallData` or `0` if `pCallData == NULL`.

- ✓ `ppResData` – pointer to a `void*` type variable, the pointer to the buffer with the results of the RPC function will be placed in, or `NULL` if the function will return no result.
- ✓ `pResDataSize` – pointer to an `int` type variable initialized to `0`. After the function returns, this variable will contain the size of the result buffer pointed by `*ppResData`.

Result

The function returns `TAPI_SUCCESS` for success or one of the error codes:

- ✓ `TAPI_NOTCONN` – no network connection to the terminal.
- ✓ `TAPI_BADPARAMS` – bad call parameters.
- ✓ `TAPI_NOFUNCTION` – the called function can not be found in module `hRemoteModule`.

Description

The function allows a synchronous call of a user's function residing on the terminal and appended to *gte.exe* as a DLL library. Before a call, the library containing the function has to be loaded to the *gte.exe* address space using the `TApiRemoteLoadLibraryEx()`function. In case of a synchronous call the control is transferred to the RPC function and the parent process awaits its result. The result is returned in the `*ppResData` buffer allocated by the function `TApiSyncRPC_VSR()`. The difference between the functions `TApiSyncRPC_VSR()` and `TApiSyncRPC()` is that the first one allows the return from the RPC and acceptance by the caller of a result of any size, while the second one limits the result size in advance. Whenever the size of the result of an RPC function is known, the `TApiSyncRPC()`function call should be used.

VERY IMPORTANT!!!

Since the result buffer is dynamically allocated by the `TApiSyncRPC_VSR()`function, it is necessary to release it after use. The application must release the result buffer using the function `TApiMemGlobalFree()`. The use of another function will cause a memory protection error (GPF) or other problems.

The functions called using `TApiSyncRPC_VSR()` have to be properly exported from a DLL library and declared with a call type `cdecl`. The returned type and the types of the parameters have to be consistent with the `SYNCRPC_VSRFUN` type defined in *gtrmapi.h*. The header of the function not including the export directive and the call convention should be as follows:

```
void MySyncRPCFun(  void *pCallData,
                    int callDataSize,
                    void **ppResData,
                    int *pResDataSize  );
```

Before returning to the parent process, the RPC function running on the terminal (e.g. `MySyncRPCFun`) should allocate an appropriate memory buffer using `GteApiMemGlobalAlloc()` and copy to it the result of the function call. The pointer to the result buffer should be placed in `*ppResData`, and the actual size of the result should be saved as `*pResDataSize`. The contents of the buffer and the information about its size will be transferred back to the caller. The buffer allocated in the RPC function will be released by *gte.exe*. The described sequence should be as follows:

```
// placing result in buffer pointed by *ppResData
*ppResData = GteApiMemGlobalAlloc(RESULT_SIZE);
if( *ppResData )
{
  *pResDataSize = RESULT_SIZE; // returning result size
}
else
{
  *pResDataSize = 0; // no result
}
```

The example of a synchronously called user's function with a variable result size is the function `UserSyncRPCRandomReplicateCallback()` from the *gteext.dll* library. The example of its call can be found in the program *testcapi.exe*.

51. TApiTerminalMode

Syntax

```
int TApiTerminalMode( void );
```

Result

The function returns `1` if the application is executed in the terminal mode or `0` otherwise.

Description

The function can be used to check, if the application is executed in the terminal mode. It can be called before the initialization of the application interface with the `TApiInitialize()`function. This makes it easy to develop applications which will take advantage of the Winflector extensions in the terminal mode, but run also in the non-terminal mode. The `TApiTerminalMode()` function will very often be the first API function called by an application. If the application is executed in the terminal mode, the application interface will be then initialized by calling `TApiInitialize()` function.

IV. *gte.exe extension interface functions (GteApi)*

The available *gte.exe* extension interface functions are described below in alphabetical order. The names of the functions start with the `GteApi`... prefix. The functions are contained in the *gteapi.dll* library. An import library *gteapi.lib* is available. Function prototypes and all required types and constants are defined in the *gteapi.h* and *gtrmapi.h* header files. All text parameters are passed to/from interface functions as UNICODE. The *gte.exe* extension interface functions are to be used from the level of DLL libraries that contain RPC functions and extend the functionalities of the standard *gte.exe*. The extension libraries can (but do not have to) use the extension interface functions. The interface has to be initialized before its functions are used by calling `GteApiInitialize()`. Following functions are exceptions and can be used before the interface is initialized:

- ✓ `GteApiInitialized()`
- ✓ `GteApiGetApiVersion()`

52. GteApiCheckConnected

Syntax

`int GteApiCheckConnected( void );`

Result

The function returns 1 if there is a connection between *gte.exe* and the application or `0` if there is no connection.

Description

The function verifies if there is a connection between *gte.exe* to which the calling DLL extension library is attached, and the application which called the RPC function.

53. GteApiGetApiVersion

Syntax

```
int GteApiGetApiVersion( GTRMVERSION *pApiVer );
```

Parameters

- ✓ `pApiVer` – pointer to `GTRMVERSION` structure which will receive the result.

Result

The function fills the `GTRMVERSION` structure and returns `0`.

Description

The function returns in the `GTRMVERSION` structure information about the extension interface version, that is the *gteapi.dll* library. The version should be identical or older than version of *gte.exe*. If this is not the case, the API initialization using `GteApiInitialize()` will fail. This means gte.exe must be identical or newer then *gteapi.dll used.*

54. GteApiGetGteVersion

Syntax

```
int GteApiGetGteVersion( GTRMVERSION *pCallerVer,
                         GTRMVERSION *pGteVer );
```

Parameters

- ✓ `pCallerVer` – reserved – should always be `NULL`.
- ✓ `pGteVer` – pointer to `GTRMVERSION` structure which will receive the result.

Result

The function returns `TAPI_SUCCESS`.

Description

The function allows reading of the version of the *gte.exe* to which the extension library is attached.

55. GteApiGetSrvInfo

Syntax

`int GteApiGetSrvInfo( GTEAPI_TRMSVINFO *pSrvInfo );`

Parameters

- ✓ `pSrvInfo` – pointer to `GTEAPI_TRMSVINFO` structure which will receive the result.

Result

The function returns `TAPI_SUCCESS`.

Description

The function allows reading of the version of the operating system on the application server, the version of the Winflector server and the application start date.

56. GteApiHwndToNetId

Syntax

`int GteApiHwndToNetId( HWND hWnd );`

Parameters

- ✓ `hWnd` – handle of one of the *gte.exe* windows corresponding to the application windows.

Result

A network identifier of a window or `0`.

Description

The function allows converting of a *gte.exe* window handle which corresponds to one of the main windows of the application into its network identifier. The network identifier can be returned from the RPC function to the application and converted there into the corresponding window of the server system using the `TApiNetIdToHwnd()`function. The function returns `0` if `hWnd` is not a handle of a window created by *gte.exe*.

57. GteApiInitialize

Syntax

```
int GteApiInitialize( void );
```

Result

The function returns `TAPI_SUCCESS` or one of the following error codes:

- ✓ `TAPI_NOGTEEXE` – *gte.exe* not found – most probably the *gteapi.dll* library has been loaded to another process.
- ✓ `TAPI_BADAPIVERSION` – the first four digits of the *gte.exe* extension interface (*gteapi.dll*) version do not correspond to the first four digits of *gte.exe* version. Verify if correct DLL libraries have been copied.
- ✓ `TAPI_CANTIMPORTFUN` – the address of one of the API functions within *gte.exe* can not be resolved.

Description

The function initializes internal structures of the *gte.exe* extension interface contained in the *gteapi.dll* library. During the initialization the *gteapi.dll* functions are linked with the auxiliary functions contained in *gte.exe*. The interface has to be initialized before the first call to any other API function. The two following functions are exceptions which can be called before API initialization:

- ✓ `GteApiInitialized()`
- ✓ `GteApiGetApiVersion()`

58. GteApiInitialized

Syntax

`int GteApiInitialized( void );`

Result

The function returns `1` if the *gte.exe* extension interface has been already correctly initialized using `GteApiInitialize()` or `0` otherwise.

Description

The function can be used anytime within an RPC function to check if the extension interface has been initialized and if API functions can be used.

59. GteApiMemGlobalAlloc

Syntax

`void *GteApiMemGlobalAlloc( unsigned size );`

Parameters

- ✓ `size` – the size of the memory block that is to be allocated.

Result

The function returns the pointer to the allocated memory block of the size defined by `size` parameter of `NULL` if the memory can not be allocated.

Description

The function allows dynamical memory allocation for an RPC function. The allocated memory block is zero-initialized. The memory allocated using `GteApiMemGlobalAlloc()`has to be released using the `GteApiMemGlobalFree()`function. If the RPC function is called by `TApiSyncRPC_VSR()` and returns a non-zero result, the described function must be used to allocate the memory block to store the result.

60. GteApiMemGlobalFree

Syntax

`void GteApiMemGlobalFree( void *ptr );`

Parameters

- ✓ `ptr` – pointer to the memory block allocated using `GteApiMemGlobalAlloc()`.

Description

The function releases memory dynamically allocated by `GteApiMemGlobalAlloc()`.

61. GteApiNetIdToHwnd

Syntax

`HWND GteApiNetIdToHwnd( WNDNETID netid );`

Parameters

- ✓ `netid` – network identifier of a window obtained from `TApiHwndToNetId()`, `TApiHwndToRemotedNetId()` or `GteApiHwndToNetId()`functions.

Result

Handle of a Windows window created by *gte.exe* or `NULL`.

Description

The function allows converting the network identifier of a window to the corresponding window (of the Windows system) created by *gte.exe*. The network identifier of a window will be in most cases transferred to the RPC function from an application. The application function can convert on the server the actual handle of the Windows system window into the network identifier using `TApiHwndToNetId()`or `TApiHwndToRemotedNetId()` and transfer it to the RPC function. The RPC function can find the terminal window which corresponds to the server window by converting the network identifier into the terminal window handle using the described function. The function returns `NULL` if the window described by the network identifier does not exist anymore.

62. GteApiRaiseAsyncAppEvent

Syntax

```
int GteApiRaiseAsyncAppEvent(   void *pClntData,
                                DWORD dataSize );
```

Parameters

- ✓ `pClntData` – pointer to data, which will be passed to application with WFLEVT_ASYNCGTEEVENT event in `clntData` field of GTEEVTINFO structure. Can be NULL.
- ✓ `dataSize` –number of bytes of data pointed by `pClntData` pointer. This info application will receive in `clntDataSize` field of GTEEVTINFO structure. Can be 0.

Result

TAPI_SUCCESS – event sent to application
TAPI_NOTCONN – error – not connected to application

Description

Functions allows user client extention DLL for gte.exe to raise an event that will be transferred to application and presented to application as WFLEVT_ASYNCGTEEVENT event by calling user function registered via `TapiRegisterAppCallback.` This function can be called in any thread context but user-registered function will always be called in main application thread context. If `pClntData` is not NULL, than `dataSize` of pointed bytes will be passed to application. Function returns immediately after sending event to application.
For example, in registration desk system client station can monitor incoming telephone calls, and if one arrives execute `GteApiRaiseAsyncAppEvent()`passing the telephone number as parameters. When application receives WFLEVT_ASYNCGTEEVENT event, it can display the window with the data of the calling customer.

63. GteApiRaiseFinalError

Syntax

```
int GteApiRaiseFinalError( wchar_t *pDescription,
                           wchar_t *pFunction,
                           int ival1,
                           int ival2 );
```

Parameters

- ✓ `pDescription` – error description as a unicode string.
- ✓ `pFunction` – unicode name of the function which had an error.
- ✓ `ival1`, `ival2` – additional diagnostics values which will be saved and displayed with the error message.

Result

The function does not return to the caller, thus it does not return any value.

Description

The function allows rising of the termination error. Following actions are part of the error handling:

- The error message and other error information is saved to the log file on the terminal. The log file is located in Client\logs directory and is called *tcllog.txt*.
- If there is a connection between *gte.exe* and the application, the error `"GTE shutdown notification received - quiting"` is written to the application log (*gapplog.txt* in *applogs* subdirectory of the Winflector server) and the application process is terminated.
- A forced termination of the *gte.exe* process with an exit code larger than 0 takes place.

As seen from the description above the function concludes the execution of the program. The control is not returned to the caller and the function does not return any value.

64. GteApiRaiseSyncAppEvent

Syntax

```
int GteApiRaiseSyncAppEvent(    void *pClntData,
                               DWORD dataSize,
                               DWORD *pResult );
```

Parameters

- ✓ `pClntData` – pointer to data, which will be passed to application with WFLEVT_SYNCGTEEVENT event in `clntData` field of GTEEVTINFO structure. Can be NULL.
- ✓ `dataSize` –number of bytes of data pointed by `pClntData` pointer. This info application will receive in `clntDataSize` field of GTEEVTINFO structure. Can be 0.
- ✓ `pResult` – pointer to memory that will receive DWORD value returned by application callback function. Can be NULL.

Result

TAPI_SUCCESS – event sent to application
TAPI_NOTCONN – error – not connected to application
TAPI_BADTHREAD – function cannot be called in the context of this thread

Description

Functions allows user client extention DLL for gte.exe to raise an event that will be transferred to application and presented to application as WFLEVT_SYNCGTEEVENT event by calling user function registered via `TapiRegisterAppCallback.` This function can be called only by threads originally created by gte.exe (including main gte.exe thread). User-registered function will always be called in main application thread context. If `pClntData` is not NULL, than `dataSize` of pointed bytes will be passed to application. Function waits until application process the event, then returns in `*pResult` value that was returned by application callback function.

Because of the existing thread restrictions (see above), we suggest to always use asynchronous events (`GteApiRaiseAsyncAppEvent`()). If necessary, after receiving asynchronous event, application can always call TApiSyncRPC() function to continue communication and event handling.

65. GteApiSetConsoleEventMask

Syntax

`unsigned GteApiSetConsoleEventMask( unsigned newMask );`

Parameters

- ✓ `newMask` – a new mask which defines the handling of console events. The mask can contain `KEY_EVENT` and `MOUSE_EVENT` (*wincon.h*) bits in any combination.

Result

The previous value of the mask defining the handling of console events.

Description

The use of the function is appropriate for applications executed in the Windows console mode only. The function allows selective blocking and unblocking of some console events. By default the console handles both keyboard events (`KEY_EVENT`) and mouse events (`MOUSE_EVENT`). By sending a mask one can set the handling of a given event type to either handling on (bit on) or handling blocked (bit zeroed). Handled events are transferred to the application. Blocked events are ignored and thus not sent to application.