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

Terminal GUI
Terminal Console
V. 2.4

Programmers manual

Table of contents

I. Introduction

The main task of Terminal GUI and Terminal 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 Terminal GUI/Console is outfitted with a set of interfaces and libraries which make it possible to further integrate the applications with the Terminal 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

Terminal GUI 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 Terminal environment. An extension of capabilities for applications working in terminal mode is accomplished by using additional Terminal functionality accessible through the *application interface*. The application interface allows calls to two classes of functions:

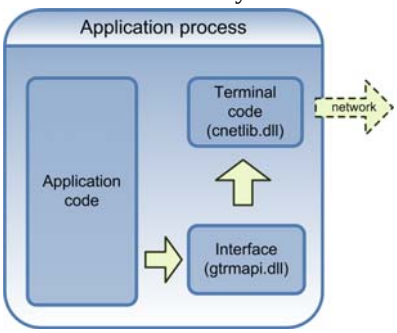
1. built-in functions,
2. users functions attached on the terminal side (RPC).

Using the *application interface*

The application interface consists of the *gtrmapi.dll* library containing functions that offer access to extended functionality of the Terminal. 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 Terminal 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 Terminal 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.

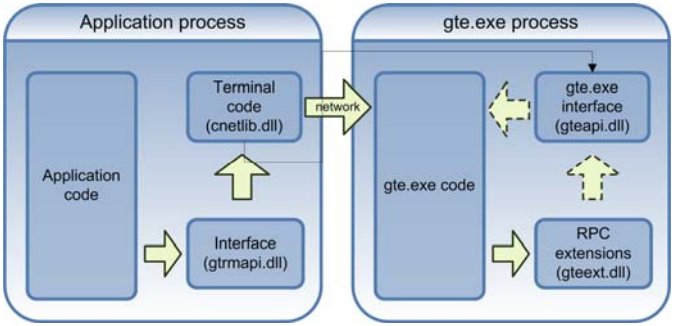


An example of the application which uses the described application interface is *testcapi.exe*.
A detailed description of the application interface functions can be found in chapter III, page 13.

Remote procedure call (RPC)

In the Terminal GUI 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 .

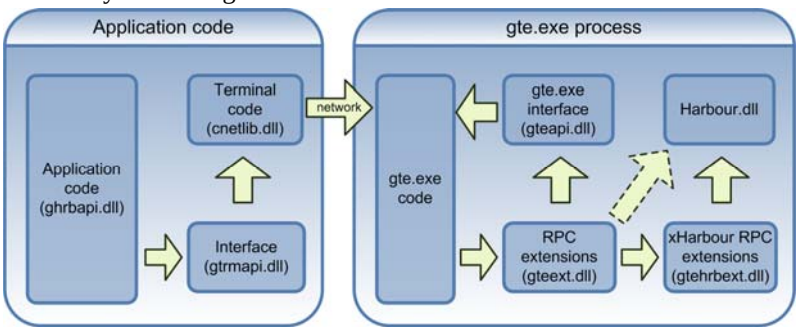
Using the *application interface* in xHarbour application

In order to facilitate the use of the application interface from the xHarbour language level we have created the *ghrbapi.lib* library. It contains xHarbour functions that call functions from the *gtrmapi.dll* library (wrappers). Apart from wrappers to interface functions, *ghrbapi.lib* contains also functions providing backward compatibility with OTC Terminal functions, such as `TrmTrmRPC()`, `TrmPutFile()`, `TrmFlPrint()`, and others. Implementation of older functions makes the migration of applications to the new Terminal GUI environment much easier.
An example xHarbour application, which takes advantage of the application interface, is the program *testhbapi.exe*.
A detailed description of xHarbour functions contained in the *ghrbapi.lib* library can be found in chapter V, page 48.

Remote procedure call (RPC) in xHarbour applications

The use of remote procedure call from the xHarbour application level has been simplified by introducing the *ghrbapi.lib* library which can be linked into the applications. The library contains the `TrmTrmRPC()` function and allows to create extension libraries in the xHarbour language.
An additional extension library *gtehrb.dll* has been created to allow direct RPC call of xHarbour functions. It features a special mode of procedure call required for the xHarbour language. Both standard xHarbour functions and procedures included in *harbour.dll* as well as users functions/procedures contained in a DLL library can be called using the `TrmTrmRPC()` function. An example of an extension library written in xHarbour is the *gtehrbext.dll* library, an example of its use can be found in application *testhbapi.exe*.
The figure below presents an xHarbour application process on the server and a *gte.exe* process running on a terminal. The arrows illustrate calls between respective elements of the processes during a remote procedure call (RPC) of an xHarbour procedure from the xHarbour level. It can be noted that the extension code from the users library (*gtehrbext.dll*) is called via the standard extension library *gtehrb.dll*. In

case of a users procedure call, *gtehrb.dll* calls to the *gtehrbext.dll*. In case of a standard function call (e.g. SQRT), *gtehrb.dll* makes a call to *harbour.dll* (dashed arrow). Either for a standard or users function call, the *harbour.dll* library has to be present as the runtime environment. The *gtehrbext.dll* library is not required if only standard functions contained in *harbour.dll* are to be called. Declaration of alternative names for *harbour.dll* and *gtehrbext.dll* is possible at the initialization of the RPC environment for xHarbour (`THbApiInitialize(.T.)` function). Only one extension library containing xHarbour code is allowed.



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. 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*

3. 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 agree within the first four digits with the version of the used Terminal. If this is not the case, the API initialization using `TApiInitialize()` will fail.

4. 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.

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.

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

5. TApiGetExpirationDate

Syntax

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

Result

The function returns the date of expiry of the evaluation version of the Terminal 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 Terminal 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.

6. 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.

7. 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.

8. 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.

9. 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.

10. 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.

11. 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.

12. 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.

13. 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.

Description

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

14. 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 Terminal 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 Terminal server user who has launched the application.

15. 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 Terminal server.

16. 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.

17. 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.

18. 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 Terminal 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()`

19. 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.

20. 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.

21. 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.

22. 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.

23. 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.

24. 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 Terminal 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 (*gtelog.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.

25. 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()`.

26. 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.

27. 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 `@ERATERM` 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.

28. 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 Terminal GUI environment the changes made by the application in windows of the Windows system are sent to the terminal with a 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.

29. 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.

30. 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*.

31. 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*.

32. 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 Terminal 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()`

33. 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.

34. 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 agree within the first four digits with the version of *gte.exe*. If this is not the case, the API initialization using `GteApiInitialize()` will fail.

35. 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.

36. 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 Terminal server and the application start date.

37. 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*.

38. 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()`

39. 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.

40. 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.

41. 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()`.

42. 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.

43. 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 the *gte.exe* directory and is called *gtelog.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 Terminal 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.

44. 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.

V. Application interface xHarbour functions (THbApi)

The functions available directly to xHarbour applications are described below. The names of the functions start with the prefix `THbApi`... or `Trm`... . Compiled functions are located in the *ghrbapi.lib* library which has to be attached when linking application. Described first are the new functions (`THbApi`) followed by `Trm`... functions. The `Trm`... functions are included to assure interface compatibility with earlier versions of the OTC Terminal software designed for xHarbour/Clipper. Before using the interface functions the interface must be initialized using `THbApiInitialize()`. The function `THbApiTerminalMode()` is an exception and can be used before interface initialization. After finishing the use of the interface the function `THbApiShutdown()` must be called.

45. THbApiGetClientDir

Syntax

`THbApiGetClientDir( <nDirectoryType> ) -> cResult`

Result

The function returns a directory (path) on the terminal or an empty string in case of an error.

Description

The function allows to obtain a directory name on the terminal defined by the `nDirectoryType` parameter. At this time the following directory types are supported:

- `TAPI_DIRTYPE_GTE (1)` – the *gte.exe* program directory.

46. THbApiInitialize

Syntax

```
THbApiInitialize( [<lInitRPC>] [,<cHarbourDllPath>
                  [,<cExtentionDllPath>] ] )-> nResult
```

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 has not been launched in the terminal mode.
- ✓ `TAPI_BADPARAMS` – bad parameters were given.
- ✓ `TAPI_BADAPIVERSION` – the first four digits of the application interface version (*gtrmapi.dll*) do not correspond to the first four digits of the Terminal software version (*cnetlib.dll*). Please verify if correct DLL libraries have been copied.
- ✓ `TAPI_CANTIMPORTFUN` – the address of one the the API functions from within the *cnetlib.dll* library could not be resolved.
- ✓ `TAPI_CNTLOAD_GTEHRBDLL` – unable to load *gtehrb.dll* on the terminal side.
- ✓ `TAPI_CNTLOAD_HARBOURDLL` – unable to load *harbour.dll* on the terminal side.
- ✓ `TAPI_BADHARBOURDLLVER` – unable to load *harbour.dll* on the terminal side due to uncombatible xHarbour versions of *harbour.dll* and *gtehrb.dll*.

Description

The function initializes internal structures of the xHarbour application interface contained in *ghrbapi.lib*. If the `lInitRPC` parameter is `.T.`, the RPC subsystem which allows calls to remote xHarbour procedures is also initialized. Following actions are part of the initialization:

- The initialization of the base application interface (*gtrmapi.dll*) by calling `TApiInitialize()`.
- Actions below are executed only if `lInitRPC` is `.T.`

- Loading of the remote extension library: *gtehrb.dll*.
- Loading of the xHarbour remote library: *harbour.dll* or `cHarbourDllPath` if specified.
- Loading of the remote extension library: *gtehrbext.dll* or `cExtentionDllPath` if specified.

A missing extension library is not considered to be an error. The `THbApiRPCExtInitialized()` function can be used to check if the extension library has been loaded.

47. THbApiInitialized

Syntax

```
THbApiInitialized( ) -> lResult
```

Result

The function returns `.T.` if the xHarbour application interface has been already initialized by calling `THbApiInitialize()` or `.F.` otherwise.

Description

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

48. THbApiRPCInitialized

Syntax

```
THbApiRPCInitialized( ) -> lResult
```

Result

The function returns `.T.` if the xHarbour application interface has been already initialized with the RPC call option or `.F.` otherwise.

Description

The function can be used anytime within an application to check if the application interface has been initialized and if RPC function calls can be made.

The function returns `.T.` if the interface has been initialized and the *harbour.dll* library has been loaded on the terminal with the RPC call option.

49. THbApiRPCExtInitialized

Syntax

```
THbApiRPCExtInitialized( ) -> lResult
```

Result

The function returns `.T.` if the xHarbour application interface has been already initialized with the RPC call option and the extension library *gtehrbext.dll* has been loaded on the terminal side or `.F.` otherwise.

Description

The function can be used anytime within an application to check if the application interface has been initialized and if RPC calls and users extensions can be used. The function returns `.T.` if the interface has been initialized and the *harbour.dll* library and the *gtehrbext.dll* users extension library have been loaded on the terminal side.

50. THbApiShutdown

Syntax

```
THbApiShutdown( ) -> nResult
```

Result

The function returns `TAPI_SUCCESS` or one of the error codes of the `TApiSyncRPC()` function.

Description

The function allows to release resources occupied by the xHarbour application interface when it is not needed anymore. Depending on the initialization option following actions take place:

- Release of the xHarbour remote extension library: *gtehrbext.dll*.

- Release of the xHarbour remote library: *harbour.dll*.
- Release of the remote extension library: *gtehrb.dll*.

ATTENTION!

The application interface functions can still be used from the C/C++ language level using `TApi`... functions from the *gtrmapi.dll* library.

51. THbApiTerminalMode

Syntax

`THbApiTerminalMode( ) -> lResult`

Result

The function returns `.T.` if the application is executed in the terminal mode or `.F.` 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 xHarbour application interface with the `THbApiInitialize()` function. This makes it easy to develop applications which will take advantage of the Terminal extensions in the terminal mode but run also in the non-terminal mode. The `THbApiTerminalMode()` function will very often be the first API function called by the application. If the application is executed in the terminal mode, the application interface will be then initialized by calling the `THbApiInitialize()` function.

52. TrmAppOS

Available in the 32-bit version only (Harbour/xHarbour).

Syntax

`TrmAppOS() -> nAppSystemID`

Description

The function returns the code of the operating system on which the application is executed.

Returned codes:

2 – Windows NT
6 – Windows 2000
7 – Windows XP
8 – Windows 2003
32 – Linux

See also: `TrmTeOS()`

53. TrmDiscTm

Syntax

`TrmDiscTm( <nDisconnectTimeout> )`

Description

The function allows to set the maximum period of 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 `nDisconnectTimeout` parameter defines the time in seconds. It can adopt values from 20 to 10000.

The function with the parameter 0

`TrmDiscTm(0)`

restores the default disconnect time, while

`TrmDiscTm(65535)`

switches off the disconnect mechanism altogether.

54. TrmFlPrint

Syntax

`TrmFlPrint( <cFileToPrint>, <nLPTnumber> ) -> nResult`

Description

The function prints out a file seen by the xHarbour application executed on the server as `cFileToPrint` on a printer attached to the terminal station. The parameter `nLPTNumber` defines the LPT port number (1, 2, or 3) on the server. Depending on the active redirect the file will be printed on the corresponding LPT port of the terminal station.
The function returns `0` in case of success. Otherwise it returns an error code (<1000) identical with the code returned by the `FERROR()` function. If the error occurred on the terminal while printing, function returns the number of correctly printed characters increased by 1000.

Example

```
Res = TrmFlPrint( ”myfile.txt”, 1 )
IF res == 0
   ? “Printed OK.”
ELSIF res < 1000
   ? “File access error (invalid file name?)”
ELSE
   ? “Error printing file, printed“, res-1000,“bytes”
ENDIF
```

55. TrmGetFile

Syntax

```
TrmGetFile( <cTermFileName>, <cNTFileName> )-> nResult
```

Description

The function transfers the file seen by the *te.exe* application, executed on the terminal, as `cTermFileName` onto the application server and saves it as `cNTFileName`. The function returns `0` in case of success. Otherwise it returns an error code identical with code returned by the `FERROR()` function. If the error occurred on the terminal, the error code is increased by 1000.

56. TrmGetPrty

Syntax

```
TrmGetPrty() -> nCurrentPriority
```

Description

The function returns a value which corresponds to the system priority of the application executed on the Windows server. Possible `nCurrentPriority` values are:

- `0` – low priority
- `1` – normal priority (default value)
- `2` – high priority
- `3` – highest priority

57. TrmIsTs

Syntax

```
TrmIsTs() -> lResult
```

Description

The mechanism of terminal transactions is not implemented in the Terminal GUI software. This function has been preserved to ensure backward compatibility of the existing code. Function returns `.T.` after call to if `TrmTsBegin()` and `.F.` after call to `TrmTsEnd()` function.

58. TrmPrCancl

Syntax

```
TrmPrCancl( <nPrinterHandle> ) -> lResult
```

Description

The function cancels printing of a file sent to the printer identified by `nPrinterHandle` using the `TrmPrSubmt()` function. The function returns `.T.` if printing has been cancelled or `.F.` otherwise.

59. TrmPrClose

Syntax

`TrmPrClose( <nPrinterHandle> )`

Description

The function closes the printer session identified by `nPrinterHandle`. The closing of a session does not remove any data sent earlier using the `TrmPrSubmt()` function from the printer queue.

60. TrmPrFile

Syntax

`TrmPrFile( <cPrinterName>, <cFileName> ) -> lResult`

Description

The function prints the `cFileName` file on the `cPrinterName` printer. The name of the printer should be identical with the name returned by the `TrmPrList()` function. The function returns `.T.` if the task has been successfully completed or `.F.` otherwise. The function does not require an open printer session.

61. TrmPrList

Syntax

`TrmPrList( ) -> aPrnTable`

Description

The function returns a table containing names and descriptions of all printers defined in the Windows NT/200x/XP/Vista system or `NIL` if there are no printers defined. The length of the returned table equals to the number of printers in the system. For every printer two string values are returned:

- ✓ name - `aPrnTable[printerNumber][1]`.
- ✓ description - `aPrnTable[printerNumber][2]`.

The name of the printer can be used when calling the `TrmPrOpen()` or `TrmPrFile()`function.

Example

```
pTab = TrmPrList()
IF pTab == NIL
  ? ”No printer(s) defined”
  QUIT
ENDIF
FOR pn = 1 TO LEN(pTab)
  ? ”Printer name: ”, pTab[pn][1]
  ? ”Printer description: ”, pTab[pn][2]
NEXT
```

The sample fragment of the code above prints out names and descriptions of all printers defined in the Windows NT system.

62. TrmPrOpen

Syntax

```
TrmPrOpen( <cPrinterName> ) -> nPrinterHandle
```

Description

The function opens a printer session on the `cPrinterName` printer and returns its handle or `0` if the session could not be opened. The returned session handle should be passed to all functions that require it. The name of the printer should be as returned by the `TrmPrList` function.

63. TrmPrPut

Syntax

```
TrmPrPut( <nPrinterHandle>,
          <cStringToPrint>,
          <nStringLength> ) -> lResult
```

Description

The function sends the number of bytes defined by `nStringLength` from the string `cStringToPrint` to the session identified by `nPrinterHandle`. The function returns `.T.` if sending of the data was successful or `.F.` otherwise.

64. TrmPrPutFl

Syntax

`TrmPrPutFl( <nPrinterHandle>, <cFileName> ) -> lResult`

Description

The functions sends the contents of the file defined by `cFileName` to the session `nPrinterHandle`. The function returns `.T.` if sending of the data was successful or `.F.` otherwise.

65. TrmPrSubmt

Syntax

`TrmPrSubmt( <nPrinterHandle> ) -> lResult`

Description

The function sends data prepared earlier with the `TrmPrPut()` and `TrmPrPutFl()` functions to a printer queue associated with the `nPrinterHandle` session. The function returns `.T.` if sending of the data was successful or `.F.` otherwise.

66. TrmPutFile

Syntax

`TrmPutFile( <cNTFileName>, <cTermFileName> ) -> nResult`

Description

The function sends a file seen by the xHarbour application executed on the Windows NT/200x/XP/Vista server as `cNTFileName` to the terminal and saves

it as file `cTermFileName.` The function returns `0` in case of success. Otherwise it returns an error code identical with code returned by the `FERROR()` function. If the error occurred on the terminal, the error code is increased by 1000

67. TrmSetPrty

Syntax

`TrmSetPrty( nNewPriority )`

Description

The function allows to change the execution priority of a terminal application on a Windows NT server. Following new `nNewPriority` settings are possible:

- `0` – low priority
- `1` – normal priority (default value)
- `2` – high priority
- `3` – highest priority

In case of a large number of active terminal sessions the `TrmSetPrty()` can be used to decrease the run time priority of an application for long and CPU intensive tasks (e.g. creating of reports and summaries). Decreasing the execution priority of parts of an application allows to maintain efficient execution of interactive processes even for a very large number of terminal sessions running.

68. TrmSvName

Syntax

`TrmSvName( ) -> nServerName`

Description

The function returns the name of Windows NT server on which the application is running or NIL.

69. TrmTeOS

Syntax

`TrmTeOS() -> nTeSystemID`

Description

The function returns the code of the operating system on which the *gte.exe* (terminal client) is executed.
Operating system codes returned:

1 - DOS
2 – Windows NT
3 – Windows 95
4 – Windows 98
5 – Windows Me
6 – Windows 2000
7 – Windows XP
8 – Windows 2003
32 - Linux

See also: `TrmAppOS()`

70. TrmTrmRPC

Syntax

`TrmTrmRPC( <cTermFunName>, ... ) -> Result`

Description

The function called by an application will execute on the terminal the extension function `cTermFunName` contained in the DLL extension library. Parameters specified after the function name are passed on to the remote function. The function returns the value returned by the remotely executed function. Remote functions can not receive nor return any array type arguments or arguments

passed by reference (@). Before calling this function it is necessary to initialize the interface using `THbApiInitialize()`in the RPC mode, thus specifying the `.T.` parameter.

Example

On the application side:

```
? TrmTrmRPC( ”FUN1”, 1, 2 )
```

On the terminal side:

```
FUNCTION FUN1
PARAMETERS P1, P2
RETURN P1+P2
```

Execution of this program will display the result value 3, which would be calculated on the terminal.

ATTENTION!

If the `TrmTrmRPC()` function is called after the connection between the terminal and the application has been lost, `NIL` value will be returned.

71. TrmTSBegin

Syntax

`TrmTSBegin()`

Description

The mechanism of terminal transactions is not implemented in the Terminal GUI software. This function has been preserved to ensure backward compatibility of the existing code. After calling `TrmTsBegin()`, `TrmIsTs()` function will return `.T.`

72. TrmTSEnd

Syntax

`TrmTSEnd()`

Description

The mechanism of terminal transactions is not implemented in the Terminal GUI software. The function has been preserved to ensure backward compatibility of the existing code. After calling `TrmTsEnd()`, `TrmIsTs()` function will return `.F.`

73. TrmUser

Syntax

```
TrmUser() -> cTermUserName
```

Description

The function returns the user name of the terminal user (obtained while connecting to the Terminal server).

74. TrmUpdate

Syntax

```
TrmUpdate()
```

Description

The function forces immediate sending to the terminal of all display changes buffered by the application.

VI. Migration of xHarbour applications to Terminal GUI environment

No migration is necessary for applications that work with the older, specialized for xHarbour/Clipper, version of OTC Terminal, which will be used in text mode only. Those applications can be launched using *te32.exe* after a standard adjustment procedure which links them to new versions of Terminal libraries.
The migration procedure described in this section refers to such modifications of an application that will make it work in the GUI environment. It is possible to create a single EXE file which will work both in the terminal environment and as a standalone windows application.

The *te32.exe* migration

If the migrating application uses *te32.exe* with users extensions attached (the RPC functions), it is necessary to transfer them over to a DLL library which can be loaded using *gte.exe*. An example of such a library is *gtehrbext.dll*. This library can be easily modified to include functions required for the migration. In order to do that one has to:

1. Add to the *mkhbextdll.bat* script the compilation of all PRG files containing the required functions.
2. At the end of *gtehrbext.c*, add the command to include *.c files created by compilation of the PRG files mentioned in 1.
3. Create a DLL library containing the required functions by executing the *mkhbextdll.bat* script.
   To facilitate the migration of applications with extended *te.exe* and *te32.exe* functionalities, the *gtehrblib.lib* library containing xHarbour functions traditionally available in *te/te32.exe* can be attached to the *gtehrbext.dll* extension library. The xHarbour functions traditionally available in the *te/te32.exe* environement are the following: `TrmCltVMaj()`, `TrmCltVMin()`, `TrmCltVSub()`, `TrmCltVBfx()`, `TrmSrvSys()`, `TrmSrvVMaj()`, `TrmSrvVMin()`, `TrmTSVMaj()`, `TrmTSVMin()`, `TrmTSVSub()`,

`TrmTSVBfx()`, `TrmSrvYear()`, `TrmSrvMnth()`, `TrmSrvDay()`, `TrmRcvBts()`, `TrmPrList()`, `TrmPrOpen()`, `TrmPrClose()`, `TrmPrPut()`, `TrmPrPutFl()`, `TrmPrFile()`, `TrmPrSubmt()`, `TrmPrCancl()`, `TrmSvName()`, `TrmTeOS()`, `TrmAppOS()`, `TrmRKBOff()`, `TrmRKBOn()`.

4. Copy the created library to the *gte.exe* directory.

The procedure described above is not required if the application uses a standard *te32.exe* version.

The application migration

If the application does not use any functions of the Terminal (`Trm`...) package, it is sufficient to exclude from the application linking process any libraries related to the Terminal. The created application can be launched as a standalone process or in the terminal mode using *gte.exe*.
If the application uses functions of the Terminal package, following steps have to be followed:

1. Remove from the application linking process any old Terminal libraries.
2. Add to the application linking process the *ghrbapi.lib* and *gtrmapi.lib* libraries.
3. Place the following call at the beginning of the application (before any terminal function call):

   `THbApiInitialize(.T.)`

   (see the description of the `THbApiInitialize()` function for details).

4. Place the following call before the end of the application:

   `THbApiShutdown()`

   (see the description of the `THbApiShutdown()` function for details).

5. Create the application EXE file.

An application created in this way can be launched in the terminal mode using *gte.exe*. If the application is to be used as a standalone process as well, it is worthwhile to create a single EXE file which can be executed both in the terminal and standalone mode. For that, one should use the `THbApiTerminalMode()` function which

checks if the application is executed in the terminal mode. All subsequent calls to Terminal functions should be made only if the application works in the terminal mode.